Садржај

Глава 1

Анализа коректности алгоритама

Делови текста у наставку су преузети из скрипте "Програмирање 2", аутора Предрага Јаничића и Филипа Марића.

Исправност тј. коректност је суштинска особина алгоритама и програма. Иако се некада у пракси користе програми за које се зна да понекад могу да дају и нетачне резултате, то најчешће није случај и од програма се захтева да буде практично апсолутно непогрешив.

Једно од централних питања у развоју програма је питање његове исправности (коректности). Софтвер је у данашњем свету присутан на сваком кораку: софтвер контролише много тога — од банковних рачуна и компоненти телевизора и аутомобила, до нуклеарних електрана, авиона и свемирских летелица. У свом том софтверу неминовно су присутне и грешке. Грешка у функционисању даљинског управљача за телевизор може бити тек узнемирујућа, али грешка у функционисању нуклеарне електране може имати разорне последице. Најопасније грешке су оне које могу да доведу до великих трошкова, или још горе, до губитка људских живота. Неке од катастрофа које су општепознате су експлозија ракете Ариане 1996. узрокована конверзијом броја из шездесетчетворобитног реалног у шеснаестобитни целобројни запис која је довела до прекорачења, затим пад сателита Криосат (енгл. Cryosat) 2005. године услед грешке у софтверу због које није на време дошло до раздвајања сателита и ракете која га је носила коштао је Европску Унију око 135 милиона евра, затим грешка у нумеричком копроцесору процесора Pentium 1994. узрокована погрешним индексима у петљи `for` у оквиру софтвера који је радио дизајн чипа, као и пад орбитера послатог на Марс 1999. узрокован чињеницом да је део софтвера користио метричке, а део софтвера енглеске јединице. Међутим, фаталне софтверске грешке и даље се непрестано јављају и оне коштају светску економију милијарде долара. Ево неких од најзанимљивијих:

- Не нарочито опасан, али веома занимљив пример грешке је грешка у програму Microsoft Excel 2007 који, због грешке у алгоритму форматирања бројева пре приказивања, резултат израчунавања израза `77.1 * 850` приказује као `100 000` (иако је интерно коректно сачуван).
- У Лос Анђелесу је 14. септембра 2004. године више од четиристо авиона у близини аеродрома истовремено изгубило везу са контролом лета. На срећу, захваљујући резервној опреми унутар самих авиона, до несреће ипак није дошло. Узрок губитка везе била је грешка прекорачења у бројачу милисекунди у оквиру система за комуникацију са авионима. Да иронија буде већа, ова грешка је била откривена раније, али пошто је до открића дошло када је већ систем био испоручен и инсталиран на неколико аеродрома, његова једноставна поправка и замена није била могућа. Уместо тога, препоручено је да се систем ресетује сваких 30 дана како до прекорачења не би дошло. Процедура није испоштована и грешка се јавила после тачно 2^{32} милисекунди, односно 49,7 дана од укључивања система.
- Више од пет процената пензионера и прималаца социјалне помоћи у Немачкој је привремено остало без свог новца када је 2005. године уведен нови рачунарски систем. Грешка је настала због тога што је систем, који је захтевао десетоцифрени запис свих бројева рачуна, код старијих рачуна који су имали осам или девет цифара бројеве допуњавао нулама, али са десне уместо са леве стране како је требало.
- Једна бака у Америци је на свој 106. рођендан добила позив да мора да крене у школу, јер је систем бележио године помоћу две цифре.
- Компаније Dell и Apple морале су током 2006. године да корисницима замене више од пет милиона лап-

топ рачунара због грешке у дизајну батерије компаније Sony која је узроковала да се неколико рачунара запали.

1.1 Облици испитивања коректности

У развијању техника верификације програма, потребно је најпре прецизно формулисати појам коректности тј. исправности програма. Исправност програма почива на појму **спецификације**. Спецификација је, неформално, опис жељеног понашања програма који треба написати. Спецификација се обично задаје у терминима **предуслова** тј. услова које улазни параметри програма задовољавају, као и **постуслова** тј. услова које резултати израчунавања морају да задовоље. Када је позната спецификација, потребно је верификовати програм, тј. доказати да он задовољава спецификацију.

Коректност се огледа кроз два аспекта:

парцијална коректност: свака вредност коју алгоритам израчуна за улазне параметре који задовољавају спецификацију (тј. предуслов) мора да задовољи спецификацију (тј. постуслов).

заустављање алгоритам мора да се заустави за све улазе који задовољавају спецификацију (тј. предуслов).

Већина алгоритама које ћемо проучавати у овом курсу биће потпуни тј. заустављање се за све допуштене улазе. За заустављајуће парцијално коректне алгоритме кажемо да су **тотално коректни**. Интересантно, доказано је да не постоје алгоритми којима би се испитивала горе наведена својства алгоритама.

Поступак показивања да је програм исправан назива се **верификовање програма**. Два основна приступа верификацији су:

динамичка верификација подразумева проверу исправности у фази извршавања програма, најчешће путем тестирања;

статичка верификација подразумева анализу изворног кода програма, често коришћењем формалних метода и математичког апарата.

Систематично тестирање је сигурно најзначајнији облик постизања високог степена исправности програма. Тестирањем на већем броју исправних улаза и упоређивањем добијених и очекиваних резултата може се открити велики број грешака. Нагласимо и да се тестирањем не може показати да је програм коректан, већ само да није коректан. Наиме, практично никада није могуће испитати понашање програма баш на свим исправним улазима. Већ програм који сабира два 32-битна броја има $2^{32} \cdot 2^{32} = 2^{64}$ исправних комбинација улазних параметара и исцрпно тестирање оваквог програма би трајало годинама. Зато се исцрпно тестирање скоро никада не спроводи, већ се програми тестирају тако да се улази бирају пажљиво, тако да покрију различите гране током извршавања програма. Обично се додатно посебно тестира понашање програма на неким граничним улазима (енгл. edge cases, corner cases), јер програми понекада не обрађују све специјалне случајеве како би требало. Многи системи за учење програмирања (такозвани грејдери, енгл. grader) оцењују ученичка решења тестирањем на већем броју тест-примера и савет почетницима је да током учења обавезно користе овакве системе.

О методама статичке верификације биће више речи у наставку овог поглавља.

1.2 Неке честе грешке у програмима

Сваки исправан програм мора да буде заснован на исправном алгоритму. Дакле, од неисправног алгоритма није могуће направити исправан програм и основна ствар приликом писања исправних програма је да се обезбеди исправност алгоритма који се примењује. Са друге стране, алгоритми се описују често на апстрактнијем нивоу него што су сами програми, и многи детаљи се занемарују. Зато се услед детаља имплементације од исправног алгоритма може добити неисправан програм. Наведимо неке честе грешке.

- Једна од најчешћих грешака представља **грешка прекорачења** (енгл. overflow). Наиме, ако се у имплементацији одабере бројевни тип података којим се не могу исправно представити сви подаци, тада програм даје неисправне резултате. Чест је случај да програмер одабере тип података који може да репрезентује исправно и улазне и излазне вредности, међутим, дешава се да међурезултати не могу да се репрезентују исправно, што се теже примети, а доводи до грешке.

- Честа грешка је **прекорачење граница низа** (енгл. buffer overflow). На пример, ако смо у низу одвојили место за 30 бројева, онда је могуће уписивати вредности само на позиције $0, 1, \ldots, 29$. Нарочито је критична позиција 30 (тј. у општем случају позиција n за низ од n елемената). Пошто у савременим програмским језицима бројање позиција у низовима креће од нуле, на позицију n није могуће уписивати вредности. У језику C++ се не врши провера опсега пре приступа елементима низа (тј. вектора) и одговорност је програмера да обезбеди да се не приступа ван граница - у супротном је понашање програма недефинисано, што значи да програм може да настави да ради неисправно и после одређеног броја инструкција да буде прекинут од стране оперативног система, али и да грешка може да прође неопажено. Уколико у петљи у низ уписујемо податке чији број не знамо унапред, неопходно је да пре сваког уписа проверимо да ли се упис врши унутар граница низа (или да користимо неки облик низа који допушта аутоматско проширивање додавањем нових елемената).
- Честа грешка је необраћање пажње на **специјалне случајеве**. На пример, ако у низу тражимо први елемент који задовољава неки услов, неопходно је да обезбедимо да програм коректно ради и у случају када ниједан елемент не задовољава тај услов. Треба пажљиво прецизирати да ли функција тада треба да врати број елемената низа или, на пример, -1, и треба осигурати да се у коду који позива ову функционалност добро реагује на ситуацију у којој тражени елемент не постоји. Специјални случајеви најчешће настају када неке вредности не постоје (када је неки скуп чије елементе разматрамо празан), затим када су улазне вредности у неком специјалном односу (на пример, да ли геометријски програм исправно ради ако су унете тачке колинеарне) и слично. При том, треба пажљиво прецизирати спецификацију задатка и одредити који специјални случајеви јесу, а који нису допуштени спецификацијом. Поново је потребно обратити пажњу на то да иако улазни параметри понекада не могу бити у неком специјалном односу, то не значи да међурезултати неће бити у том односу, па је онда потребно програме ипак прилагодити да обрате пажњу на све специјалне случајеве.

1.3 Индуктивно-рекурзивна конструкција

Кључна идеја у конструкцији алгоритама је то да је конструкција алгоритама веома тесно повезана са доказивањем теорема математичком индукцијом. **Математичка индукција**, у свом основном облику, је следећи начин доказивања особина природних бројева. Нека је P произвољно својство које се може формулисати за природне бројеве. Тада важи

$$(P(0) \wedge (\forall n)(P(n) \Rightarrow P(n+1))) \Rightarrow (\forall n)(P(n))$$

Дакле, да бисмо доказали да сваки природан број има неко својство P (тј. да бисмо доказали $(\forall n)(P(n))$), довољно је да докажемо да нула има то својство (тј. $P(0)$) и да докажемо да чим неки број има то својство, има га и његов следбеник (тј. да докажемо $(\forall n)(P(n) \Rightarrow P(n+1))$). Прво тврђење се назива **база индукције**, а друго **индуктивни корак**. Приницип математичке индукције је прилично јасан – на основу базе знамо да 0 има својство P, на основу корака да њен следбеник тј. 1 има својство P, на основу корака да његов следбеник тј. 2 има својство P итд. Интуитивно нам је јасно да на овај начин можемо стићи до било ког природног броја, који сигурно мора имати својство P. База се може формулисати и за веће вредности од нуле, али онда само можемо да тврдимо да елементи који су већи или једнаки од базе имају својство P.

Основни приступ конструкције алгоритама је тзв. **индуктивни** тј. **рекурзивни** приступ. Он у свом основном облику подразумева да се решење проблема веће димензије проналази тако што умемо да решимо проблем истог облика, али мање димензије и да од решења тог проблема добијемо решење проблема веће димензије (у напреднијим облицима је могуће и да се решава већи број проблема мање димензије). Притом за почетне димензије проблема решење морамо да израчунавамо директно, без даљег свођења на проблеме мање димензије. Ако се приликом свођења димензија проблема увек смањује, конструисани алгоритми ће се увек заустављати.

- Имплементација алгоритма може бити таква да промењиве унутар петље итеративно ажурирају своје вредности кренувши од вредности које представљају решења елементарних проблема, па до крајњих вредности које представљају решења задатог проблема. Пошто је ово прилично слично принципу математичке индукције, кажемо да је алгоритам дефинисан **индуктивно**.
- Имплементација може бити таква да функција која решава полазни проблем сама себе позива да би решила проблем истог облика, али мање димензије (осим у случају елементарних проблема, који се директно решавају) и тада кажемо да је алгоритам дефинисан **рекурзивно**.

Индуктивна конструкција лежи у основни практично свих итеративних алгоритама које смо до сада разматрали. На пример, алгоритам израчунавања збира серије бројева (на пример, збира елемената неког низа) почива на томе да знамо да израчунамо збир празне серије (то је 0) и да ако знамо збир серије од k елемената, тада умемо да израчунамо и збир серије која се добија проширивањем те серије додатним $k+1$-вим елементом (то радимо тако што дотадашњи збир увећамо за тај нови елемент).

```
int zbir = 0;
for (int i = 0; i < a.size(); i++)
    zbir = zbir + a[i];
```

Дакле, и у овом алгоритму имамо индуктивну базу (која одговара иницијализацији промењиве пре уласка у петљу) и индуктивни корак (који одговара телу петље, у ком се ажурира вредност резултујуће промењиве, у овом случају збира). База може одговарати и случају једночланог (а не обавезно празног) низа, али тада не можемо да гарантујемо да ће алгоритам радити исправно у случају празног низа. То одговара варијанти алгоритма у којој збир иницијализујемо на први елемент низа, па га увећавамо редом за један по један елемент од позиције 1 надаље.

Рекурзивна имплементација израчунавања збира елемената низа може бити следећа (у њој се приликом решавања проблема димензије $n > 0$ експлицитно захтева решавање проблема димензије $n - 1$).

```
int zbir(int a[], int n) {
    if (n == 0)
       return 0;
    else
       return zbir(a, n-1) + a[n-1];
}
```

Дефинисање алгоритама индуктивно-рекурзивом конструкцијом је у веома тесној вези са доказивањем њихове коректности. Иако постоје формални оквири за доказивање коректности императивних програма (пре свега *Хорова логика*), ми ћемо се бавити искључиво неформалним доказима и веза између логике у којој вршимо доказивање и (императивног) програмског језика у којем се програм изражава биће прилично неформална.

Рецимо и да ћемо приликом доказивања коректности програма обично игнорисати ограничења записа бројева у рачунару и подразумеваћемо да је опсег бројева неограничен и да се реални бројеви записују са максималном прецизношћу. Дакле, нећемо обраћати пажњу на грешке које могу настати услед прекорачења или поткорачења вредности током извођења аритметичких операција (иако реално то често може бити узрок грешака у програмима).

1.3.1 Доказ коректности рекурзивних функција

Проблем: Дефинисати функцију која одређује минимум непразног низа бројева и доказати њену коректност.

Алгоритам се веома једноставно конструише индуктивно-рекурзивном конструкцијом. Димензија проблема у овом примеру је број елемената низа.

База: Ако низ има само један елемент, тада је тај елемент уједно и минимум.

Корак: У супротном, претпоставимо да некако умемо да решимо проблем за мању димензију и на основу тога покушајмо да добијемо решење за цео низ. Дакле, претпоставимо да је дужина низа $n > 1$ и да умемо да нађемо број m који представља минимум првих $n - 1$ елемената низа. Минимум целог низа дужине n је мањи од бројева m и преосталог, n-тог елемента низа (ако бројање креће од 0, то је елемент a_{n-1}).

На основу овога можемо дефинисати рекурзивну функцију.

```
#include <iostream>
using namespace std;

int min2(int a, int b) {
    return a < b ? a : b;
}
```

```
int minNiza(int a[], int n) {
    if (n == 1)
       return a[0];
    else {
       int m = minNiza(a, n-1);
       return min2(m, a[n-1]);
    }
}

int main() {
    int a[] = {3, 5, 4, 1, 6, 2, 7};
    int n = sizeof(a) / sizeof(int);
    cout << minNiza(a, n) << endl;
}
```

Рецимо и да смо уместо дефинисања функције `min2` за одређивање минимума два броја могли користити и функцију `std::min` из заглавља `<algorithm>`. Ипак, у овим програмима нећемо користити специфичне могућности језика C++, да бисмо нагласили да технике које у овом поглављу уводимо нису ни по чему специфичне за тај језик.

Коректност претходног алгоритма се може формулисати у облику следеће теореме.

Теорема: За сваки непразан низ a (низ за који је дужина $|a| \geq 1$) и за свако $1 \leq n \leq |a|$ позив `minNiza(a, n)` враћа најмањи међу првих n елемента низа a (са a и n су обележене вредности низа `a` и променљиве `n`, а са $|a|$ дужина низа `a`).

Ту теорему можемо доказати индукцијом.

- Базу индукције представља случај $n = 1$, тј. позив `minNiza(a, 1)`. На основу дефиниције функције `minNiza` резултат је `a[0]` тј. први члан низа a_0 и тада тврђење тривијално важи (јер је он уједно најмањи међу првих 1 елемената низа).
- Као индуктивну хипотезу можемо претпоставити да ако важи $1 \leq n - 1 < |a|$, тада позив `minNiza(a, n-1)` враћа најмањи од првих $n-1$ елемената низа a. Из те претпоставке потребно је да докажемо да за n које задовољава $1 < n \leq |a|$ позив `minNiza(a, n)` враћа најмањи од првих n елемената низа a (при том је a непразан низ). На основу дефиниције функције `minNiza`, позив `minNiza(a, n)` ће вратити минимум бројева m (који представља резултат позива `minNiza(a, n-1)`) и a_{n-1}. Пошто су услови индуктивне хипотезе задовољени, на основу индуктивне хипотезе знамо да ће m бити најмањи међу првих $n-1$ елемената низа a. Зато ће минимум броја m и n-тог елемента низа (елемента a_{n-1}) бити најмањи међу првих n елемената низа a.

Примећујемо огромну сличност између рекурзивне конструкције алгоритма и индуктивног доказа његове коректности. Стога слободно можемо да кажемо да су рекурзија и индукција "две стране исте медаље" (индукцију користимо као технику доказивања, а рекурзију као технику дефинисања функција тј. конструкције алгоритама).

Рецимо и да је овај облик коришћења математичке индукције мало нестандардан, јер се не користи директно индукција по природним бројевима, већ се користи индукција по структури рекурзивне функције у којој се, из претпоставке да сваки рекурзивни позив враћа коректан резултат, доказује да функција враћа коректан резултат. Таква теорема индукције се може доказати на основу класичне математичке индукције по броју рекурзивних позива, под претпоставком да се докаже да се рекурзивна функција увек зауставља.

1.3.2 Доказ коректности итеративних алгоритама - инваријанте петље

Један од основних појмова у анализи и разумевању итеративних програма су **инваријанте петљи**. То су логички услови који важе непосредно пре петље, затим након сваког извршавања наредби у телу петље и непосредно након извршавања целе петље. Корисне инваријанте су оне које гарантују коректност алгоритма који та петља имплементира. Инваријанте суштински описују значење свих променљивих унутар петље. Илуструјмо појам инваријанте на једном једноставном примеру.

Размотримо следећу, класичну имплементацију алгоритма за одређивање минимума непразног низа бројева.

```
#include <iostream>
#include <algorithm>
```

```
using namespace std;

int min2(int a, int b) {
    return a < b ? a : b;
}

int minNiza(const vector<int>& a) {
    int m = a[0];
    for (int i = 1; i < a.size(); i++)
        m = min2(m, a[i]);
    return m;
}

int main() {
    vector<int> a{3, 5, 4, 1, 6, 2, 7};
    cout << minNiza(a) << endl;
}
```

У сваком кораку петље, део низа чији минимум знамо постаје дужи за по један елемент. Алгоритам креће од префикса низа дужине 1 и поставља променљиву `m` на вредност првог елемента низа a_0. У сваком кораку петље, претпостављамо да променљива m садржи вредност минимума првих i елемената низа, а онда у телу петље обрађени део низа проширујемо додајући $i+1$-ви елемент низа, на позицији i. Минимум проширеног низа се израчунава као минимум минимума првих i елемената низа (чија је вредност смештена у променљивој `m`) и додатног елемента низа a_i. Након извршавања тела петље, део низа чији минимум је познат је проширен на $i+1$ елемент. На крају петље је i једнако дужини низа, па променљива `m` садржи минимум целог низа.

Пре него што пређемо на формални доказ претходог разматрања, скренимо пажњу на то да именоване величине у математици (тачније алгебри) и у програмирању имају различите особине. Наиме, именоване величине у математици (параметри, непознате) означавају једну вредност док у (императивном) програмирању именоване величине имају динамички карактер и мењају своје вредности током извршавања програма по правилима задатим самим програмом. На пример, бројачка променљива `i` у некој петљи може редом имати вредности 1, 2 и 3. Да бисмо направили разлику између променљивих и њихових текућих вредности, користићемо различит фонт - променљиву програма ћемо обележавати са `i`, а њену вредност са i. Ако желимо да разликујемо стару и нову вредност променљиве `i`, користићемо ознаке i и i'. Ако желимо да нагласимо да је променљива редом узимала неку серију вредности, користићемо ознаке i_0 (почетна вредност променљиве `i`), i_1, i_2, … У ситуацијама у којима се вредност променљиве не мења (на пример, ако је дужина низа током целог трајања програма иста), нећемо обраћати пажњу на разлику између променљиве програма (нпр. `n`) и њене вредности (нпр. n). Елементе низова ћемо такође обележавати индексима и обично ћемо претпостаљати да бројање креће од нуле (нпр. a_0, a_1, …).

Формално, можемо доказати следећу теорему.

Теорема: Ако је низ a дужине $n \geq 1$, непосредно пре почетка петље, у сваком кораку петље (и на њеном почетку, непосредно након провере услова, али и на њеном крају, непосредно након извршавања тела), као и након извршавања целе петље важи да је $1 \leq i \leq n$ и да је m минимум првих i елемената низа (где је i текућа вредност променљиве `i`, а m текућа вредност променљиве `m`).

Ово тврђење можемо доказати индукцијом и то по броју извршавања тела петље (обележимо тај број са k). Напоменимо само да ћемо петљу `for` сматрати само скраћеницом за петљу `while`, тако да ћемо иницијализацију петље сматрати за кôд који се извршава пре петље, док ћемо корак петље сматрати као последњу наредбу тела петље.

```
    int n = a.size();
    int m = a[0];
    int i = 1;
    while (i < n) {
        m = min2(m, a[i]);
        i++;
    }
```

Такође, имплицитно ћемо подразумевати да се током извршавања петље низ ни у једном тренутку не мења (и то се експлицитно може доказати индукцијом). Ни променљива `n` не мења своју вредност.

Да бисмо у доказу били прецизнији, обележимо са $m_0, m_1, \ldots, m_k, \ldots$ вредности промењиве `m`, а са $i_0, i_1, \ldots, i_k, \ldots$ вредност промењиве `i` након $0, 1, \ldots, k, \ldots$ извршавања тела петље. Пошто промењива `n` не мења своју вредност, употребљаваћемо само ознаку n.

- Базу индукције чини случај $k = 0$ тј. случај када се тело петље није још извршило. Пре уласка у петљу промењива `i` се иницијализује на 1 (важи $i_0 = 1$). Пошто претпостављамо да је низ непразан, важи да је $1 \leq i = i_0 = 1 \leq n$. Промењива `m` се иницијализује на вредност `a[0]` (важи $m_0 = a_0$), што је заиста минимум једночланог префикса низа a. Дакле, услови су задовољени пре првог извршавања тела петље.

- Претпоставимо сада као индуктивну хипотезу да тврђење важи након k извршавања тела петље. Дакле, претпостављамо да услови теореме важе за вредности m_k и i_k тј. да је $1 \leq i_k \leq n$ и да је m_k једнако минимуму првих i_k елемената низа (са i_k и m_k обележавамо вредности промењивих након k извршавања тела петље). Ако је услов петље испуњен, то ће уједно бити и вредности промењивих на почетку тела петље, пре њеног $k+1$-вог извршавања. Након k извршавања тела петље важи да је $i_k = k+1$, јер је промењива `i` имала почетну вредност 1 и тачно k пута је увећана за 1 (и ово би се формално могло доказати индукцијом).

  Из индуктивне хипотезе и претпоставке да је услов петље `i < n` испуњен (тј. да је $i_k < n$) докажимо да након $k + 1$ извршавања тела петље услови теореме важе и за вредности m_{k+1} и i_{k+1} (са m_{k+1} и i_{k+1} обележавамо вредности промењивих након $k + 1$ извршавања тела петље). Вредности m_{k+1} и i_{k+1} се могу лако одредити на основу вредности m_k и i_k, анализом једног извршавања тела петље. Важи да је $i_{k+1} = i_k + 1 = k + 2$. Зато, пошто је $1 \leq i_k = k + 1 < n$, важи и да је $1 \leq i_{k+1} = k + 2 \leq n$, па је услов који се односи на распон вредности промењиве `i` очуван. Докажимо и да је m_{k+1} минимум првих i_{k+1} елемента низа. Важи да је m_{k+1} минимум вредности m_k и елемента a_{i_k}, тј. a_{k+1}. На основу индуктивне хипотезе знамо да је m_k минимум првих $i_k = k + 1$ елемената низа. Зато ће m_{k+1} бити минимум првих $k + 2$ елемената низа (закључно са елементом a_{k+1}), што је тачно i_{k+1} елемената низа, па и други услов остаје очуван.

Означимо са i и m вредности промењивих `i` и `m` након извршавања петље. На основу доказаног тврђења знамо да услови наведени у њему важе и након завршетка петље. Када се петља заврши, важи да је $i = n$ (јер на основу првог услова знамо да је $1 \leq i \leq n$, а услов петље `i < n` није испуњен). На основу другог услова знамо да је m минимум n чланова низа (што је заправо цео низ, јер је n његова дужина), тј. да промењива `m` садржи тражену вредност, чиме је доказана парцијална коректност. Заустављање се доказује једноставно тако што се докаже да се у сваком кораку петље ненегативна вредност $n - i$ смањује за по 1, док не постане 0.

Ако размотримо структуру претходног разматрања, можемо установити да смо идентификовали логичке услове који су испуњени непосредно пре и непосредно након сваког извршавања тела петље. Такви услови се називају **инваријанте петље**. Да бисмо доказали да је неки услов инваријанта петље, довољно је да докажемо:

(1) да тај услов важи пре првог уласка у петљу и

(2) да из претпоставке да тај услов важи пре неког извршавања тела петље и да је услов петље испуњен докажемо да тај услов важи и након извршавања тела петље.

Те две чињенице нам, на основу индуктивног аргумента, гарантују да ће услов бити испуњен пре и после сваке итерације петље, као и након извршавања целе петље (ако се она икада заустави), тј. да ће тај услов бити инваријанта петље (тај доказ се може спровести класичном математичком индукцијом на основу броја извршавања тела петље). Приметимо да први корак одговара доказивању базе индукције, а други доказивању индуктивног корака.

Свака петља има пуно инваријанти, међутим, од интереса су нам само оне инваријанте које у комбинацији са условом прекида петље (под претпоставком да петља није прекинута наредбом `break`) имплицирају услов који нам је потребан након петље. Ако је петља једина у неком алгоритму, обично је то онда услов коректности самог алгоритма. Дакле, након доказа леме која чини основу доказа да је неки услов инваријанта петље, потребно је да докажемо и

(3) да из тога да инваријанта важи након завршетка петље и да услов петље није испуњен следи коректност алгоритма.

Дакле, општа структура анализе програма коришћењем инваријанти се може описати на следећи начин.

```
<incijalizacija>
// ovde vazi <invarijanta>
```

```
while (<uslov>)
   // ovde vaze i <uslov> i <invarijanta>
   <telo>
   // ovde vazi <invarijanta>
// ovde ne vazi <uslov>, a vazi <invarijanta>
```

Изолујмо кључне делове претходног доказа и прикажимо их у формату који ћемо и убудуће користити приликом доказивања инваријанти петљи (индукција ће у тим доказима бити само имплицитна).

Лема: Ако је низ a дужине $n \geq 1$, услов да је $1 \leq i \leq n$ и да је m минимум првих i елемената низа је инваријанта петље (где са i обележавамо текућу вредност промењиве `i`, а са m текућу вредност промењиве `m`).

- Пре уласка у петљу промењива `i` се иницијализује на 1 (важи $i = 1$). Пошто претпостављамо да је низ непразан, важи да је $1 \leq i \leq n$. Промењива `m` се иницијализује на вредност `a[0]` (важи $m = a_0$), што је заиста минимум једночланог префикса низа a.
- Претпоставимо да тврђење важи након уласка у петљу тј. да је вредност промењиве `m` (означимо је са m) једнака минимуму првих i чланова низа (где је i вредност промењиве `i` на уласку у петљу), да је $1 \leq i \leq n$, као и да је услов петље испуњен тј. да је $i < n$.

  Пошто је након извршавања тела петље вредност промењиве `i` увећана за један, важи да је $i' = i + 1$ (где са i' обележавамо вредност промењиве i након извршавања тела и корака петље). Пошто је важи да је $i < n$ и $1 \leq i \leq n$, након извршавања тела петље, важиће да је $1 \leq i' \leq n$.

  Нова вредност промењиве `m` (означимо је са m') биће једнака мањој од вредности m и a_i. На основу претпоставке важи да је m једнако минимуму првих i елемената низа, тј. минимуму бројева a_0, …, a_{i-1}, па је m' једнако минимуму бројева a_0, …, a_i, што је управо минимум првих $i+1$ елемената низа, па је заиста m' минимум првих i' елемената низа.

Теорема: Након извршавања петље, промењива `m` садржи минимум целог низа.

На основу инваријанте важи да је $1 \leq i \leq n$, а пошто по завршетку петље њен услов није испуњен, важи да је $i = n$. На основу инваријанте важи и да промењива `m` садржи минимум првих i елемената низа, а пошто је $i = n$, где је n број чланова низа, то је заправо минимум целог низа.

У наставку овог поглавља видећемо још неколико примера примене технике инваријанте петље. Мора се признати да када се техника користи потпуно формално, да би се доказала коректност већ написаног програмског кода, то не делује нарочито инспиришуће (поготово, ако су програми једноставни и ако је једноставно интуитивно разумети разлоге њихове коректности). Ретко када се у практичном програмирању коректност заиста доказује потпуно формално (осим у случају софтвера који може да угрози велики број живота, попут, на пример, софтвера који управља метро-системом у Паризу, који јесте у потпуности формално верификован). Међутим, аргументе и инваријанте на којима коректност почива програмер често "проврти по глави". Видећемо и да се техника инваријанти може употребити и пре него што је програм написан у циљу извођења програмског кода из спецификације. Јасне инваријанте често једнозначно указују на то како програмски кôд треба да изгледа и на тај начин помажу у процесу програмирања.

Задаци који су одабрани нису ни по чему посебни – они ће бити поновљени у поглављима у којима се уводе опште програмерске технике које се у њима примењују.

Задатак: Тробојка

Написати програм који учитава низ целих бројева а затим га трансформише тако да елементи буду подељени у три дела у зависности од задатих вредности A и B. У првом делу су елементи мањи од задате вредности A (вредности из интервала $[-\infty, A)$), у другом елементи већи или једнаки задатој вредности A и мањи или једнаки задатој вредности B (вредности из интервала $[A, B]$), а у трећем елементи већи од задате вредности B (вредности из интервала $(B, +\infty)$). Није битно у ком се редоследу налазе елементи унутар делова. Учитати елементе у низ, а затим реорганизовати редослед елемената у том низу (не користити помоћне низове).

Улаз: У једној линији стандардног улаза налази се број елемената низа, N, а затим се, у наредној линији налазе елементи низа раздвојени размацима. У последње две линије се налазе цели бројеви A и B одвојени празнином, и при томе је $A < B$.

Излаз: Исписати елементе резултујућег низа на стандардни излаз (могуће је исписати елементе сваке од три групе у посебном реду, раздвојене размацима, а могуће је исписати и цео низ у једном реду или у више редова).

Пример

Улаз	*Излаз*
`10`	`1 2`
`1 3 5 4 8 5 7 2 3 6`	`5 3 5 3`
`3`	`7 6 8`
`5`	

Решење

Један пролаз кроз низ

Задатак можемо решити помоћу само једног пролаза кроз низ и то "у месту" тј. без коришћења помоћног низа. Алгоритам у наставку познат је под називом "Холандска застава тробојка" (енгл. Dutch national flag) и приписује се чувеном информатичару Дајкстри (енгл. Edsger W. Dijkstra).

Одржаваћемо три променљиве l, d и i и током петље наметнућемо да важи $0 \leq l \leq i \leq d \leq n$ и да важе следећи услови.

- У интервалу позиција $[0, l)$ налазиће се елементи мањи од A тј. бројеви из интервала $(-\infty, A)$,
- у интервалу позиција $[l, i)$ налазиће се елементи из интервала $[A, B]$,
- у интервалу позиција $[i, d)$ налазиће се елементи који још нису испитани,
- у интервалу позиција $[d, n)$ налазиће се елементи који су већи од B тј. елементи из интервала $(B, +\infty)$.

Дакле, одржавамо распоред `<<<<===???>>>`, где су са `<` обележени елементи прве групе, са `=` елементи друге, са `?` елементи треће групе, а са `>` елементи четврте групе.

Да би инваријанта важила пре уласка у петљу, јасно је да мора да важи да је $i = 0$ и $d = n$ (јер су сви елементи из интервала $[i, d) = [0, n)$ неиспитани). Такође, да бисмо били сигурни да су и интервалу $[0, l)$ сви елементи мањи од A, тај интервал мора бити празан и мора да важи да је $l = 0$. Након овакве иницијализације и интервал $[l, i) = [0, 0)$ и интервал $[d, n) = [n, n)$ је празан, па задовољава наметнути услов.

Петља ће се извршавати док год има неиспитаних елемената, а то је док је $i < d$. Размотримо како треба да изгледа тело петље, да би услови били одржани.

- Ако је елемент на позицији i мањи од броја A тада ћемо га заменити са елементом на позицији l (првим елементом из интервала $[A, B]$), након чега можемо увећати и i и l.
- У супротном, ако је елемент на позицији i мањи или једнак од B он припада интервалу $[A, B]$ и већ је на свом допуштеном месту, па само можемо увећати вредност i.
- У супротном елемент је већи од B и тада можемо смањити вредност d и разменити елемент на позицији i са елементом на (умањеној) позицији d, не мењајући вредност i (да би се елемент који је управо доведен на позицију i могао испитати у наредној итерацији).

На крају петље важи да је $i = d$. Уз остале наметнуте услове тврђење одатле следи (елементи из интервала позиција $[0, l)$ су мањи од A, елементи из интервала позиција $[l, i) = [l, d)$ су између A и B, интервал непрегледаних елемената $[i, d)$ је празан, док су елементи из интервала $[d, n)$ већи од B. Дакле, низ је разбијен на надовезане сегменте $[0, l)$, $[l, d)$ и $[d, n)$ и у сваком сегменту се налазе одговарајући елементи.

Пример. Размотримо рад алгоритма на једном примеру. Нека је $A = 4$, $B = 7$ и нека низ има садржај `5 1 8 6 3 9 4 2`. У наставку ћемо приказати стање низа током извођења алгоритма.

```
l             d
5 1 8 6 3 9 4 2
i

l             d
5 1 8 6 3 9 4 2
  i

  l           d
```

```
1 5 8 6 3 9 4 2
    i

  l           d
1 5 2 6 3 9 4 8
    i

    l         d
1 2 5 6 3 9 4 8
      i

    l         d
1 2 5 6 3 9 4 8
        i

      l       d
1 2 3 6 5 9 4 8
          i

      l     d
1 2 3 6 5 4 9 8
          i

      l     d
1 2 3 6 5 4 9 8
            i
```

Анализа сложености. У сваком кораку петље се или увећава `i` или смањује `d`, док се не сусретну, што се дешава у n корака. Сложеност овог приступа је, дакле, $O(n)$.

```
// funkcija organizuje elemente vektora tako da se prvo nalaze elementi
// za koje vazi da su iz intervala (-Inf, A), nakon toga dolaze
// elementi iz intervala [A, B], i nakon toga elementi iz intervala
// (B, Inf)
void podelaNiza(vector<int>& niz, int A, int B) {
  // - u intervalu pozicija [0, l) su elementi iz intervala (-Inf, A)
  // - u intervalu pozicija [l, i) su elementi iz intervala [A, B]
  // - u intervalu pozicija [i, d) su jos neispitani elementi
  // - u intervalu pozicija [d, n) su elementi iz intervala (B, Inf)
  int l = 0, i = 0, d = niz.size();
  // dok god postoje neispitani elementi
  while (i < d) {
    if (niz[i] < A)
      // menjamo tekuci element sa prvim elementom iz intervala [A, B]
      swap (niz[i++], niz[l++]);
    else if (niz[i] <= B)
      // tekuci element ostaje na svom mestu
      i++;
    else
      // menjamo tekuci element sa poslednjim neispitanim
      swap(niz[i], niz[--d]);
  }
}
```

Задатак: Први који није дељив

Размотримо низ бројева $210, 2310, 390, 30, 510, 66, 6, 138, 46, 106, 59, 17, 23$. Он је интересантан из неколико разлога. На пример, првих пет бројева је дељиво са 10, а после ниједан број није дељив са 10. Првих десет бројева је парно, а после су сви бројеви непарни. Првих осам бројева је дељиво са 6, а после ниједан број

није дељив са 6. Прва два броја су дељива са 210, а после ниједан број није дељив са 210, итд. Покушај да пронађеш још оваквих правилности. Напиши програм који за сваки унети делилац одређује колико бројева је дељиво њиме. Сматрати да за сваки унети делилац важи наведена правилност.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^5$), а затим у наредном реду n природних бројева (мањих од 10^{18}) раздвојених по једним размаком. Након тога се до краја улаза уносе делиоци (сваки у посебном реду). За сваки делилац се сигурно зна (и то није потребно проверавати) да се у низу налазе прво бројеви који јесу, а затим бројеви који нису дељиви тим делиоцем.

Излаз: За сваки унети делилац у посебном реду исписати број елемената низа који су њиме дељиви.

Пример

Улаз	*Излаз*
13	5
210 2310 390 30 510 66 6 138 46 106 59 17 23	10
10	8
2	10
6	0
2	5
4	
15	

Решење

Бинарна претрага

Захваљујући интересантној особини низа, задатак ефикасно може бити решен применом алгоритма бинарне претраге. У питању је варијанта алгоритма бинарне претраге у ком се уместо позиције конкретне вредности у сортираном низу захтева проналажење прве позиције на којој се налази елемент који задовољава неки услов. Наиме, под претпоставком да се у низу прво налазе елементи који не задовољавају тај услов, а затим елементи који задовољавају тај услов, *преломну тачку* (тренутак када се из једне прелази у другу групу елемената) можемо наћи бинарном претрагом. Дакле, ако је низ облика `-------+++++++`, бинарном претрагом можемо пронаћи позицију последњег минуса, првог плуса, број минуса или број плусева, где смо са `-` означили оне елементе који не задовољавају, а са `+` оне елементе који задовољавају дати услов.

Ручно имплементирана бинарна претрага

Током рада алгоритма, одржавамо две променљиве l и d такве да важи инваријанта да је $0 \leq l \leq d + 1 \leq n$ и да су

- лево од l тј. у интервалу позиција $[0, l)$ елементи који не задовољавају услов,
- десно од d тј. у интервалу позиција (d, n) елементи који задовољавају услов.

У интервалу позиција $[l, d]$ налазе се елементи чији статус још није познат. На почетку су сви елементи непознати, па је јасно да интервал $[l, d]$ треба иницијализовати на $[0, n - 1]$, тј. променљиву l треба иницијализовати на нулу, а d на вредност $n - 1$. Интервали $[0, l)$ и (d, n) су празни, па је инваријанта очувана (услов $l \leq d + 1$ се своди на $0 \leq n$, што је тривијално испуњено).

Ако интервал позиција $[l, d]$ није празан тј. ако је $l \leq d$, проналазимо му средину $s = l + \left\lfloor \frac{d-l}{2} \right\rfloor$.

- Ако елемент на позицији s задовољава услов, тада на основу монотоности услов задовољавају и сви елементи десно од s. Зато померамо d за једно место лево од средине тј. вредност променљиве d постављамо на $s - 1$.
- Ако елемент на позицији s не задовољава услов, тада на основу монотоности услов не задовољавају ни сви елементи лево од s. Зато померамо l за једно место десно од средине тј. вредност променљиве l постављамо на $s + 1$.

Претрага траје све док се интервал $[l, d]$ не испразни, тј. док је $l \leq d$. Тада је $l = d + 1$ и елементи који не задовољавају услов се налазе на позицијама $[0, l) = [0, d]$, док се елементи који не задовољавају услов налазе на позицијама $(d, n) = [d + 1, n) = [l, n)$. Дакле, први елемент који задовољава услов је на позицији l, а последњи који не задовољва услов на позицији d.

Пример. Прикажимо рад алгоритма на једном примеру.

```
l               d
1 7 3 5 9 11 2 8 6
        s

          l     d
1 7 3 5 9 11 2 8 6
            s

          ld
1 7 3 5 9 11 2 8 6
          s

          d  l
1 7 3 5 9 11 2 8 6
```

Имплементација се може направити на следећи начин.

```
int prviKojiNijeDeljiv(const vector<long long>& a, long long k) {
  int n = a.size();
  int l = 0, d = n-1;
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (a[s] % k != 0)
      d = s - 1;
    else
      l = s + 1;
  }
  return l;
}
```

Доказ коректности. Докажимо формално коректност овог алгоритма. Уз поменуте услове инваријанта је и да важи $0 \leq l \leq d+1 \leq n$.

Након иницијализације $l = 0, d = n-1$, услови су испуњени (интервали $[0, l)$ и (d, n) су празни, док је услов $0 \leq l \leq d+1 \leq n$ еквивалентан услову $0 \leq 0 \leq n \leq n$ и тривијално је испуњен.

Ако интервал позиција $[l, d]$ није празан тј. ако је $l \leq d$, проналазимо му средину $s = l + \lfloor \frac{d-l}{2} \rfloor$. Пошто је $l \leq d$, важи и да је $l \leq s \leq d$.

- Ако елемент на позицији s задовољава услов, тада на основу монотоности услов задовољавају и сви елементи десно од s. Зато вредност промењиве d постављамо на $s-1$ (нове вредности промењивих су $l' = l$ и $d' = s-1$). Тиме инваријанта остаје на снази (посебно, сви елементи у интервалу позиција $(s-1, n) = [s, n)$ задовољавају услов). Важи и услов $0 \leq l' \leq d'+1 \leq n$, јер је он еквивалентан услову $0 \leq l \leq s \leq n$.
- Ако елемент на позицији s не задовољава услов, тада на основу монотоности услов не задовољавају ни сви елементи лево од s. Зато вредност промењиве l постављамо на $s+1$ (нове вредности промењивих су $l' = s+1$ и $d' = d$). Тиме инваријанта остаје одржана (посебно, ниједан елемент у интервалу позиција $(0, l) = [0, s]$ не задовољава услов). Важи и услов $0 \leq l' \leq d'+1 \leq n$ који је еквивалентан услову $0 \leq s+1 \leq d+1 \leq n$.

Када се интервал испразни, тада је $l > d$, па пошто важи $0 \leq l \leq d+1 \leq n$, важи и $l = d+1$. На основу инваријанте знамо да су елементи који задовољавају услов на позицијама $(d, n) = [l, n]$. Зато је први елемент који задовољава услов је на позицији l (што је уједно и број елемената који не задовољавају услов). Елементи који не задовољавају услов су на позицијама $[0, l) = [0, d+1) = [0, d]$, па је последњи елемент који не задовољава на позицији d.

Заустављање се лако доказује тако што се доказује да се у сваком кораку петље интервал $[l, d]$ тј. његова дужина $d-l+1$ смањује, што је прилично очигледно и када је $l' = l$ и $d' = s-1 < d$ и када је $l < l' = s+1$ и $d' = d$.

Анализа сложености. Пошто се у сваком кораку претраге ширина интервала $[l, d]$ преполови, пошто се иницијално креће од интервала $[0, n-1]$ који има n елемената и пошто се алгоритам завршава када се интервал

испразни, сложеност алгоритма је $O(\log n)$. Наиме, дужина интервала после k корака је $\left\lfloor \frac{n}{2^k} \right\rfloor$ и важи да је $\left\lfloor \frac{n}{2^k} \right\rfloor < 1$ када је $k > \log_2 n$.

Исправљање грешака на основу формалне анализе кода

Када је кôд коректан, доказ је обично неинформативан. Помаже нам да "мирно спавамо", али ништа више од тога. Много интересантнија ситуација се дешава у случају када нам формално резоновање о коду помаже да детектујемо и исправимо грешке у програму (тзв. багове). Погледајмо наредни покушај имплементације алгоритма.

```
int l = 0, d = n;
while (l < d) {
  int s = l + (d - l) / 2;
  if (a[s] % k != 0)
    d = s-1;
  else
    l = s+1;
}
cout << d+1 << '\n';
```

На основу инцијализације делује да покушавамо да претражимо полузатворени интервал $[l, d)$. Пошто је у питању бинарна претрага, изгледа да се намеће инваријанта да је $0 \leq l \leq d \leq n$ и да су:

- сви елементи из $[0, l)$ дељиви са k,
- ниједан елемент из интервала $[d, n)$ није дељив са k.

На почетку су оба та интервала празна, па инваријанта за сада добро функционише. Ако погледамо услов петље, делује да петља ради док се интервал непознатих елемената $[l, d)$ не испразни (заиста, када је $l \geq d$, тај интервал је празан). За сада све ради како треба. Покушамо сада да проверимо да ли извршавање тела петље одржава инваријанту.

- Ако a_s није дељив са k, тада се променљива `d` поставља на вредност $d' = s - 1$. На основу инваријанте треба да важи да ниједан елемент у интервалу $[d', n)$ није дељив са k. Међутим, ми то не знамо, јер само знамо да је a_s није дељив са k, али не знамо да a_{s-1} није дељив са k. Дакле, овде се сигурно крије грешка у коду. Ако доделу `d = s-1` заменимо са `d = s`, тада ће инваријанта бити одржана (јер знамо да a_s није дељив са k, па са k неће бити дељив ниједан елемент иза њега).
- Ако a_s јесте дељив са k, тада се променљива `l` поставља на вредност $l' = s + 1$. На основу инваријанте треба да важи да су сви елементи у интервалу $[0, l')$ дељиви са k, међутим, то ће овде бити испуњено, јер је a_s дељив са k, па су са k дељиви и сви елементи испред њега. Дакле, у овом случају је кôд коректан и инваријанта остаје одржана.

На крају, када се петља заврши можемо закључити да важи да је $l = d$ (јер све време важи да је $l \leq d$, а након петље не важи да је $l < d$). У коду се за позицију првог елемента који није дељив са k проглашава позиција $d + 1$. Иако је у оригиналној варијанти кода `l` могло без проблема да се замени са `d+1`, у овој варијанти то није могуће. Наиме, ми на основу инваријанте овог кода знамо да се на позицији $l = d$ налази елемент који није дељив са k, а да се на позицији $l - 1$ налази елемент који јесте дељив са k (осим када је $l = 0$ и тада нема елемената дељивих са k). Зато крајњи резултат није коректан и потребно га је заменити са `d`, јер се први елемент који није дељив са k налази на позицији d (осим када су сви елементи дељиви са k, када је $d = n$, но и тада је d исправна повратна вредност). Дакле, формалном анализом смо открили и исправили две грешке.

Програмери често програм исправљају тако што насумице покушавају да помере индексе за 1 лево или десно, да замене мање са мање или једнако и слично. Већ на овако кратким програмима се види да је простор могућих комбинација велики, а да је могућност за грешку приликом таквог експерименталног приступа веома велика. Стога је увек боље застати, формално анализирати шта је потребно да кôд ради и исправити га на основу резултата формалне анализе.

На крају, скренимо пажњу на још један детаљ исправљеног програма. Парцијална коректност је јасна на основу анализе коју смо спровели, међутим, заустављање може бити доведено у питање, с обзиром на наредбу `d = s`. Заустављање доказујемо тако што показујемо да се у сваком кораку смањује број непознатих елемената, тј. да дужина интервала $[l, d)$ која је једнака $d - l$ у сваком кораку петље опада. Пошто је $l \leq d$ инваријанта, смањивање не може трајати довека, па се у неком тренутку програм зауставља. Поставља се питање да ли се $d - l$ смањује и у измењеном коду у коме се јавља наредба `d=s`. Одговор је потврдан, а образложење је

суптилно. Прво, на основу услова петље важи да је $l < d$. Даље, вредност s се израчунава наредбом `s = l + (d - l) / 2` што нам да је $s = \left\lfloor \frac{l+d}{2} \right\rfloor$. Због заокруживања наниже, важи да је $s < d$ и зато се након одређивања $d' = s$, $l' = l$ вредност $d' - l'$ смањује у односу на $d - l$. Важи и да је $l \leq s$, али пошто је у другој грани $l' = s + 1$ и $d' = d$, вредност $d' - l'$ се опет смањује у односу на $d - l$. Да је заокруживање којим случајем вршено навише (нпр. `s = l + (d - l + 1) / 2`), програм би могао упасти у бесконачну петљу.

```
int prviKojiNijeDeljiv(const vector<long long>& a, long long k) {
  int n = a.size();
  int l = 0, d = n;
  while (l < d) {
    int s = l + (d - l) / 2;
    if (a[s] % k != 0)
      d = s;
    else
      l = s + 1;
  }
  return d;
}
```

Види другачија решења овог задатка.

Задатак: Најмањи број који није збир елемената скупа

Дат је скуп природних бројева (задат у облику сортираног низа). Одредити најмањи природан број који није збир неких елемената тог скупа (сваки елемент скупа може само једном учествовати у збиру).

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^3$), а затим у наредном реду сортиран низ од n различитих природних бројева мањих од 10^4.

Излаз: На стандардни излаз исписати тражени најмањи природан број који није збир неких елемената тог скупа.

Пример

Улаз	*Излаз*
`8` `1 2 4 7 15 32 35 48`	`30`

Решење

Чињеница да су елементи сортирани олакшава решење задатка. Обрађиваћемо елемент по елемент и одржаваћемо границу до које смо сигурни да се сваки број може представити као збир неког подскупа. Можда мало изненађујуће, та граница је у сваком кораку једнака збиру свих тренутно учитаних елемената. Ако је нови учитани елемент строго већи од збира свих претходних елемената увећаног за један, онда се тај увећани збир не може добити као подскуп. У супротном можемо бити сигурни да се сви бројеви од 0 па до збира свих елемената (у који је укључен и нови елемент) могу добити као збир неког подскупа. Наиме, пошто је у претходном кораку било могуће добити све бројеве од 1 до збира свих елемената без тог новог, када у све те подскупове укључимо нови елемент добићемо све бројеве од тог новог елемента, па до збира свих елемената са тим новим елементом.

Пример. На пример, нека је дат низ $1, 2, 3, 5, 14, 20, 27$.

- 0 можемо добити као збир празног скупа $\{\}$.
- 1 можемо добити као збир скупа $\{1\}$.
- Када у претходне скупове укључимо и 2, можемо добити све бројеве закључно са 3 (2 као $\{2\}$ и 3 као $\{1, 2\}$).
- Када у претходне скупове укључимо и 3, можемо добити све бројеве закључно са 6 (4 као $\{1, 3\}$, 5 као $\{2, 3\}$ и 6 као $\{1, 2, 3\}$).
- Када у претходне скупове укључимо 5 можемо добити све бројеве закључно са 11 (7 као $\{2, 5\}$, 8 као $\{1, 2, 5\}$ и 9 као $\{1, 3, 5\}$, 10 као $\{2, 3, 5\}$ и 11 као $\{1, 2, 3, 5\}$).
- Пошто је наредни број 14, јасно је да се број 12 не може никако добити.

```
int n;
cin >> n;
```

```
// sabiranjem elemenata trenutnog sukpa mogu se dobiti svi elementi
// iz intervala [0, mozeDo]
int mozeDo = 0;
for (int i = 0; i < n; i++) {
  int x; cin >> x;
  if (x > mozeDo + 1)
    break;
  mozeDo += x;
}
cout << mozeDo + 1 << endl;
```

Анализа сложености. Програм се решава једним проласком кроз низ бројева и сложеност је прилично очигледно $O(n)$.

Доказ коректности. Докажимо и формално коректност, овог алгоритма тј. програма датог у прилогу.

Лема: Нека је m вредност промењиве `mozeDo`. Инваријанта петље је да је $0 \leq i \leq n$, да је m збир првих i елемената низа и да се сваки број из интервала $[0, m]$ може добити као збир неког подскупа првих i елемената низа.

Пре уласка у петљу је $i = 0$ и $m = 0$. Збир првих $i = 0$ елемената низа је по дефиницији нула (тј. m). Број 0 је једини елемент интервала $[0, m] = [0, 0]$ и он се може добити као збир празног подскупа (тј. 0 елемената полазног низа).

Претпоставимо да тврђење важи пре уласка у петљу.

- Ако је $a_i > m + 1$, тврдимо да је $m + 1$ тражени најмањи број. На основу инваријанте знамо да су сви бројеви из интервала $[0, m]$ покривени, тако да мањи број од $m + 1$ не може бити решење. Докажимо да број $m + 1$ не може бити збир подскупа. Пошто је низ сортиран, сви елементи од a_i до a_{n-1} су строго већи од $m+1$. Дакле, ни један од тих елемената не сме бити укључен у подскуп јер би њиховим укључивањем збир већ премашио $m + 1$. Подскуп се мора састојати само од елемената a_0 до a_{i-1}, међутим, пошто је m њихов збир, збир сваког њиховог подскупа је мањи или једнак m. Дакле, $m + 1$ се не може постићи и он је тражено решење.
- Ако је $a_i \leq m + 1$, тада је $m' = m + a_i$, $i' = i + 1$ и тврдимо да је m' збир свих елемената a_0, …, a_i и да се сваки број из интервала $[0, m']$ може представити као збир неког подскупа првих $i' = i + 1$ елемената низа. Прва тврдња је прилично очигледна, јер је по претпоставци m збир свих елемената a_0, …, a_{i-1}, а $m' = m + a_i$. На основу претпоставке знамо да сви бројеви из $[0, m]$ могу бити збирови подскупова првих i елемената низа. Слично и сви бројеви из интервала $[a_i, a_i + m]$ се могу добити као збир неког подскупа првих $i' = i + 1$ елемента низа. Наиме, тај подскуп ће бити унија елемента a_i и оног подскупа првих i елемената низа чији је збир једнак разлици између тог броја и броја a_i – он је из $[0, m]$, па на основу претпоставке такав подскуп постоји. Пошто је $a_i \leq m + 1$ унија интервала $[0, m]$ и $[a_i, a_i + m]$ је $[0, a_i + m] = [0, m']$. Зато је сваки елемент из $[0, m']$ једнак збиру неког подскупа првих i' елемената низа, па инваријанта остаје очувана.

Теорема: Случај када се петља заврши прекидом, јер је $a_i > m+1$ је већ размотрен. Када се петља заврши, важи да је $i = n$. На основу инваријанте m је збир свих елемената низа, и сваки број из $[0, m]$ јесте збир неког подскупа првих $i = n$ елемената низа, тј. целог низа. Зато је $m + 1$ најмањи елемент који није могуће добити (јер се укључивањем свих елемената добија највише m) и исписано решење је исправно.

Задатак: Бинарни запис

Напиши програм који на основу неозначеног целог броја n формира и исписује његов 32-битни бинарни запис.

Улаз: Са стандардног улаза се уноси број n ($0 \leq n \leq 2^{32} - 1$).

Излаз: На стандардни излаз исписати 32-битни бинарни запис броја n.

Пример 1

Улаз	*Излаз*
123456	00000000000000011110001001000000

Пример 2

Улаз	*Излаз*
`16777215`	`00000000111111111111111111111111`

Решење

Нека је низ од 32 логичке вредности попуњен вредношћу `false`. Бинарни запис одређујемо тако што одређујемо једну по једну бинарну цифру броја, здесна налево. У сваком кораку петље одређујемо остатак при дељењу броја n са 2, и на наредно место у низу (у кораку i на место i) уписујемо `true` ако је тај остатак једнак 1. На крају петље, исписујемо садржај низа уназад.

Пример. Прикажимо извршавање алгоритма на примеру превођења броја 38 у бинарни запис. Приказаћемо само кораке који се изводе док n не постане 0 (од тог тренутка надаље низ се само попуњава нулама).

```
n      niz b
38
19       0
9       10
4      110
2     0110
1    00110
0   100110
```

Имплементација се може направити на следећи начин.

```
// broj koji se prevodi
unsigned long n;
cin >> n;

// niz binarnih cifara, redom, od cifre najmanje do cifre najvece
// tezine
bool binarneCifre[32] = {false};
// prevodjenje
for (int i = 0; n > 0; i++, n /= 2)
  binarneCifre[i] = n % 2;

// ispisujemo rezultat (od cifre najmanje tezine
for (int i = 31; i >= 0; i--)
  cout << (binarneCifre[i] ? '1' : '0');
cout << endl;
```

Доказ коректности. Докажимо формално коректност овог алгоритма.

Да бисмо лакше одредили инваријанту проширимо пример извршавања програма вредношћу бинарног броја тренутно записаног у низу b и одговарајућим степеном двојке.

```
n      niz b    b     2^i
38              0     1
19       0      0     2
9       10      2     4
4      110      6     8
2     0110      6    16
1    00110      6    32
0   100110     38    64
```

Сада се лако може приметити да у сваком реду важи да је $2^i \cdot n + b = 38$ (заиста, важи да је $1 \cdot 38 + 0 = 2 \cdot 19 + 0 = 4 \cdot 9 + 2 = 8 \cdot 4 + 6 = 16 \cdot 2 + 6 = 32 \cdot 1 + 6 = 64 \cdot 0 + 38 = 38$.

Лема: Услов $2^i \cdot n + b = n_0$ је инваријанта петље, где је b број тренутно кодиран низом бинарних цифара (ако логичка вредност на позицији k у низу одговара цифри b_k, нека је $b = \sum_{k=0}^{31} b_k 2^k$), где је i текућа вредност променљиве `i`, док је n_0 почетна, а n текућа вредност неозначеног броја `n`.

- Заиста на почетку је $n = n_0$, $i = 0$ и $b = 0$ па тврђење важи.

- Претпоставимо да тврђење важи при уласку у петљу. Промењиве се током извршавања тела и корака петље мењају на следећи начин. $n' = n \operatorname{div} 2$, $b' = b + 2^i \cdot (n \bmod 2)$ и $i' = i + 1$. Тада је $2^{i'} \cdot n' + b' = 2^{i+1} \cdot (n \operatorname{div} 2) + b + 2^i \cdot n \bmod 2 = 2^i \cdot (2 \cdot (n \operatorname{div} 2) + n \bmod 2) + b$. На основу дефиниције целобројног дељења важи да је $2 \cdot (n \operatorname{div} 2) + n \bmod 2 = n$, па је вредност претходног израза једнака $2^i \cdot n + b$, а на основу претпоставке о томе да инваријанта важи на уласку у тело петље знамо да је то једнако n_0.

Теорема: По завршетку алгоритма низ садржи бинарни запис неозначеног броја `n`.

Како је по изласку из петље $n = 0$, на основу инваријанте важи да је $b = n_0$ тј. да низ садржи бинарни запис полазног броја.

Заустављање је прилично очигледно јер је `n` ненегативан број који се у сваком кораку стриктно смањује (све док не достигне нулу).

Анализа сложености. У овом облику алгоритма се увек спровди 32 корака, па је сложеност константна. У општем случају можемо сматрати да је линеарна у односу на број цифара у бинарном запису (она логаритамски зависи од величине броја n).

Задатак: Број формиран од датих цифра с лева на десно

Написати програм којим се формира природан број од учитаних цифара, ако се цифре броја учитавају слева на десно (редом од цифре највеће тежине до цифре јединица).

Улаз: Свака линија стандарног улаза, њих највише 9, садржи по једну цифру.

Излаз: На стандарном излазу приказати формирани број.

Пример

Улаз	*Излаз*

`4`
`1`
`0`
`9`
	`4109`

Решење

Читамо цифру по цифру са стандардног улаза, све док не дођемо до краја и формирамо број додавајући му прочитану цифру на десну страну (као цифру најмање тежине). Алгоритам се назива *Хорнерова шема* и на основу тог алгоритма број се гради тако што се у сваком кораку претходна вредност броја помножи са 10 и добијени производ се увећа за вредност наредне цифре (тако се цифра допише на десни крај претходног броја).

Пример. Анализирајмо пример у којем учитавамо редом цифре 3, 2, 7 и 5 и треба да добијемо број 3275. На основу дефиниције позиционог записа тај број је једнак вредности израза $3 \cdot 10^3 + 2 \cdot 10^2 + 7 \cdot 10 + 5$. Међутим, претходни израз можемо израчунати на следећи начин $((3 \cdot 10 + 2) \cdot 10 + 7) \cdot 10 + 5$, што доводи до Хорнеровог поступка. Ако се он имплементира итеративно, променљива `n` којом се представља вредност броја узима редом вредности 0, 3, 32, 327 и 3275.

Пошто број цифара није унапред познат, цифре ћемо учитавати у петљи којом учитавамо бројеве до краја стандардног улаза.

Број који формирамо памтићемо у променљивој `n` коју иницијизујемо на нулу. Када прочитамо цифру, додајемо је као цифру јединица на до сада формирани број. Додавање цифре јединица на број `n`, постижемо тако што помножимо броја `n` са 10 и додамо му учитану цифру. Ако унесемо k цифара, на описан начин цифру коју смо прву прочитали множићемо са 10 тачно $k - 1$ пут, другу прочитану цифру множимо са 10 тачно $k - 2$ пута, и тако редом, последњу прочитану цифру не множимо са 10 (то је цифра јединица).

```
int cifra;
int n = 0;
while (cin >> cifra)
  n = n * 10 + cifra;
cout << n << endl;
```

Доказ коректности. Докажимо и формално коректност Хорнерове схеме. Претпоставимо да ће се редом учитавати цифре a_{k-1}, a_{k-2}, …, a_1, a_0. Инваријанта петље је то да је n вредност броја који се добија

записом до сада прочитаних и обрађених цифара. Након i извршавања тела петље то су цифре од a_{k-1} до a_{k-i} и тврдимо да важи

$$n = (a_{k-1} \dots a_{k-i})_{10} = a_{k-1}10^{i-1} + a_{k-2}10^{i-2} + \dots + a_{k-i}10^0.$$

- *База*. Након 0 извршавања тела петље важи да је $n = 0$ и $i = 0$. Број $(a_{k-1} \dots a_{k-i})_{10} = (a_{k-1} \dots a_k)_{10}$ нема ниједну цифру и вредност му је нула.
- *Корак*. Претпоставимо да инваријанта важи на уласку у тело петље. Нека је $i' = i + 1$. Из тела петља види се да је $n' = 10 \cdot n + a_{k-i-1}$. Пошто на основу индуктивне хипотезе важи $n = a_{k-1}10^{i-1} + a_{k-2}10^{i-2} + \dots + a_{k-i}10^0$, важи и да је

  $n' = 10(a_{k-1}10^{i-1}+a_{k-2}10^{i-2}+\dots+a_{k-i}10^0)+a_{k-i-1} = a_{k-1}10^i+a_{k-2}10^{i-1}+\dots+a_{k-i}10+a_{k-i-1}$.

  Пошто је $i' = i + 1$, важи да је

  $n' = a_{k-i}10^{i'-1} + a_{k-2}10^{i'-2} + \dots + a_{k-i'+1}10 + a_{k-i'}$,

  па је инваријанта очувана.

Када се петља заврши, биће учитано k цифара, па ће важити да је $i = k$, тј.

$$n = (a_{k-1} \dots a_{k-i})_{10} = a_{k-1}10^{i-1} + a_{k-2}10^{i-2} + \dots + a_1 \cdot 10^1 + a_0 \cdot 10^0,$$

што значи да n садржи декадну вредност броја формираног од свих k цифара.

1.4 Додатни задаци за вежбу

Задатак: Аритметички троугао

Колики је збир бројева у датом реду следећег троугла?

```
         1
      2  3  4
   5  6  7  8  9
10 11 12 13 14 15 16
        ...
```

Улаз: Редни број k ($1 \leq k \leq 5 \cdot 10^5$), реда троугла чији збир треба израчунати (бројање редова почиње од 1).

Излаз: Збир вредности у задатом реду троугла.

Пример

Улаз	*Излаз*
3	35

Решење

Итерација

До решења се може доћи коришћењем петљи. У првој петљи одређујемо први елемент k-тог реда, а уз то одређујемо и број елемената у њему. Први елемент одређујемо тако што саберемо број елемената у претходних $k - 1$ редова, док број елемената сваког реда тако што у сваком кораку број елемената претходног реда увећавамо за два (сваки наредни ред има тачно два елемента више од претходног). Дакле, у петљи одржавамо почетак и број елемената текућег реда (иницијализујемо их на један, јер први ред почиње од један и има тачно један елемент) и $k - 1$ пута почетак реда увећавамо за број елемената текућег реда, а број елемената текућег реда за два.

Пример. Прикажимо ово на примеру одређивања првог елемента реда 5.

k	pocetak	brojElemenata
1	1	1
2	2	3

```
3   5         5
4   10        7
5   17        9
```

Након тога, у другој петљи одређујемо збир елемената у k-том реду, тако што на збир који иницијализујемо на нулу додајемо један по један елемент тог реда, коришћењем алгоритма сабирања серије бројева – елементи су узастопни природни бројеви, па их је лако набројати.

```
long long zbirRedaTrougla(long long k) {
  // odredjujemo prvi broj u k-tom redu trougla
  long long pocetak = 1;
  long long brojElemenata = 1;
  for (int i = 1; i < k; i++) {
    pocetak += brojElemenata;
    brojElemenata += 2;
  }

  // odredjujemo zbir elemenata u k-tom redu trougla
  long long zbir = 0;
  for (long long i = pocetak; i < pocetak + brojElemenata; i++)
    zbir += i;
  return zbir;
}
```

Анализа сложености. У првој петљи обилазимо k редова којима одређујемо почетак помоћу две операције сабирања. Пажљивом анализом можемо закључити да у реду k има $2k - 1$ елемената (мада ту чињеницу нисмо употребили у програму), који се у другој фази сабирају. Сложеност обе фазе, па и укупног решења је, дакле, $O(k)$. Задатак, наравно, може да се реши и у мањој сложености од ове, без коришћења петљи.

Доказ коректности. Докажимо формално коректност овог поступка. Инваријанта петље је да након m извршавања њеног тела важи да је $i = m+1$ и $1 \leq i \leq k$, као и да променљива `pocetak` садржи први елемент реда $i = m + 1$, а променљива `brojElemenata` садржи број елемената реда $i = m + 1$.

- Пошто се променљива `i` иницијализује на вредност 1, након $m = 0$ корака важи да је $1 = 0 + 1$ и $1 \leq 1 \leq k$. Пошто су и обе променљиве иницијализоване на вредност 1, након $m = 0$ корака петље, променљиве заиста садрже први елеменат и број елемената реда $i = m + 1 = 0 + 1 = 1$.
- Претпоставимо да инваријанта важи након m корака петље и да је услов петље испуњен тј. да је $i < k$.

  Докажимо да тврђење важи и након $m' = m + 1$ извршавања тела и корака петље. Пошто након извршавања корака важи $i' = i + 1$, а пошто је на основу индуктивне хипотезе важило да је $i = m + 1$, важи и да је $i' = m' + 1$. Пошто је услов петље био испуњен, важило је $i < k$, па уз индуктивну претпоставку $1 \leq i \leq k$, важи $1 \leq i' \leq k$.

  Означимо са p и b вредности променљивих `pocetak` и `brojElemenata` на улазу у тело петље, а са p' и b' њихове нове вредности, након извршења тела и корака петље. Анализом додела у телу петље, јасно видимо да је $p' = p+b$ и $b' = b+2$. На основу претпоставке важи да је p први елемент реда $i = m+1$, а да је b број елемената реда $i = m+1$. Сабирањем првог елемента било ког реда у троугла и броја елемената тог реда троугла добија се први елемент наредног реда троугла. Дакле, важи да је $p + b$ први елемент реда $i+1 = m+2$ троугла, па, важи да је $p' = p+b$ први елемент реда $i' = i+1 = m+2 = m'+1$. На основу дефиниције троугла важи јасно и да сваки наредни ред има и два елемента више него претходни. Зато, пошто је b број елемената реда $m + 1$, важи и да је $b' = b + 2$ број елемената реда $i' = m' + 1$.
- На крају петље услов није испуњен, па важи да је $i \geq k$. Уз услов $1 \leq i \leq k$, мора да важи да је $i = k$. На основу инваријанте знамо да променљива `pocetak` садржи број елемената реда $i = m + 1 = k$. Дакле, петља је коректно извршила свој задатак и у променљиву `pocetak` сместила први елемент реда k.

На сличан начин се може формално доказати и коректност друге петље (инваријанта је да након m њених корака променљива `zbir` садржи збир првих m елемената реда k, а да променљива i садржи вредност елемента на позицији m у том реду, ако се позиције броје од 0).

Напомена. Приметимо да смо у овој петљи увели и нову променљиву којом смо регистровали број елемената у текућем реду троугла. То што смо ту променљиву имали на располагању нам је помоглно да ажурирамо

вредност почетног елемента наредног реда, међутим, "кредит" који смо добили на почетку тела петље морали смо да вратимо на крају тако што смо морали да ажурирамо и вредност те променљиве (и тако је припремимо за наредну итерацију). Ова техника је позната под именом **ојачавање индуктивне хипотезе** и често се користи приликом конструкције алгоритама.

Задатак: Двобојка

Напиши програм који организује елементе низа тако да прво иду сви парни елементи, а затим непарни, при чему међусобни редослед парних и непарних елемената није битан. Елементе прво учитати у низ, а затим тај низ трансформисати у линеарном времену (само једним пролазом кроз низ).

Улаз: У првој линији стандардног улаза унети природан број n ($1 \leq n \leq 50000$) - број елемената низа, а у наредној линија унети n природних бројевава у границама од 1 до 1000.

Излаз: На станардни излаз исписати елементе низа уређене на тражени начин, раздвојене са по једним размаком.

Пример

Улаз	*Излаз*
`10`	`2 6 8 10 4 5 3 1 9 11`
`2 5 3 6 1 8 9 10 11 4`	

Решење

Трансформације низа "у месту" - два показивача

Постоји неколико начина да се низ "у месту" трансформише само једним проласком кроз низ (ти приступи имају линеарну временску сложеност). Приказаћемо неколико могућности, све засноване на техници два показивача.

Парни лево, непарни десно - размена наопаких

Претпоставићемо да су у сваком кораку петље познати индекси l и d тако да су елементи низа груписани тако да су:

- сви елементи на позицијама из интервала $[0, l)$ парни,
- сви елементи на позицијама из интервала $[l, d]$ још непрегледани и
- сви елементи на позицијама из интервала (d, n) непарни.

Инваријанта је, дакле, да је распоред елемената у низу облика `ppp???nnn`, где су `l` и `d` позиција првог тј. последњег непознатог елемента (обележеног упитником).

На почетку иницијализујемо l на нулу, а d на $n-1$ (тада су интервали $[0, l)$ и (d, n) празни, а сви елементи у интервалу $[l, d] = [0, n-1]$ су још непрегледани). Петљу у приниципу извршавамо док још има непрегледаних елемената тј. док је $l \leq d$, међутим, у овом задатку можемо је завршити и корак раније. Петља се извршава док је $l < d$ и завршава када је $l = d$, јер какав год да је тај последњи непрегледани елемент на позицији $l = d$, он ће се моћи припојити или левом или десном делу низа и неће бити потребе премештати га.

У телу петље радимо следеће:

- Прво испитујемо да ли је елемент на позицији l паран и ако јесте, онда само увећавамо вредност l за 1.
- У супротном, проверавамо да ли је елемент на позицији d непаран и ако јесте, онда само умањујемо вредност d за 1.
- На крају, ако ниједна од претходне две провере није успела, знамо да је елемент на позицији l непаран, а елемент на позицији d паран, размењујемо их, увећавамо l за 1 и умањујемо d за 1.

Када се петља заврши, елементи су у жељеном редоследу.

Пример. Прикажимо рад алгоритма на једном примеру.

```
l                 d
3  8  7  4  5  1  6  2

   l              d
2  8  7  4  5  1  6  3
```

```
      l           d
2  8  7  4  5  1  6  3

         l     d
2  8  6  4  5  1  7  3

            l  d
2  8  6  4  5  1  7  3

            ld
2  8  6  4  5  1  7  3
```

Имплементација се може направити на следећи начин.

```
// odrzavamo uslov
// [0, l) - parni
// (d, n) - neparni
// [l, d] - nepoznati

// u pocetku su svi nepoznati
int l = 0, d = n-1;
// dok jos ima nepoznatih elemenata
while (l < d) {
  // ako je na mestu l paran, ostavljamo ga na svom mestu i pomeramo
  // se na naredni element
  if (a[l] % 2 == 0)
    l++;
  // ako je na mestu d neparan, ostavljamo ga na svom mestu i
  // pomeramo se na prethodni element
  else if (a[d] % 2 != 0)
    d--;
  else
    // na mestu l je neparan, a na mestu d je paran broj, pa ih
    // razmenjujemo i pomeramo se po oba kraja
    swap(a[l++], a[d--]);
}
```

Анализа сложености. Променљива `l` се увећава, а променљива `d` се умањује све док се не сусретну, што се догађа у тачно n корака, па је укупна сложеност алгоритма $O(n)$.

Доказ коректности. Докажимо и формално коректност претходног алгоритма. Током извршавања важи раније описана инваријанта, а важи и да је $0 \leq l \leq d + 1 \leq n$.

Иницијализацијом вредности l на нулу, а вредности d на $n - 1$ постижемо да су ови услови на почетку задовољени (јер су сви елементи у интервалу $[l, d] = [0, n - 1]$ још непрегледани).

У телу петље радимо следеће:

- Прво испитујемо да ли је елемент на позицији l паран и ако јесте, онда само увећавамо вредност l за 1 чиме наметнута инваријанта остаје да важи. Заиста, парни су били сви елементи из интервала $[0, l)$, паран је и елемент на позицији l, па су парни сви елементи из интервала $[0, l] = [0, l')$, где је $l' = l + 1$, нова вредност променљиве l. Пошто је $d' = d$, елементи на позицијама из интервала (d', n) су непарни.
- У супротном, проверавамо да ли је елемент на позицији d непаран и ако јесте, онда само умањујемо вредност d за 1 чиме наметнута инваријанта опет остају на снази (аргументација је слична оној у претходном случају).
- На крају, ако ниједна од претходне две провере није успела, знамо да је елемент на позицији l непаран, а елемент на позицији d паран, размењујемо их, увећавамо l за 1 и умањујемо d за 1 чиме инваријанта остаје да важи. Заиста, важи да је $l' = l + 1$ и $d' = d - 1$. Сви елементи из интервала позиција $[0, l)$ су парни, а паран је елемент на позицији l (јер је разменом паран елемент са позиције d доведен на позицију l). Зато су парни сви елементи у интервалу $[0, l')$. Слично, пошто су сви елементи у интервалу

позиција (d, n) били непарни, а пошто је након размене непарни елемент са позиције l доведен на позицију d, непарни су и сви елементи из интевала $[0, d')$.

Када се петља заврши, распоред је коректан. Наиме, пошто на крају петље услов $l < d$ није испуњен, а пошто је $l \leq d + 1$, тада је $l = d$ или је $l = d + 1$.

- Ако је $l = d + 1$, знамо да је низ разбијен на сегмент парних елемената на позицијама $[0, l)$ и непарних на позицијама $(d, n) = [d + 1, n) = [l, n)$.
- Размотримо случај $l = d$.
  - Ако је елемент на позицији l паран, пошто су на основу инваријанте парни и сви елементи на позицијама $[0, l)$, знамо да ће парни бити сви елементи на позицијама $[0, l]$, док ће елементи на позицијама $(d, n) = [l + 1, n)$ бити непарни.
  - Слично, ако је елемент на позицији l непаран, тада су парни сви елементи на позицијама $[0, l)$, а непарни су сви елементи на позицијама $[l, n)$ (јер на основу инваријанте знамо да су још непарни и елементи на позицијама $(d, n) = (l, n)$).

  У оба случаја је, дакле, постигнут жељени распоред елемената.

Приметимо да смо прекидом петље када је $l = d$, уштедели једну итерацију петље (што није нарочито значајно), али смо закомпликовали доказ коректности, па се природно поставља питање колико је та оптимизација имала смисла.

Парни, непарни, па непознати

У овом решењу инваријанта је мало другачија. Памтимо индекс k и текући индекс i и претпостављамо да су

- елементи на позицијама из интервала $[0, k)$ парни,
- елементи из интервала $[k, i)$ непарни,
- елементи из интервала $[i, n)$ још непрегледани.

Дакле, намећемо инваријанту да је распоред облика `pppnnn???`, где је i позиција првог непознатог, а k позиција првог непарног елемената.

Размотримо како да из инваријанте закључимо како треба иницијализовати променљиве. Пошто су сви елементи из интервала $[i, n)$ непрегледани, променљиву i морамо инцијализовати на 0. Пошто су сви елементи из интервала $[0, k)$ парни, а $[k, i)$ непарни, и k морамо поставити на 0.

Инваријанта јасно диктира и услов петље. Наиме, петља се извршава док још има непрегледаних елемената тј. док је $i < n$. У сваком кораку петље i се увећава за 1 (користимо класичну бројачку петљу `for` по променљивој i), чиме се сужава интервал непрегледаних елемената.

Размотримо како треба да изгледа тело петље да би инваријанта остала испуњена.

- Ако је елемент на текућој позицији i паран, онда га размењујемо са првим непарним елементом, а то је елемент на позицији k. Изузетак је случај када је $k = i$, када заправо не долази до размене (елемент се мења сам са собом). У оба случаја се k увећава за 1.
- У супротном, елемент на позицији i је непаран он остаје на свом месту и у телу петље није потребно ништа урадити (грану `else` у коду није потребно наводити).

Пример. Прикажимо рад алгоритма на једном примеру.

```
k                   n
3  8  7  4  5  1  6  2
i

k                   n
3  8  7  4  5  1  6  2
   i

   k                n
8  3  7  4  5  1  6  2
      i
```

```
   k                 n
8  3  7  4  5  1  6  2
      i

      k              n
8  4  7  3  5  1  6  2
         i

      k              n
8  4  7  3  5  1  6  2
            i

      k              n
8  4  7  3  5  1  6  2
               i

         k           n
8  4  6  3  5  1  7  2
                  i

            k         n
8  4  6  2  5  1  7  3
                     i
```

Имплементација се може направити на следећи начин.

```
// odrzavamo uslov
// [0, k) - parni
// [k, i) - neparni
// [i, n) - nepoznati

// u pocetku su svi elementi nepoznati
int k = 0;
for (int i = 0; i < n; i++)
  // ako je element paran razmenjujemo ga sa prvim neparnim
  if (a[i] % 2 == 0)
    swap(a[i], a[k++]);
  // a ako je neparan, ne pomeramo ga
```

Анализа сложености. У програму се користи класична бројачка петља `for` која се завршава у n корака, па је укупна сложеност алгоритма $O(n)$.

Доказ коректности. Докажимо и формално коректност претходног поступка. Уз описану инваријанту важи и да је $0 \leq k \leq i \leq n$.

Пошто је на почетку $i = k = 0$, важи да су сви елементи у интервалу $[i, n) = [0, n)$ непрегледани. Интервали $[0, k) = [k, i) = [0, 0)$ су празни, па задовољавају услове, а тривијално Важи и да је $0 \leq k \leq i \leq n$.

У телу петље се врше следеће акције.

- Ако је елемент на текућој позицији i паран, онда га размењујемо са првим непарним елементом, а то је елемент на позицији k. Изузетак је случај када је $k = i$, када заправо не долази до размене (елемент се мења сам са собом). У оба случаја се k увећава за 1. Дакле, на основу инваријанте знамо да су елементи на позицијама $[0, k)$ парни, да је након размене елемент на позицији k паран, па су парни и сви елементи на позицијама $[0, k') = [0, k + 1)$. Елементи на позицијама $[k', i') = [k + 1, i + 1)$ су непарни. Наиме, ако је $k = i$, овај интервал је празан, а ако је $k < i$, тада је пре размене елемент на позицији k био непаран (јер на основу инваријанте знамо да су сви елементи на позицијама $[k, i)$ били непарни, па самим тим и елемент на позицији k, који је сада доведен на позицију i).
- У супротном, елемент на позицији i је непаран он остаје на свом месту. Пошто се у телу петље не дешава ништа, важи $k' = k$ и $i' = i$, а инваријанта прилично очигледно остаје на снази.

По завршетку петље услов $i < n$ није испуњен, па пошто на основу инваријанте важи $0 \leq k \leq i \leq n$, важи да

је $i = n$. Зато је интервал непознатих $[i, n)$ празан, сви елементи из интервала $[0, k)$ су парни, из интервала $[k, i) = [k, n)$ су непарни и постигнут је тражени распоред.

Заустављање се једноставно доказује јер се у сваком кораку сужава интервал непознатих $[i, n)$.

Задатак: Разлика сума до мах и од мах

Уноси се масе предмета, одредити разлику суме маса предмета до првог појављивања предмета највеће масе и суме маса предмета после првог појављивања предмета највеће масе (предмет највеће масе није укључен ни у једну суму).

Улаз: У првој линији стандардног улаза налази се број предмета n ($1 \leq n \leq 50000$). Свака од наредних n линија садржи по један природан број из интервала $[1, 50000]$, ти бројеви представљају масе сваког од n предмета.

Излаз: У првој линији стандарног излаза приказати тражену разлику маса.

Пример

Улаз	*Излаз*
5 10 13 7 13 4	-14

Објашњење

Предмет највеће масе је 13. Збир маса пре његовог првог појављивања је 10, а после његовог првог појављивања је 24. Зато је тражена разлика -14.

Решење

Директно решење

Решење до којег се најједноставније долази је да се масе свих предмета учитају у низ, да се након тога пронађе позиција највећег елемента у низу и да се затим одреди збир елемената низа пре те позиције и збир елемената низа после те позиције.

Оптимизовано решење

Задатак можемо решити и без коришћења низова. У овом решењу задатка ћемо комбиновати алгоритам одређивања максимума серије бројева и алгоритам сабирања серије учитаних бројева. У петљи ћемо учитавати једну по једну масу предмета, одржавајући при том збир маса пре прве појаве максималне масе, збир маса после прве појаве максималне масе и саму вредност максималне масе предмета.

На почетку учитавамо масу првог предмета (он постоји по условима задатка), максимум иницијализујемо на ту учитану вредност, а два збира на нулу (јер није виђен ни један предмет ни пре, ни после тог првог предмета који је уједно и прво појављивање до тада максималне масе).

Затим, у петљи учитавамо остале предмете, један по један. Могућа су два случаја.

- Ако је маса учитаног предмета строго већа од до тада максималне масе, онда тај предмет представља прво појављивање предмета максималне масе (он је максимални од свих до тада виђених). Зато је потребно да израчунамо збир маса свих предмета пре њега. Међутим, ми знамо вредност збира маса свих предмета пре ранијег предмета максималне масе, знамо његову масу и знамо збир маса предмета након њега (у тај збир још није укључен текући предмет), тако да је збир предмета пре текућег предмета (новог максимума) једнак збиру ове три вредности. Максимум се ажурира на масу текућег предмета, а збир маса предмета после новог максимума се ажурира на нулу (јер за сада нисмо видели ни један такав предмет).
- У супротном, текући предмет не представља прво појављивање максимума (он је или мањи од максимума, или је једнак максимуму, али није његово прво појављивање). Зато се предмет максимум не мења, не мења се серија предмета пре њега, док се серија предмета након њега продужава текућим предметом. Зато је у овом случају само потребно ажурирати збир маса предмета после максимума тако што се тај збир увећа за масу текућег предмета.

Доказ коректности. Докажимо формално коректност овог алгоритма. Инваријанту петље чине следећи услови:

- $1 \leq i \leq n$.
- Променљива `max` садржи вредност максимума првих i елемената низа (елемената $a_0, \ldots, a_{i-1}$). Претпостављамо да се прво појављивање те вредности јавља на некој позицији m између 0 и $i-1$ тј. да су сви елементи $a_0, \ldots, a_{m-1}$ строго мањи од вредности променљиве `max`, да a_m има вредност `max`, а да сви елементи $a_{m+1}, \ldots, a_{i-1}$ имају вредност већу или једнаку од променљиве `max`.
- Променљива `zbirPreMax` садржи збир свих елемената пре првог појављивања максималног елемента `max` тј. вредност $a_0 + \ldots + a_{m-1}$.
- Променљива `zbirPosleMax` садржи збир свих елемената после првог појављивања максималног елемента `max` тј. вредност $a_{m+1} + \ldots + a_{i-1}$.

Докажимо да су услови заиста инваријанта петље.

- Пре уласка у петљу, променљива `max` има вредност a_0, променљиве `zbirPreMax` и `zbirPosleMax` имају вредност 0, а променљива `i` вредност 1. Тада је $m = 0$, па су сви услови испуњени (скупови елемената $a_0, \ldots, a_{m-1}$ и $a_{m+1}, \ldots, a_{i-1}$ су празни).
- Претпоставимо да услови важе пре уласка у петљу.
  - Претпоставимо да је a_i строго веће од вредности `max`. Тада на основу инваријанте знамо да је a_i строго веће од свих вредности $a_0, \ldots, a_{i-1}$, па се нова вредност максимума јавља на позицији $m' = i$.

    Тада је нова вредност променљиве `zbirPreMax` једнака збиру старих вредности променљивих `zbirPreMax`, `zbirPosleMax` и `max`, па на основу варијанте важи да је та нова вредност једнака $(a_0 + \ldots + a_{m-1}) + a_m + (a_{m+1} + \ldots + a_{i-1})$, што је једнако $a_0 + \ldots + a_{m'-1}$. Ово је тачно збир свих елемената пре нове позиције максимума за коју смо утврдили да је позиција $m' = i$. Уједно за све ове елементе важи да су строго мањи од `max`.

    Нова вредност променљиве `max` једнака је a_i што јесте нова вредност максимума.

    Нова вредност променљиве `zbirPosleMax` једнака је 0, што је једнако збиру вредности $a_{m'+1} + \ldots + a_{i-1}$ (јер је $m' = i$, па је тај скуп елемената празан).
  - Претпоставимо да је a_i мање или једнако од вредности `max`. Тада се прво појављивање максимума у делу низа $a_0, \ldots, a_i$ поклапа са првим појављивањем максимума у делу низа $a_0, \ldots, a_{i-1}$ и налази се такође на позицији m, па је $m' = m$.

    Променљиве `max` и `zbirPreMax` не мењају своје вредности (што је у реду, јер је $m' = m$, па је вредност `max` једнака $a_m = a_{m'}$, док је вредност променљиве `zbirPreMax` једнака $a_0 + \ldots + a_{m'-1} = a_0 + \ldots + a_{m-1}$).

    Променљива `zbirPosleMax` увећава своју вредност за a_i, па, пошто је њена стара вредност била једнака $a_{m+1} + \ldots a_{i-1}$, пошто је $m' = m$ и $i' = i + 1$, важи да је њена нова вредност једнака $a_{m+1} + \ldots + a_i = a_{m'+1} + \ldots + a_{i'-1}$.

На крају петље не важи да је $i < n$, па пошто је $i \leq n$, важи да је $i = n$. Зато на основу инваријанте знамо да променљиве `zbirPreMax` `zbirPosleMax` заиста имају вредности збира свих елемената пре првог и после првог појављивања максималног елемента у делу низа $a_0 \ldots a_{i-1}$, што је заправо цео низ (јер је $i = n$, а n је укупан број елемената).

```
// ukupan broj predmeta
int n;
cin >> n;

// ucitavanje mase prvog predmeta
int m;
cin >> m;
// najveca do sada ucitana masa
int max = m;
// zbir masa pre najvece do sada ucitane
int zbirPreMax = 0;
// zbir masa posle najvece do sada ucitane
```

```
int zbirPosleMax = 0;
for (int i = 1; i < n; i++) {
  // ucitavanje mase sledeceg predmeta
  cin >> m;
  if (m > max) {
    // korekcija najvece mase i zbirova
    zbirPreMax += max + zbirPosleMax;
    max = m;
    zbirPosleMax = 0;
  } else
    zbirPosleMax += m;
}

// prikaz rezultata
cout << zbirPreMax - zbirPosleMax << endl;
```

Задатак: Обртање цифара

Улаз: Са стандардног улаза се уноси природан број n ($0 \leq n \leq 10^9$).

Излаз: На стандардни излаз исписати број који се добија обртањем цифара броја n.

Пример 1		**Пример 2**	
Улаз	*Излаз*	*Улаз*	*Излаз*
12345	54321	12000	21

Решење

У овом поступку се комбинују два алгоритма. Први је алгоритам за одређивања цифара у декадном запису броја, здесна налево, одређивањем остатка при целобројном дељењу. Други алгоритам је Хорнерова схема за одређивање вредности броја на основу његових цифара слева надесно. Тај алгоритам је описан у задатку Број формиран од датих цифра с лева на десно. Скидамо једну по једну цифру броја n здесна и дописујемо је на резултат, такође здесна.

Пример. Размотримо поступно рад алгоритма на једном примеру.

n	r	i
123000	0	0
12300	0	1
1230	0	2
123	0	3
12	3	4
1	32	5
0	321	6

Коректност овог алгоритма се заснива на следећој инваријанти. Након i корака петље број n садржи број који се добије када се од почетног броја n_0 обрише последњих i цифара, док број r садржи број који се добије обртањем тих последњих i цифара броја n_0.

Имплементација се може направити на следећи начин.

```
int obrni(int n) {
  int r = 0;
  while (n > 0) {
    r = 10*r + n % 10;
    n /= 10;
  }
  return r;
}
```

Доказ коректности. Пошто је програм написан у терминима математичких операција, ако желимо да спроведемо потпуно прецизан, математички формалан доказ, и инваријанту морамо изразити у терминима математичких операција. Нажалост, испоставиће се да крајње нуле у запису броја, које постају водеће нуле у

резултату значајно компликују прецизно изражавање инваријанте и наставак доказа чине прилично мукотрпним. Наиме, број који се обрће може се завршити нулама, тако да на пример бројеви 123, 1230, 12300 сви имају исту вредност броја који се записује истим цифрама као оригинални број али у обрнутом поретку (321). Али број корака петље који ће бити потребан да се та вредност израчуна ће се разликовати у тим ситуацијама (и биће једнак броју цифара броја n). Отуда је битно приметити да је број завршних нула у запису броја битан за предложени алгоритам.

Лема: Услов да је $n_0 = n \cdot 10^i + rev(r) \cdot 10^s$ (где је s број завршних нула у запису броја n_0) је инваријанта петље. При том је број r једнак 0 док је $i \leq s$ (има 0 цифара у запису), односно има тачно $i - s$ цифара у запису када је $i > s$ и водећа цифра му је различита од нуле. Важи и да је $n \geq 0$.

Докажимо ову лему.

- Пре првог уласка у петљу је $n = n_0$, $r = 0$ и $i = 0$, па услов очигледно важи.
- Претпоставимо да тврђење важи на уласку у петљу. Ефекат тела петље је да је $n' = n \operatorname{div} 10$, да је $r' = 10r + n \bmod 10$ и да је $i' = i + 1$. Потребно је да докажемо да вредност $n' \cdot 10^{i'} + rev(r') \cdot 10^s$ остаје једнака n_0. Након извршавања тела петље, она је једнака $(n \operatorname{div} 10) \cdot 10^{i+1} + rev(10r + n \bmod 10) \cdot 10^s$. Ову вредност даље израчунавамо у зависности од односа променљивих i и s.
  - Ако је $i < s$, петља скида нуле са краја записа броја n, тада је $r = 0$ и $n \bmod 10 = 0$. Зато је претходна вредност једнака $(n \operatorname{div} 10) \cdot 10^{i+1} = (10 \cdot (n \operatorname{div} 10) + n \bmod 10) \cdot 10^i = n \cdot 10^i$, што је на основу претпоставке једнако n_0 (јер је и у претходном кораку $r = 0$).
  - Ако је $i = s$ петља је стигла до прве цифре различите од нуле, па је још увек $r = 0$, а $n \bmod 10 \neq 0$. Пошто је $n \bmod 10$ једноцифрен број, важи да је $rev(n \bmod 10) = n \bmod 10$. Пошто је $i = s$, вредност израза који фигурише у инваријанти једнака је $(n \operatorname{div} 10) \cdot 10^{i+1} + (n \bmod 10) \cdot 10^i = (10 \cdot (n \operatorname{div} 10) + n \bmod 10) \cdot 10^i = n \cdot 10^i$, а то је на основу претпоставке једнако n_0 (јер је и у претходном кораку $r = 0$). Важи и да је $r' = n \bmod 10$, па је $0 < r' < 10$ и r' је једноцифрен број. Зато важи да r' има тачно $i' - s = i + 1 - i = 1$ цифара и да му је водећа цифра различита од нуле.
  - Ако је $i > s$, пошто r има тачно $i - s$ цифара, тада важи да је $rev(10r + n \bmod 10) = (n \bmod 10) \cdot 10^{i-s} + rev(r)$. Заиста, израз са леве стране означава број који се добија обртањем редоследа цифара броја који се од броја r добија дописивањем цифре $n \bmod 10$ (она може бити и нула) с десне стране. Што се тиче израза са десне стране, на основу инваријанте знамо да је r број који има тачно $i - s$ цифара и оне одговарају степенима од 10^0 до 10^{i-s-1}. Зато се цифра $n \bmod 10$ множењем са фактором 10^{i-s} поставља као прва цифра иза које следе све цифре записа броја r у обрнутом редоследу. Дакле, леви и десни израз имају исту вредност. На основу тога, знамо да је вредност израза који фигурише у инваријанти након извршавања тела једнака $n' \cdot 10^{i'} + rev(r') \cdot 10^s$ = $(n \operatorname{div} 10) \cdot 10^{i+1} + rev(10r + n \bmod 10) \cdot 10^s$ = $(n \operatorname{div} 10) \cdot 10^{i+1} + ((n \bmod 10) \cdot 10^{i-s} + rev(r)) \cdot 10^s$ = $(10 \cdot (n \operatorname{div} 10) + n \bmod 10) \cdot 10^i + rev(r) \cdot 10^s$ = $n \cdot 10^i + rev(r) \cdot 10^s$, што је једнако n_0 на основу претпоставке. Пошто је $r' = 10r + n \bmod 10$, онда важи да r' има једну цифру више него r (тј. има $i - s + 1 = i' - s$ цифара), а водећа цифра је различита од нуле (водећа цифра се није променила).

Теорема: Након извршавања кода важи да је $r = rev(n_0)$.

Пошто је $n \geq 0$ и није $n > 0$ (јер се петља завршила), важи да је $n = 0$. Зато је $n_0 = rev(r) \cdot 10^s$. тј. $rev(r) = \frac{n_0}{10^s}$. Пошто је водећа цифра броја r различита од нуле и пошто је последња цифра броја $\frac{n_0}{10^s}$ такође различита од нуле (јер n_0 има тачно s завршних нула које се дељењем са 10^s бришу), важи да је $r = rev(\frac{n_0}{10^s})$. Међутим, важи и да је $rev(n_0) = rev(\frac{n_0}{10^s})$, јер $rev(n_0)$ има тачно s водећих нула које се могу обрисати без утицаја на коначни резултат.

Заустављање се доказује прилично једноставно јер је $n \geq 0$ и константно се смањује, па мора доћи до нуле.

Задатак: Растављање на просте чиниоце

Ако је дато неколико простих бројева, њихов производ се може веома лако и брзо одредити. Међутим, ако је дат производ, често је веома тешко одредити просте бројеве који га сачињавају. Напиши програм који што ефикасније решава тај проблем.

Улаз: Са стандардног улаза се уноси један природан број n ($1 \leq n \leq 2 \cdot 10^9$).

Излаз: На стандардни излаз исписати просте чиниоце броја n, уређене од најмањих до највећих, раздвојене размаком.

Пример

Улаз	*Излаз*
900	2 2 3 3 5 5

Решење

Алгоритам факторизације

Потенцијални чиниоци f броја n се испитују редом, у петљи, кренувши од броја 2. У сваком кораку испитује се да ли је број n дељив бројем f и док год јесте дељив, у унутрашњој петљи, он се дели бројем f приј-ављујући при том чинилац f (пошто се у склопу услова унутрашње петље врши провера дељивости n са f, није потребна посебна провера дељивости наредбом гранања пре те петље). Након тога прелази се на следећи потенцијални чинилац (за један већи од претходног). Иако се може помислити да је за сваки потенцијални чинилац потребно проверити да ли је он прост број (јер нас занимају само прости чиниоци), то није потребно радити. Наиме, у поступку претраге који смо навели, ако текући кандидат f није прост, он не може да дели број n, јер смо све његове просте чиниоце већ дељењем уклонили из броја n. На пример, када f достигне вредност 6, број n не може бити дељив њиме јер је претходно исцрпно издељен бројем 2 (а касније и бројем 3). Заиста, ако претпоставимо супротно да f дели n и да је f сложен број, тада би f имао неки прости чинилац мањи од њега и то би уједно био прост чинилац броја n. Међутим, то није могуће јер смо пре увећања броја f на његову текућу вредност утврдили да текући број n не може бити дељив ни једним бројем мањем од f (иначе бисмо га делили са f, а не увећавали f). Дакле сложени чиниоци се елиминишу тако што се утврди да текући број n није дељив њима, што је много ефикасније него примењивати на њих тест простости.

У најједноставнијој имплементацији, описани поступак траје све док се број n дељењем својим чиниоцима не сведе на број 1.

Анализа сложености. Иако коректан, алгоритам који се завршава свођењем броја на 1 је прилично неефикасан и за бројеве који имају велике просте чиниоце ради веома споро (покушајте извршавање програма нпр. за број 1000000007). Проблем настаје јер се делиоци последњег простог чиниоца испитују све док се не дође до самог тог броја. С обзиром на ограничење бројевног типа, број чинилаца можемо сматрати практично константним (не може их бити више од 32), па је сложеност $O(f_k)$, где је f_k највећи прост фактор полазног броја, што је $O(n)$ када је n прост број.

На срећу, алгоритам је једноставно поправити, тако што се растављање заустави чим се утврди да је текућа вредност променљиве n прост број (а видећемо да за то није потребно чекати да вредност f достигне n).

Пример. Прикажимо рад алгоритма на једном примеру.

n	f	cinioc
3300	2	2
1650	2	2
825	2	-
825	3	3
275	3	-
275	4	-
275	5	5
55	5	5
11	5	-
11	6	-
11	7	-
11	8	-
11	9	-
11	10	-
11	11	11
1	11	-

Имплементација се може направити на следећи начин.

```
// ucitavamo broj koji treba rastaviti na proste cinioce
int n;
cin >> n;
int f = 2; // prvi potencijalni prost cinilac je 2
// dok se broj deljenjem sa svojim prostim ciniocima ne svede na 1
```

```
while (n > 1) {
  while (n % f == 0) {   // dok je n deljivo sa f
    cout << f << " ";    //   prijavljujemo pronađeni prost cinilac
    n /= f;              //   delimo broj njime
  }
  f = f + 1;             //   prelazimo na sledeceg kandidata
}
cout << endl;
```

Доказ коректности. Докажимо коректност претходног алгоритма и формално, коришћењем технике инваријанти петље. Централна инваријанта петље је то да текућа вредност промењљиве n није дељива ни једним бројем из интервала $[2, f)$, као и да је почетни број n_0 производ до сада исписаних простих бројева и текуће вредности промењљиве n.

Пре уласка у петљу је $f = 2$, па је интервал $[2, f)$ празан и први део инваријанте тривијално важи. Важи и да је $n = n_0$ и да ниједан број није исписан, па и други део инваријанте важи.

Претпоставимо да инваријанта важи на уласку у петљу.

- Ако је n дељив бројем f, исписује се број f. Он је очигледно чинилац броја n. Претпоставимо да је сложен и да се може раставити као $f = f_1 \cdot f_2$, за $f_1 > 1$ и $f_2 > 1$. Тада би број n био дељив са f_1 и са f_2 који припадају интервалу $[2, f)$, што је супротно инваријанти, па закључујемо да d мора бити прост. Дељењем броја n са f добија се нови број n', који такође није дељив ни са једним бројем из $[2, f)$, па први део инваријанте остаје очуван. Почетна вредност n_0 остаје једнака производу исписаних бројева и текуће вредности n (јер је то важило на основу инваријанте, исписано је f, а n је подељено са f), па и други део инваријанте остаје очуван.
- Ако n није дељив бројем f, тада се f увећава за 1 (нека је $f' = f + 1$). Да би први део инваријанте био одржан, треба да важи да n није дељив ни са једним бројем из интервала $[2, f') = [2, f]$, но то важи, јер на основу инваријанте знамо да n није дељив ни са једним бројем из интервала $[2, f)$, а на основу експлицитне провере услова знамо да n није дељиво ни са f. Ниједан број није исписан, нити је број n промењен, па други део инваријанте тривијално наставља да важи.

На основу инваријанте знамо да је n_0 једнак производу свих исписаних простих чинилаца и тренутне вредности броја n. Пошто је када се алгоритам заврши она једнака 1, исписана је проста факторизација броја n.

Глава 2

Сложеност израчунавања

Важно питање за практичну примену написаних програма је то колико ресурса програм захтева за своје извршавање. Најважнији ресурси су сигурно време потребно за извршавање програма и заузета меморија, мада се могу анализирати и други ресурси (на пример, код мобилних уређаја важан ресурс је утрошена енергија). Дакле, обично се разматрају:

- **временска** сложеност алгоритма;
- **просторна (меморијска)** сложеност алгоритма.

На пример, ако један програм израчунава потребан број за 10 секунди, а други за два и по минута, јасно је да је први програм практично примењивији. Међутим, ако први програм за своје извршавање захтева преко 10 гигабајта меморије, други око 1 гигабајт, а ми имамо рачунар са 4 гигабајта меморије, први програм нам је практично неупотребљив (иако ради много брже од другог). Ипак, с обзиром на то да савремени рачунарски системи имају прилично велику количину меморије, време је чешће ограничавајући фактор и у наставку ћемо се чешће бавити анализом временске ефикасности алгоритама.

При том, прилично је релативно колико брзо програм треба да ради да бисмо га сматрали ефикасним. На пример, ако програм успе да за пола сата реши неки нерешен математички проблем, који људи годинама нису могли да реше, он је свакако користан и можемо га сматрати веома ефикасним. Са друге стране, ако програм уграђен у аутомобил контролише кочнице приликом проклизавања, њему и неколико стотина милисекунди израчунавања може бити превише, јер ће за то време аутомобил неконтролисано слетети са пута.

Понашање програма (па и количина утрошених ресурса), наравно, зависи од његових улазних параметара. Јасно је, на пример, да ће програм брже израчунати просечну оцену двадесетак ученика једног одељења, него просечну оцену неколико десетина хиљада ученика који полажу Државну матуру. Такође се може претпоставити да понашање програма не зависи од конкретних оцена које су ученици добили, већ само од броја ученика. Зато сложеност алгоритма често изражавамо у функцији *величине (димензије) његових улазних параметара*, а не самих вредности параметара. Величина улазне вредности може бити број улазних елемената које треба обрадити, број битова потребних за записивање улаза који треба обрадити, сам улазни број који треба обрадити итд. Увек је потребно експлицитно навести у односу на коју величину улазне вредности се разматра сложеност.

Са друге стране, неки се алгоритми не извршавају исто за све улазе исте величине, па је потребно наћи начин за описивање ефикасности алгоритма на разним могућим улазима исте величине.

- **Анализа најгорег случаја** заснива процену сложености алгоритма на најгорем случају (на случају за који се алгоритам најдуже извршава – у анализи временске сложености, или на случају за који алгоритам користи највише меморије – у анализи просторне сложености). Та процена може да буде варљива, тј. превише песимистична. На пример, ако се програм у 99,9% случајева извршава испод секунде, док се само у 0,1% случајева извршава за око 10 секунди, анализом најгорег случаја закључили бисмо да ће се програм извршавати за око 10 секунди. Са друге стране, анализа најгорег случаја нам даје јаке гаранције да програм који је у најгорем случају довољно ефикасан у свим случајевима може да се изврши са расположивим ресурсима.
- У неким ситуацијама могуће је извршити **анализу просечног случаја** и израчунати просечно време извршавања алгоритма, али да би се то урадило, потребно је прецизно познавати простор допуштених

улазних вредности и вероватноћу да се свака допуштена улазна вредност појави на улазу програма. У случајевима када је битна гаранција ефикасности сваког појединачног извршавања програма процена просечног случаја може бити варљива, превише оптимистична, и може да се деси да у неким ситуацијама програм не може да се изврши са расположивим ресурсима. На пример, анализа просечног случаја би за претходни програм пријавила да се у просеку извршава испод једне секунде, међутим, за неке улазе он се може извршавати и преко десет секунди.

- Анализа најбољег случаја је, наравно, превише оптимистична и никада нема смисла.

Некада се анализа врши тако да се процени укупно време потребно да се изврши одређен број сродних операција. Тај облик анализе назива се **амортизована анализа** и у тим ситуацијама нам није битно време извршавања појединачних операција, већ само збирно време извршавања свих операција.

У наставку ће, ако није другачије речено другачије, бити подразумевана анализа најгорег случаја.

2.1 Мерење времена извршавања

Понашање за конкретне вредности улазних параметара се може експериментално одредити, тестирањем рада програма. На пример, размотримо наредна два алгоритма (дата у псеудокоду) који израчунавају збирове природних бројева од 1 до i, за свако i из интервала 1 до n.

```
zbir = 0
foreach i in [1, n]:
    zbir += i
    print(zbir)
```

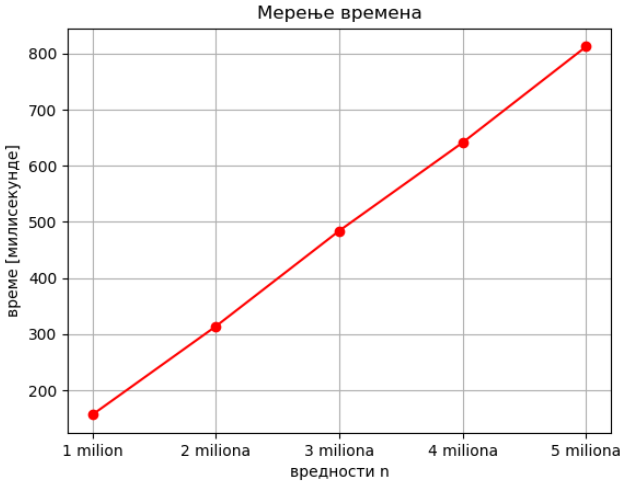
и

```
function zbir(k):
    zbir = 0
    foreach i in [1, k]:
        zbir += i
    return zbir

foreach k in [1, n]:
    print(zbir(k))
```

Ако се први алгоритам имплементира (на пример, у програмском језику Python) и ако се измери његово време извршавања за различите вредности n, добијају се следећи резултати (као резултат приказан је збир свих бројева од 1 до n).

```
n =  1000000, rezultat:   500000500000, vreme: 0.16 sekundi
n =  2000000, rezultat:  2000001000000, vreme: 0.31 sekundi
n =  3000000, rezultat:  4500001500000, vreme: 0.49 sekundi
n =  4000000, rezultat:  8000002000000, vreme: 0.65 sekundi
n =  5000000, rezultat: 12500002500000, vreme: 0.83 sekundi
```

Већ одавде се види да су времена приближно сразмерна вредностима n. Још прегледнији начин да уочимо ову пропорционалност је приказивање графика времена израчунавања суме у зависности од n, са кога се јасно види да код првог програма време извршавања приближно линеарно зависи од n.

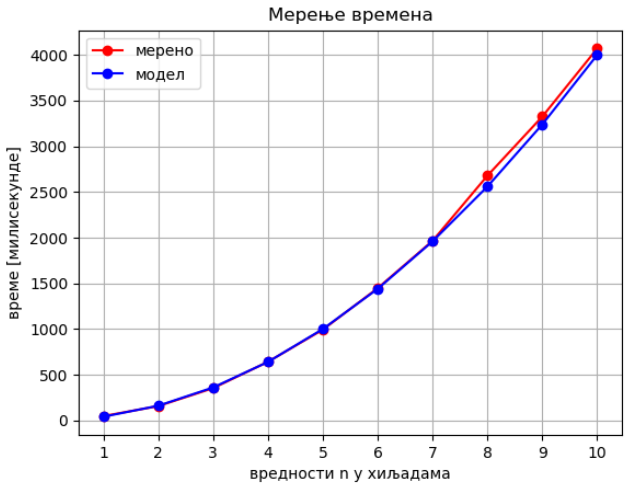


Слика 2.1: Линеарна зависност измереног времена

Ако се други алгоритам имплементира и ако се измери његово време извршавања (на пример, у програмском језику Python), добијају се следећи резултати (поново је приказан само збир свих бројева од 1 до n).

```
n =  1000, rezultat:   500500, vreme:    31.24 ms
n =  2000, rezultat:  2001000, vreme:   156.25 ms
n =  3000, rezultat:  4501500, vreme:   359.35 ms
n =  4000, rezultat:  8002000, vreme:   633.19 ms
n =  5000, rezultat: 12502500, vreme:   993.25 ms
n =  6000, rezultat: 18003000, vreme:  1448.38 ms
n =  7000, rezultat: 24503500, vreme:  1984.94 ms
n =  8000, rezultat: 32004000, vreme:  2547.53 ms
n =  9000, rezultat: 40504500, vreme:  3291.45 ms
n = 10000, rezultat: 50005000, vreme:  4049.11 ms
```

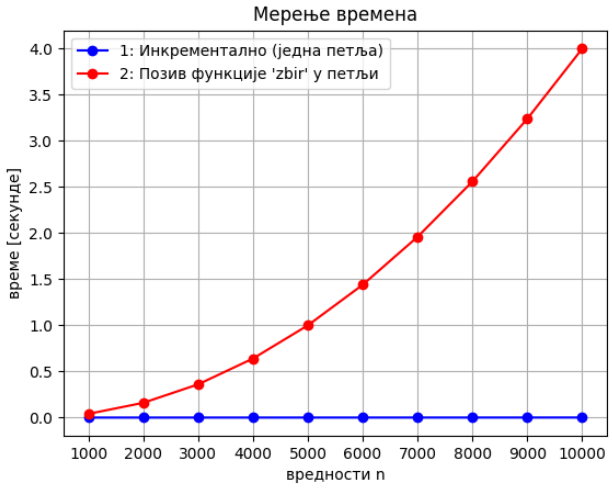
У овом случају зависност није линеарна. Наиме, када се n удвостручи, време се уместо 2 пута, отрпилике увећа 4 пута, што сугерише да је време извршавања сразмерно са квадратом величине улаза, тј. да је у питању квадратна зависност. Рачунањем $\frac{t(n)}{n^2}$ за разне n добијамо приближно сталну вредност 0.00004, што значи да се време извршавања $t(n)$ може апроксимирати као $t(n) \approx 0.00004n^2$. На следећој слици видимо график измерених вредности и график вредности добијене квадратне функције која моделира време извршавања.



Слика 2.2: Квадратна зависност измереног времена

Два дата графика се у значајној мери поклапају, што показује да је $0.00004n^2$ веома добра процена времена извршавања алгоритма за разне вредности n. Наравно, време извршвања увек зависи од програмског језика и конкретног рачунара на ком је мерење извршено, али експеримент говори да можемо очекивати да ће време рада другог алгоритма увек бити прилижно квадратна функција величине улаза. Ово се, наравно, може и формално доказати анализом броја извршених инструкција.

На основу измерених времена видимо да је први програм неупоредиво бржи у односу на други. Линеарно време извршавања је толико мање од квадратног, да када се оба времена прикажу на истом графику, делује да је линеарно време стално једнако нули.

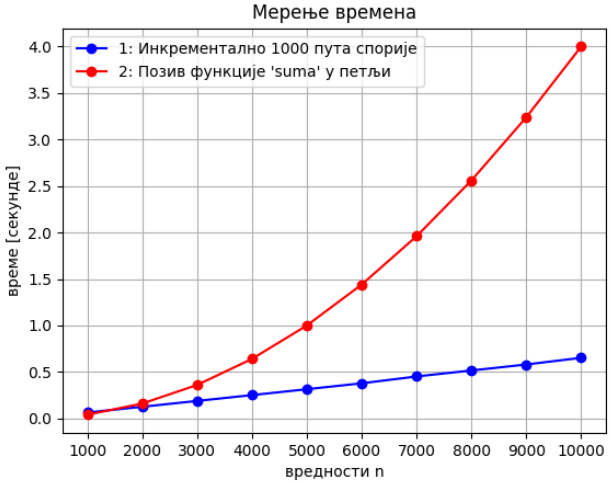


Слика 2.3: Однос линеарног и квадратног времена

2.2 Асимптотска анализа сложености

Мерење времена извршавања се може урадити за сваки програм, али оно није поуздан аргумент само за себе, већ служи пре свега за почетно стицање осећаја о ефикасности појединих алгоритама (да знамо шта треба да докажемо), као и за потврду закључака добијених теоријским разматрањем. Осим тога, често нам је потребно да можемо да унапред дамо неку грубу процену потребних ресурса за произвољне улазне вредности, без покретања програма (па чак и пре писања програма, само на основу алгоритма који ће бити примењен).

Питање које се природно поставља је то на ком ће се рачунару програм извршавати. Наравно, ако је један рачунар два пута бржи (у неком сегменту) од другог, за очекивати је да ће се програм на њему извршавати два пута брже. Ипак, показаће се да су разлике између ефикасних и неефикасних алгоритама толико велике, да је то што је неки рачунар 2, 3 или чак 10 пута бржи од другог заправо небитно и не може да надомести то колико је неефикасан алгоритам лошији од ефикасног. На пример, наредна слика приказује како би изгледао однос времена извршавања ако би се бржи алгоритам извршавао на 1000 пута споријем рачунару. Као што видимо, линеарна функција се мало "одлепила" од нуле, међутим, и даље су њене вредности много мање него код квадратне функције. Другим речима, алгоритам линеарне сложености, чак и када се успори 1000 пута, и даље је много бржи од алгоритма квадратне сложености (и што је n веће, разлика у брзини је све већа).



Слика 2.4: Однос линеарног и квадратног времена на рачунарима различите брзине

Да бисмо проценили зависност времена извршавања од димензије проблема, основни приступ је да покушамо да конструишемо функцију $f(n)$ која одређује зависност броја инструкција које алгоритам треба да изврши у односу на величину улаза n.

Израчунајмо број наредби сабирања које се изврше у првом и у другом програму из претходног примера (програми извршавају и друге наредбе, попут оних потребних да се организују петље, међутим, претпоставићемо да су нам сабирања једино значајна).

Јасно је да се у првом програму врши једно сабирање по петљи, па је укупан број сабирања једнак броју корака извршавања петље, а то је n. Ово је линеарна зависност, што је у складу са измереним временима.

У другом програму анализа је мало компликованија. За било које дато k, функција `zbir` врши око k сабирања. Пошто се функција позива за све вредности k од 1 до n, то се укупно изврши $1 + 2 + \ldots + n = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$ сабирања. И ако бисмо рачунали све инструкције, добили бисмо да је број инструкција нека квадратна функција облика $an^2 + bn + c$, што је опет у складу са измереним временима.

Ако (поједностављено) претпоставимо да се свака инструкција на рачунару извршава за једну наносекунду ($10^{-9}s$), а да број инструкција зависи од величине улаза n на основу функције $f(n)$, тада је време потребно да се алгоритам изврши дат у следећим табелама.

Алгоритми чија је сложеност одозго ограничена полиномијалним функцијама, у принципу се сматрају ефикасним.

$n/f(n)$	$\log n$	$\sqrt{n}$	n	$n \log n$	n^2	n^3
10	0,003 μs	0,003 μs	0,01 μs	0,033 μs	0,1 μs	1 μs
100	0,007 μs	0,010 μs	0,1 μs	0,644 μs	10 μs	1 ms
1,000	0,010 μs	0,032 μs	1,0 μs	9,966 μs	1 ms	1 s
10,000	0,013 μs	0,1 μs	10 μs	130 μs	0,1 s	16,7 min
100,000	0,017 μs	0,316 μs	100 μs	1,67 ms	10 s	11,57 dan
1,000,000	0,020 μs	1 μs	1 ms	19,93 ms	16,7 min	31,7 god
10,000,000	0,023 μs	3,16 μs	10 ms	0,23 s	1,16 dan	3×10^5 god
100,000,000	0,027 μs	10 μs	0,1 s	2,66 s	115,7 dan	
1,000,000,000	0,030 μs	31,62 μs	1 s	29,9 s	31,7 god	

Алгоритми чија је сложеност одоздо ограничена експоненцијалном или факторијелском функцијом се сматрају неефикасним.

$n/f(n)$	2^n	$n!$
10	1 μs	3,63 ms
20	1 ms	77,1 god
30	1 s	$8,4 \times 10^{15}$ god
40	18,3 min	
50	13 dan	
100	4×10^{13} god	

Можемо поставити и питање која димензија улаза се отприлике може обрадити за одређено време. Одговор је дат у наредној табели.

t	n	$n \log n$	n^2	n^3	2^n	$n!$
$1ms$	10^6	$63,000$	$1,000$	100	20	9
$10ms$	$10 \cdot 10^6$	$530,000$	$3,200$	215	23	10
$100ms$	$100 \cdot 10^6$	$4,5 \cdot 10^6$	$10,000$	465	27	11
$1s$	10^9	$40 \cdot 10^6$	$32,000$	$1,000$	30	12
$1min$	$60 \cdot 10^9$	$1,9 \cdot 10^9$	$245,000$	$3,900$	36	14

Из претходних табела јасно је да време извршавања суштински зависи од функције $f(n)$. На пример, ако поредимо алгоритме код којих је $f_1(n) = n$, $f_2(n) = 5n$, $f_3(n) = n^2$ и $f_4(n) = 2n^2 + 3n + 2$, јасно нам је да ће се за $n = 10^6$, време извршавања првог и другог алгоритма мерити милисекундама, док ће се време извршавања трећег и четвртог алгоритма мерити минутима. Код функције f_4, јасно је да је време које потиче од фактора $3n$ (три милисекунде) и 2 (две наносекунде) апсолутно занемариво у односу на време које долази од фактора $2n^2$ (око 33 минута). За прва два алгоритма рећи ћемо да имају *линеарну временску сложеност*, а за друга два да имају *квадратну временску сложеност*.

Чак ни педесет пута бржи рачунар неће помоћи да се трећи или четврти алгоритам изврше брже од првог или другог. Иако смо поједностављено претпоставили да се све инструкције извршавају исто време (једну наносекунду), што није случај у реалности, из претходних табела је јасно да нам тај поједностављени модел даје сасвим добру основу за поређење различитих алгоритама и да прецизнија анализа не би ни по чему значајно променила ситуацију. Сложеност се обично процењује на основу изворног кода програма. Савремени компилатори извршавају различите напредне оптимизације и машински кôд који се извршава може бити прилично другачији од изворног кода програма (на пример, компилатор може скупу операцију множења заменити ефикаснијим битовским операцијама, може наредбу која се више пута извршава у петљи изместити ван петље и слично). Детаљи који се у изворном коду не виде, попут питања да ли се неки податак налази у кеш-меморији или је потребно приступати РАМ-у, такође могу веома значајно да утичу на стварно време извршавања програма. Савремени процесори подржавају проточну обраду и паралелно извршавање инструкција, што такође чини стварно понашање програма другачијим од класичног, секвенцијалног модела који се најчешће подразумева приликом анализе алгоритама. Дакле, стварно време извршавања програма зависи од карактеристика конкретног рачунара на ком се програм извршава, али и од карактеристика програмског преводиоца, па и оперативног система на ком се програм извршава. Међутим, поново наглашавамо да ништа од тих фактора не може променити однос између времена извршавања алгоритама линеарне и алгоритама квадратне сложености, за велике улазе (код малих улаза, сви алгоритми раде веома ефикасно, па нам обрада малих улаза није интересантна).

Дакле, можемо закључити да нам за је за грубу процену времена потребног за извршавање неког алгоритма, чији број инструкција полиномијално зависи од величине улаза n довољно да знамо само који је степен тог полинома. Можемо слободно да занемаримо све мономе мањег степена, а можемо и слободно да занемаримо коефицијенте уз водећи степен, као и коефицијент којим се одређује брзина стварног рачунара у односу на овај фиктивни, за који смо приказали времена. Наиме у реалним ситуацијама сви ти коефицијенти могу да утичу да ће програм бити бржи или спорији највише десетак пута (па нек је и стотинак пута), али не могу да утичу на то да се за велики улаз алгоритам чији је број инструкција квадратни изврши брже од алгоритма чији је број инструкција линеаран (говоримо о односу минута и милисекунди).

Горња граница сложености се обично изражава коришћењем O-нотације.

Дефиниција: Ако постоје позитивна реална константа c и природан број n_0 такви да за функције f и g над

природним бројевима важи $f(n) \leq c \cdot g(n)$ за све природне бројеве n веће од n_0 онда пишемо $f(n) = O(g(n))$ и читамо "f је велико ‚о' од g".

У неким случајевима користимо и ознаку Θ која нам не даје само горњу границу, већ прецизно описује асимптотско понашање.

Дефиниција: Ако постоје позитивне реалне константе c_1 и c_2 и природан број n_0 такви да за функције f и g над природним бројевима важи $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ за све природне бројеве n веће од n_0, онда пишемо $f(n) = \Theta(g(n))$ и читамо „f је велико 'тета' од g".

Дакле, асимптотским ознакама смо занемарили мономе мањег степена и сакрили константе уз највећи степен полинома. Стварно време извршавања зависи и од константи сакривених у асимптотским ознакама, међутим, асимптотско понашање обично прилично добро одређује његов ред величине (да ли су у питању микросекунде, милисекунде, секунде, минути, сати, дани, године).

Наведимо карактеристике основних класа сложености.

- $O(1)$ – *константна сложеност*, алгоритми линијско-разгранате структуре који се извршавају практично моментално, нпр. алгоритми у којима се имплементира нека математичка формула;
- $O(\log n)$ – *логаритамска сложеност*, изузетно ефикасно, нпр. бинарна претрага;
- $O(\sqrt{n})$ – *коренска сложеност*, "логаритам за оне са јефтинијим улазницама" - немамо најбоља места, али ипак можемо да гледамо утакмицу, нпр. испитивање да ли је број прост, факторизација броја на просте чиниоце;
- $O(n)$ – линеарна сложеност, оптимално, када је за решење потребно погледати цео улаз, нпр. минимум/максимум серије елемената;
- $O(n \log n)$ – *квазилинеарна сложеност*, "линеарни алгоритам за оне са јефтинијим улазницама", ефикасни алгоритми засновани на декомпозицији (нпр. сортирање обједињавањем), ефикасном сортирању, коришћењу структура података са логаритамским временом приступа;
- $O(n^2)$ – *квадратна сложеност*, обично (али не обавезно) угнежђене петље, нпр. сортирање селекцијом, сортирање уметањем;
- $O(n^3)$ – *кубна сложеност*, обично (али не обавезно) вишеструко угнежђене петље, нпр. множење матрица;
- $O(2^n)$ – *експоненцијална сложеност*, изузетно неефикасно, нпр. испитивање свих подскупова;
- $O(n!)$ – *факторијелна сложеност*, изузетно неефикасно, нпр. испитивање свих пермутација.

Иако су класе поређане редом, не треба претпостављати да су оне "подједнако размакнуте". На пример, честа је заблуда да су алгоритми сложености $O(n \log n)$ по својој ефикасности негде између $O(n)$ и $O(n^2)$. Истина је заправо да су они прилично слични класи $O(n)$ и да је често тешко емпријски измерити разлику између те две класе, а да су и једни и други неупоредиво бржи од алгоритама квадратне сложености. На пример, ако је $n = 10^6$, веома груба процена (када се занемаре сви константни коефицијенти) броја корака линеарног алгоритма је 10^6, квазилинеарног алгоритма је око $20 \cdot 10^6$, а број корака квадратног алгоритма је 10^{12}. Дакле, на тако великом улазу квазилинеарни алгоритам би био тек пар десетина пута спорији од линеарног (конкретан однос, наравно, зависио би од констаних фактора), док би квадратни алгоритам био милион пута спорији од линеарног и неких педесет хиљада пута спорији од квазилинеарног.

2.3 Математичке основе

Да бисмо могли да анализирамо сложеност потребно је да владамо одређеним математичким апаратом. У наставку ћемо резимирати основне математичке појмове које ћемо користити у анализи сложености алгоритама.

2.3.1 Сумирање

Током анализе алгоритама често имамо потребу да израчунамо одређене коначне суме. Са њима сте се сретали у средњој школи и курсевима дискретне математике. Резимирајмо их кроз неколико најзначајнијих примера.

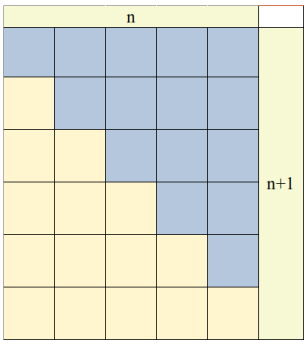
2.3.1.1 Аритметички низ

Гаусу се приписује да је још као дете израчунао да је

$$1 + 2 + \ldots + n = \sum_{k=0}^{n} k = \frac{n(n+1)}{2}.$$

Заиста, у овом збиру се крије $n/2$ парова чији је збир $n+1$ (ово, наравно, важи само када је n паран број, али нам даје одличну интуицију која нам помаже да ову формулу лако запамтимо). Прецизније, ако означимо тај збир са S онда је $2S = S + S = (1 + 2 + \ldots + n) + (n + (n-1) + \ldots + 1) = n \cdot (n+1)$.

Некада слика говори више од речи.



Слика 2.5: Гаусова формула

На основу претходног једноставно се изводи да је збир првих n чланова аритметичког низа чији је први члан a, а разлика између свака два суседна члана једнака r једнака

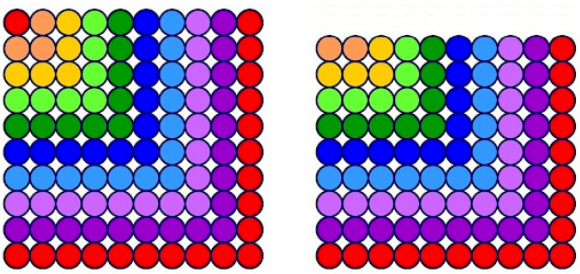
$$a + (a + r) + (a + 2r) + \ldots + (a + (n-1) \cdot r) = \sum_{k=0}^{n-1} (a + k \cdot r) = n \cdot a + r\frac{n(n-1)}{2}.$$

Интуиција нам опет говори да се овде крије $\frac{n}{2}$ парова чији је збир $a_0 + a_{n-1}$, што опет доводи до формуле $\frac{n}{2}(2a + (n-1) \cdot r)$.

Један важан аритметички низ је низ непарних бројева. Важи да је $1+3+5+\ldots+(2k-1) = \frac{k}{2} \cdot (1+(2k-1)) = k^2$.

Израчунавање збира узастопних парних бројева се лако своди на збир узастопних бројева. $2+4+6+\ldots+2k = 2(1 + 2 + 3 + \ldots + k) = k(k+1)$.

Поново слика говори више од речи.



Слика 2.6: Збир непарних бројева од 1 до $2k-1$ и збир парних бројева од 2 до $2k$

2.3.1.2 Геометријски низ и ред

Изведимо формулу за збир првих n чланова геометријског низа коме је први члан a а количник свака два узастопна члана $q \neq 1$. Обележимо тражену суму са S.

$$S = a + a \cdot q^1 + a \cdot q^2 + \ldots + a \cdot q^{n-2} + a \cdot q^{n-1} = \sum_{k=0}^{n-1} a \cdot q^k.$$

Ако леву и десну страну претходне једнакости помножимо са $1 - q$ добијамо једнакост:

$S \cdot (1 - q) = a \cdot (1 - q) + a \cdot q \cdot (1 - q) + a \cdot q^2 \cdot (1 - q) + \ldots + a \cdot q^{n-2} \cdot (1 - q) + a \cdot q^{n-1} \cdot (1 - q)$

Извршимо множења на десној страни једнакости:

$$S \cdot (1 - q) = a - a \cdot q + a \cdot q - a \cdot q^2 + a \cdot q^2 - a \cdot q^3 + \ldots + a \cdot q^{n-2} - a \cdot q^{n-1} + a \cdot q^{n-1} - a \cdot q^n$$

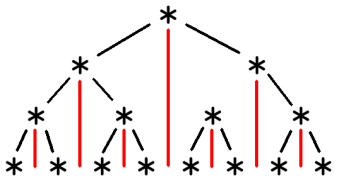
Сређивањем последњег израза добијамо $S \cdot (1 - q) = a - a \cdot q^n$. Према томе, пошто је $q \neq 1$, важи

$$S = a \cdot \frac{1 - q^n}{1 - q}.$$

За $|q| < 1$ геометријски ред конвергира и сума му је $\frac{a}{1-q}$.

Нама ће најчешће бити корисни случајеви $q = 2$ и $q = 1/2$.

На основу претходне формуле, за $a = 1$ и $q = 2$, важи да је $1 + 2 + \ldots + 2^{n-1} = 2^n - 1$. Ова формула има интересантно тумачење. Сума са леве стране представља укупан број чворова на првих n нивоа потпуног бинарног дрвета, док је израз са десне стране за један мањи од броја чворова на наредном нивоу $n+1$. Дакле, на сваком наредном нивоу бинарног дрвета има један чвор више него што је чворова на свим претходним нивоима дрвета.



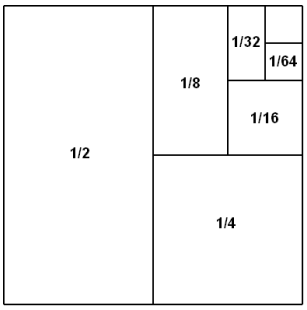
Слика 2.7: Број чворова на најнижем нивоу бинарног дрвета за један је већи од укупног броја чворова на претходним нивоима

За $a = 1$ и $q = 1/2$ добијамо да је

$$1 + 1/2 + \ldots + (1/2)^{n-1} = \frac{1 - (1/2)^n}{1 - 1/2} = 2 - (1/2)^{n-1}.$$

Са порастом n ова вредност се приближава вредности 2 (свакако је њоме ограничена одозго).

Опет слика говори више од речи.



Слика 2.8: Збир геометријског реда за $a = 1/2$, $q = 1/2$

2.3.1.3 Степене суме

Прикажимо како можемо израчунати суму квадрата свих природних бројева од 1 до n. Важи да је

$$(k+1)^3 - k^3 = k^3 + 3k^2 + 3k + 1 - k^3 = 3k^2 + 3k + 1.$$

Зато је

$$\begin{aligned} 2^3 - 1^3 &= 3 \cdot 1^2 + 3 \cdot 1 + 1 \\ 3^3 - 2^3 &= 3 \cdot 2^2 + 3 \cdot 2 + 1 \\ &\dots \\ (n+1)^3 - n^3 &= 3 \cdot n^2 + 3 \cdot n + 1 \end{aligned}$$

Сабирањем претходних једнакости добијамо

$$(n+1)^3 - 1 = 3 \cdot (1^2 + \ldots + n^2) + 3 \cdot (1 + \ldots + n) + (1 + \ldots + 1)$$

тј.

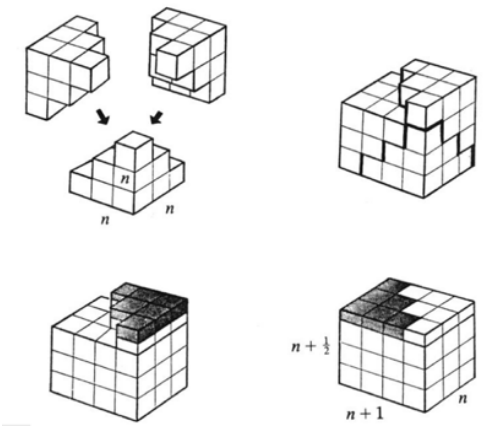
$$3 \cdot \sum_{k=1}^{n} k^2 = (n+1)^3 - 1 - 3\sum_{k=1}^{n} k - \sum_{k=1}^{n} 1.$$

На основу раније изведених формула за збир аритметичког низа, следи да је

$$\sum_{k=1}^{n} k^2 = \frac{1}{3} \cdot \left(n^3 + 3n^2 + 3n - 3\frac{n(n+1)}{2} - n \right) = \frac{n \cdot (2n+1) \cdot (n+1)}{6} = \frac{1}{3}\left(n \cdot (n + \frac{1}{2}) \cdot (n+1) \right).$$

Дакле, сума квадрата природних бројева од 1 до n се асимптотски понаша као $\frac{n^3}{3}$.

Опет слика говори више од речи.



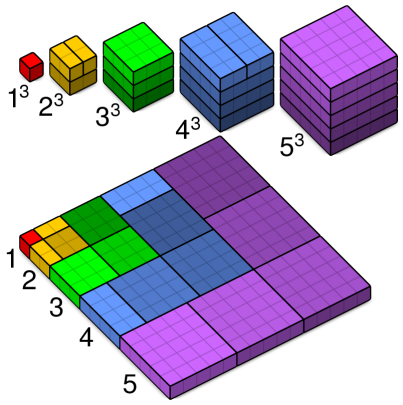
Слика 2.9: Збир квадрата свих природних бројева од 1 до n

Потпуно аналогно, сумирањем израза $(k+1)^4 - k^4$ од 1 до n и применом до сада изведених формула може се показати да је

$$1^3 + 2^3 + \ldots + n^3 = \sum_{k=1}^{n} k^3 = \frac{(n(n+1))^2}{4}.$$

Ово тврђење, познато као Никомахова теорема показује да је збир кубова свих природних бројева од 1 до n једнак квадрату збира свих природних бројева од 1 до n и асимптотски се понаша као $\frac{n^4}{4}$.

Опет слика говори више од речи.

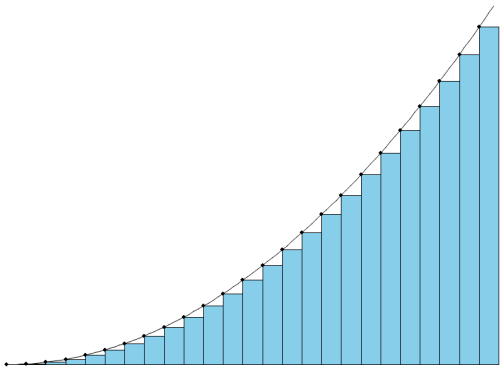


Слика 2.10: Збир кубова свих природних бројева од 1 до n

Пошто ће нас у анализи алгоритама најчешће занимати само асимптотско понашање функција, најважније је запамтити да се сума n p-тих степена свих природних бројева од 1 до n асимптотски понаша као $\frac{n^{p+1}}{p+1}$.

2.3.1.4 Примена диференцијалног и интегралног рачуна у израчунавању и оцени сума

За израчунавање и оцену сума могу се користити и изводи и интеграли. Приметимо да је неодређени интеграл функције x^p функција $\frac{x^{p+1}}{p+1}$, а да се збир p-тих степена свих природних бројева од 1 до n асимптотски понаша управо као $\frac{n^{p+1}}{p+1}$. То није случајност. Размотримо монотоно растућу функцију f (таква је функција $f(x) = x^n$) на домену $x \geq 0$. Сума $\sum_{k=0}^{n-1} f(k)$ се визуелно може представити као површина n правоугаоника (свакоме је ширина 1, а висина k-тог је $f(k)$). На слици је приказана сума првих 25 потпуних квадрата. Са слике је прилично очигледно да је та сума (збир површина правоугаоника) веома блиска површини испод криве $f(x) = x^2$ која се може израчунати (тј. чије се асимптотско понашање може проценити) применом одређених интеграла.



Слика 2.11: Процена суме одређеним интегралом

Илуструјмо ово и мало прецизније. Површина испод криве $f(x)$ за $k \leq x \leq k+1$ једнака је одређеном интегралу $\int_k^{k+1} f(x)\,dx$. Пошто је функција растућа, та површина је већа од површине правоугаоника чија је висина $f(k)$ и ширина 1, а мања од површине правоугаоника чија је висина $f(k+1)$ и ширина 1 тј. важи

$$f(k) \leq \int_k^{k+1} f(x)\,dx \leq f(k+1).$$

Зато је

$$\sum_{k=0}^{n-1} f(k) \leq \int_0^n f(x)\,dx \leq \sum_{k=0}^{n-1} f(k+1).$$

Горњу границу суме можемо директно прочитати из прве неједнакости. Пошто је

$$\sum_{k=0}^{n-1} f(k+1) = \sum_{k=0}^{n-1} f(k) - f(0) + f(n),$$

из друге неједнакости следи и доња граница, па важи:

$$\int_0^n f(x)\,dx + f(0) - f(n) \leq \sum_{k=0}^{n-1} f(k) \leq \int_0^n f(x)\,dx.$$

Случај када је $f(x)$ монотоно опадајућа функција за $x \geq 0$ се обрађује аналогно (само је потребно уместо $\leq$ користити $\geq$).

На пример, понашање суме $\sum_{k=0}^{n-1} ka^k = a+2a^2+3a^3+\ldots+(n-1)a^{n-1}$, можемо проценити израчунавањем одређеног интеграла $\int_0^n xa^x\,dx$.

Он се једноставно може израчунати парцијалном интеграцијом за $u = x$ (па је $du = dx$) и $dv = a^x\,dx$, одакле је $v = \frac{a^x}{\ln a}$. Зато је

$$\int_0^n xa^x\,dx = \int_0^n u\,dv = (uv)\big|_0^n - \int_0^n v\,du = \frac{na^n}{\ln a} - \frac{\int_0^n a^x\,dx}{\ln a} = \frac{na^n}{\ln a} - \frac{(a^n-1)}{\ln^2 a},$$

па се ова функција асимптотски понаша као na^n.

Ову суму је могуће израчунати и егзактно, применом диференцирања. Наиме, важи да је

$$\sum_{k=0}^{n-1} x^k = 1 + x + \ldots + x^{n-1} = \frac{x^n - 1}{x - 1}.$$

Диференцирањем обе стране по x добијамо

$$1 + 2x + 3x^2 + \ldots + (n-1)x^{n-2} = \frac{nx^{n-1}(x-1) - (x^n - 1)}{(x-1)^2}.$$

Множењем са x добијамо

$$\sum_{k=0}^{n-1} kx^k = x + 2x^2 + 3x^3 + \ldots + (n-1)x^{n-1} = x\frac{nx^{n-1}(x-1) - (x^n - 1)}{(x-1)^2}.$$

На пример, за $x = 2$ добијамо да је $\sum_{k=0}^{n-1} k2^k = (n-2)\cdot 2^n + 2$.

Напоменимо и да се та сума веома једноставно може израчунати и сасвим елементарним техникама. Нека је $S_x(n) = \sum_{k=0}^{n-1} kx^k$. Тада множењем ове једнакости са x, одузимањем добијеног резултата од полазне једнакости и применом формуле за збир геометријског низа добијамо

$$\begin{aligned}
S_x(n) &= x^1 + 2x^2 + 3x^3 + \ldots + (n-2)x^{n-2} + (n-1)x^{n-1} \\
x \cdot S_x(n) &= x^2 + 2x^3 + 3x^4 + \ldots + (n-2)x^{n-1} + (n-1)x^n \\
S_x(n) - xS_x(n) &= x^1 + x^2 + x^3 + \ldots + x^{n-1} + (n-1)x^n \\
(1-x)S_x(n) &= x(x^0 + \ldots + x^{n-2}) + (n-1)x^n = x\frac{1-x^{n-1}}{1-x} + (n-1)x^n
\end{aligned}$$

Одатле следи да је

$$S_x(n) = \frac{x - x^n}{(1-x)^2} + (n-1)x^n$$

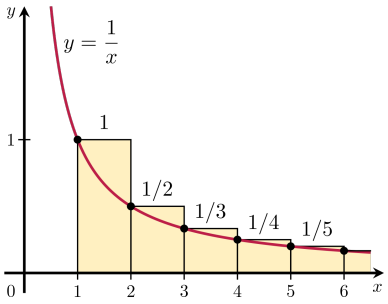
Иако није добијен идентичан израз ономе који је добијен применом диференцирања, лако се показује да су добијени изрази једнаки. Заменом вредности $x = 2$ поново добијамо да је $S_2(n) = \sum_{k=0}^{n-1} k2^k = (n-2) \cdot 2^n + 2$.

2.3.1.5 Хармонијски ред

Размотримо збир $H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n}$. Њега не можемо егзактно израчунати, али можемо га прилично добро приближно проценити.

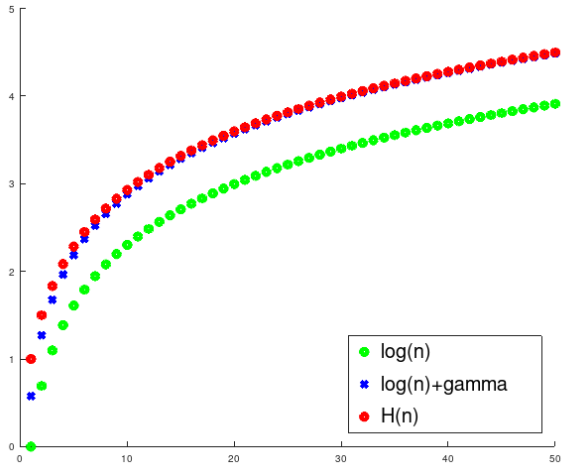
Прво покажимо да $H(n)$ дивергира и да тежи бесконачности како n тежи бесконачности. Важи да је $H(n) > 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \ldots$ Наиме, важи да је $\frac{1}{3} > \frac{1}{4}$, затим да је $\frac{1}{5} > \frac{1}{8}$, $\frac{1}{6} > \frac{1}{8}$ и $\frac{1}{7} > \frac{1}{8}$ итд. Међутим, важи да је $\frac{1}{4} + \frac{1}{4} = \frac{1}{2}$, да је $\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} = \frac{1}{2}$ итд. Зато је десни збир једнак $1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \ldots$, а овај збир јасно дивергира тј. тежи бесконачности са повећањем броја сабирака.

Слично би се могло доказати и коришћењем процене суме одређеним интегралом. Број $H(n)$ се може проценити као $\int_1^{n+1} \frac{1}{x} \mathrm{d}x$, који је једнак $(\ln x)\big|_1^{n+1}$ тј. $\ln(n+1)$.



Слика 2.12: Процена збира хармонијског реда $H(n)$ одређеним интегралом

Ако се нацрта график функције $H(n)$ може се уочити да она веома споро тежи бесконачности и може се приметити веома сличан облик графику функције $\ln n$. Заиста, доказује се да разлика између функција $H(n)$ и $\ln n$ веома брзо конвергира и тежи константи γ која је позната под именом Ојлер-Маскеронијева константа и чија је вредност приближно једнака $\gamma = 0.57722$. Ништа се не би променило и да се посматра однос $H(n)$ са функцијом $\ln(n+1)$ (јер са повећањем n разлика између $\ln(n+1)$ и $\ln n$ веома брзо тежи нули), осим што би процена била мало прецизнија за мале вредности n.



Слика 2.13: Збир хармонијског реда $H(n)$ и однос са логаритамском функцијом $\ln n$

2.4 Сложеност неких честих облика петљи

Прикажимо сада кроз неколико примера анализу сложености итеративно имплементираних алгоритама. Иако нећемо посматрати решења конкретних задатака, потрудићемо се да у примерима покријемо облике петљи који се јављају у великом броју конкретних алгоритама и решења конкретних задатака.

У примерима петљи који следе, претпоставља се да кôд у телу петљи који није приказан не утиче на бројачке променљиве и не мења границе петљи.

```
for (int i = 0; i < n; i++)
   // kod slozenosti O(1)
```

Сложеност претходне петље је $O(n)$.

```
for (int i = m; i < n; i++)
   // kod slozenosti O(1)
```

Сложеност претходне петље је $O(n - m)$.

```
for (int i = 0; i < n; i += 2)
   // kod slozenosti O(1)
```

Сложеност претходне петље је $O(n)$. Пошто се петља извршава за парне вредности бројачке променљиве, тело петље се извршава око $\frac{n}{2}$ пута и константни фактор је $\frac{1}{2}$, али је сложеност и даље линеарна.

```
for (int i = 0, j = n-1; i < j; i++, j--)
   // kod slozenosti O(1)
```

Сложеност претходне петље је $O(n)$. Показивачи се сусрећу приближно на средини опсега, а тело петље се извршава око $\frac{n}{2}$ пута.

```
for (int i = 0; i < m; i++)
   for (int j = 0; j < n; j++)
      // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(mn)$. Заиста, спољашња петља се извршава m, а у њеном телу се унутрашња петља извршава n пута, па се тело унутрашње петље изврши тачно mn пута.

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
      // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(n^2)$. Заиста, спољашња петља се извршава n, а у њеном телу се унутрашња петља извршава n пута, па се тело унутрашње петље изврши тачно n^2 пута.

```
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
      // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(n^2)$. Број извршавања тела унутрашње петље је $(n-1)+(n-2)+\ldots+2+1$, што је једнако $\frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$. Константни фактор је $\frac{1}{2}$, али је сложеност квадратна. До истог резултата можемо доћи ако схватимо да у сваком кораку унутрашње петље пар бројача одређује једну комбинацију бројева од 0 до $n-1$. Зато број извршавања тела одговара броју двочланих комбинација скупа од n елемената, што је једнако $\binom{n}{2} = \frac{n(n-1)}{2}$.

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
      for (int k = 0; k < n; j++)
         // kod slozenosti O(1)
```

Сложеност претходних петљи је прилично очигледно $O(n^3)$.

```
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
      for (int k = j+1; k < n; j++)
         // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(n^3)$. Најлакши начин да се ово закључи је да се примети да сваком извршавању тела одговара једна трочлана комбинација елемената скупа $0, \ldots, n-1$. Пошто трочланих комбинација има $\binom{n}{3} = \frac{n(n-1)(n-2)}{3 \cdot 2 \cdot 1}$, сложеност је кубна, а константни фактор је $\frac{1}{6}$.

```
for (int i = 0; i < m; i++)
    // kod slozenosti O(1)

for (int i = 0; i < n; i++)
    // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(m + n)$. Наиме, тело прве петља се изврши m пута, а затим тело друге петље n пута, па се тела обе петље укупно изврше $m + n$ пута.

```
for (int i = 0; i < n; i++)
    // kod slozenosti O(1)

for (int i = 0; i < n; i++)
    // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(n)$. Наиме, тело прве петље се изврши n пута, а затим тело друге n пута, па се тела обе петље укупно изврше $2n$ пута, но то је $O(n)$.

```
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
       // kod slozenosti O(1)

for (int i = 0; i < n; i++)
    // kod slozenosti O(1)
```

Сложеност претходних петљи је $O(n^2)$. Наиме, тело угнежђених петљи се изврши $\frac{n(n-1)}{2}$ пута, а затим тело друге петље n пута, што је заправо занемариво мало у односу на број извршавања тела угнежђених петљи. Дакле, први део кода апсолутно доминира временом извршавања и сложеност је $O(n^2)$.

Могло би се помислити да број угнежђених петљи одговара степену полинома, али то није увек случај.

```
for (int i = 1; i*i <= n; i++)
   // kod slozenosti O(1)
```

Иако садржи једну петљу, сложеност претходног кода није $O(n)$, већ $O(\sqrt{n})$. Наиме, петља се извршава све док је $i^2 \leq n$ тј. док је $i \leq \sqrt{n}$.

```
for (int i = 1; i < n; i *= 2)
   // kod slozenosti O(1)
```

Иако претходни код садржи петљу, његова сложеност је $O(\log n)$, јер се вредност променљиве `i` дуплира у сваком кораку, све док не престигне граничну вредност n.

```
for (int i = 0; i < 10; i++)
   // kod slozenosti O(1)
```

Сложеност претходног кода је $O(1)$. Иако је присутна петља, број њених извршавања је увек 10 и не зависи ни од једног параметра, па га можемо сматрати малом константом.

```
for (int i = 1; i <= n; i++)
    for (int j = 1; j < i; j *= 2)
          // kod slozenosti O(1)
```

Сложеност претходног кода је $O(n \log n)$. Сложеност унутрашње петље, за свако конкретно `i` је $O(\log i)$, па је укупна сложеност отприлике једнака $\log 1 + \log 2 + \ldots + \log n$, а за ово се може показати да је $O(n \log n)$ (јасно је да је израз мањи или једнак $n \log n$ јер је сваки сабирак мањи или једнак $\log n$, међутим, може се показати и да је збир већи или једнак од $\frac{n}{2} \log \frac{n}{2}$ (што је такође $\Theta(n \log n)$), занемаривањем првих $\frac{n}{2}$ сабирака, након чега остају само сабирци који су већи или једнаки $\log \frac{n}{2}$).

```
for (int i = n; i >= 1; i /= 2)
    for (int j = 1; j < i; j++)
          // kod slozenosti O(1)
```

Сложеност претходног кода је $O(n)$. Наиме, број извршавања унутрашње петље је $n + \frac{n}{2} + \frac{n}{4} + \ldots$, за шта се лако може показати да је одозго ограничено са $2n$.

```
for (int i = 0; i < n; i++) {
   for (int j = i+1; j < n && P(a[j]); j++)
          // kod slozenosti O(1)
   // kod slozenosti O(1)
   i = j;
}
```

Овакав код се може срести у алгоритмима у којима се анализирају све серије узастопних елемената низа који задовољавају неко својство `P` (на пример, коришћењем петљи овог облика можемо пронаћи најдужу серију узастопних парних елемената).

Иако постоје угнежђене петље, сложеност претходног кода је $O(n)$. Број извршавања унутрашње петље зависи од стања низа `a`, па не знамо унапред ни колико пута ће та петља бити покренута нити колико ће се пута њено тело извршити при сваком покретању. Међутим, да бисмо одредили укупну сложеност целог кода, то нам није ни потребно. Можемо извршити амортизовану анализу и израчунати укупан број извршавања тела унутрашње петље. Кључни детаљ је то што унутрашња петља креће од текуће вредности спољашње бројачке промењиве, док спољашња бројачка промењива након унутрашње петље наставља тамо где се унутрашња петља завршила. Зато при сваком пролазу кроз тело унутрашње петље промењива `j` има строго вредност већу него при сваком претходном пролазу, што се може десити највише n пута.

```
int j = 0;
for (int i = 0; i < n; i++) {
    while (j < n && P[j]) {
       // kod slozenosti O(1)
       j++;
    }
   // kod slozenosti O(1)
}
```

Иако постоје угнежђене петље, сложеност претходне петље је $O(n)$. Кључни детаљ је то што унутрашња петља нема иницијализацију промењиве `j` на нулу и бројач `j` у унутрашњој петљи се све време само увећава (исто као и бројач `i` у спољашњој петљи). Укупан број корака је стога ограничен са $2n$.

```
int l = 0, d = n-1;
while (l < d) {
  do l++; while (l < d && P(a[l]));
  do d--; while (l < d && Q(a[d]));
  if (l < d)
     // kod slozenosti O(1)
}
```

Сличан кôд се, на пример, може срести у алгоритму партиционисања низа. И претходни алгоритам је сложености $O(n)$ иако и он садржи угнежђене петље. То поново можемо утврдити амортизованом анализом (јер не знамо појединачни број извршавања унутрашњих петљи, али можемо лако проценити укупан корака који се у њима направи). Наиме, промењива `l` се само увећава кренувши од почетка, а `d` се само смањује кренувши од краја низа, док се не сусретну, што ће се десити у највише n корака.

Дакле, иако нам у већини случајева угнежђеност петљи описује сложеност алгоритма, треба бити обазрив и код анализирати пажљивије, да се сложеност не би преценила.

2.5 Скривена сложеност

Често процену сложености грубо вршимо тако што анализирамо структуру петљи у програму, занемарујући остале операције (обично за све сем петљи сматрамо да је $O(1)$). То може бити прилично варљиво, јер се у коду могу позивати функције (било кориснички дефинисане, било библиотечке) које нису константне сложености. Још горе, и неки оператори могу бити неконстантне сложености (обично линеарне).

```
string rez = "";
for (char c : s)
```

```
rez = c + rez;
```

Иако има само једну петљу, претходни фрагмент може бити сложености $O(n^2)$, где је n дужина ниске `s`. Наиме, додавање карактера на почетак ниске у језику C++ може бити линеарне сложености $O(n)$, где је n дужина ниске (јер захтева померање наредних карактера надесно).

2.6 Рекурентне једначине

Сложеност рекурзивних функција се често може описати рекурентним једначинама. Решење рекурентне једначине је функција $T(n)$ и за решење ћемо рећи да је у затвореном облику ако је изражено као елементарна функција по n (и не укључује са десне стране поновно реферисање на функцију T). Често ћемо се задовољити да уместо потпуно прецизног решења знамо само његово асимптотско понашање. Подсетимо се неколико најчешћих рекурентних једначина.

У првој групи се проблем своди на проблем димензије која је тачно за један мања од димензије полазног проблема.

- *Једначина*: $T(n) = T(n-1) + O(1), T(0) = O(1)$. *Пример*: Тражење минимума низа. *Решење*: $O(n)$.
- *Једначина*: $T(n) = T(n-1) + O(\log n), T(0) = O(1)$. *Пример*: Формирање балансираног бинарног дрвета. *Решење*: $O(n \log n)$.
- *Једначина*: $T(n) = T(n-1) + O(n), T(0) = O(1)$. *Пример*: Сортирање селекцијом. *Решење*: $O(n^2)$.

У другој групи се проблем своди на два (или више) проблема чија је димензија за један или два мања од димензије полазног проблема. То обично доводи до експоненцијалне сложености.

- *Једначина*: $T(n) = 2T(n-1) + O(1), T(0) = O(1)$. *Пример*: Ханојске куле. *Решење*: $O(2^n)$
- *Једначина*: $T(n) = T(n-1) + T(n-2) + O(1), T(0) = O(1)$. *Пример*: Фибоначијеви бројеви. *Решење*: $O(2^n)$

У наредној групи се проблем своди на један (или више) потпроблема који су значајно мање димензије од полазног (обично су бар дупло мањи). Ово доводи до полиномијалне сложености, па често и до веома ефикасних решења.

- *Једначина*: $T(n) = T(n/2) + O(1), T(0) = O(1)$. *Пример*: Бинарна претрага сортираног низа. *Решење*: $O(\log n)$.
- *Једначина*: $T(n) = T(n/2) + O(n), T(0) = O(1)$. *Пример*: Проналажење медијане (средишњег елемента) низа. *Решење*: $O(n)$.
- *Једначина*: $T(n) = 2T(n/2) + O(1), T(0) = O(1)$. *Пример*: Обилазак потпуног бинарног дрвета. *Решење*: $O(n)$.
- *Једначина*: $T(n) = 2T(n/2) + O(n), T(0) = O(1)$. *Пример*: Сортирање обједињавањем. *Решење*: $O(n \log n)$.

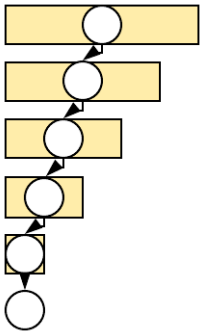
Ако су границе у самим једначинама егзактне, скоро у свим претходно набројаним једначинама дато решење није само горње ограничење, већ је асимптотски егзактно. На пример, решење једначине $T(n) = 2T(n/2) + \Theta(n), T(1) = \Theta(1)$ има решење $T(n) = \Theta(n \log n)$. Изузетак је пример Фибоначијевог низа где понашање јесте експоненцијално, али основа није 2, већ златни пресек $(1 + \sqrt{5})/2$.

Потпуно формално и прецизно извођење и доказивање асимптотског понашања решења ових једначина нам неће бити у директном фокусу. Много важније је стећи неку интуицију зашто су решења баш таква каква јесу (уз одређену дозу резерве, јер овакве грубе апроксимације некада могу довести до грешке). Један начин да се то уради је да се крене са "одмотавањем" рекурзије и да се види до чега се долази.

2.6.1 Једначина $T(n) = T(n-1) + O(1)$, $T(0) = O(1)$

На пример, код једначине $T(n) = T(n-1) + O(1)$ и $T(0) = O(1)$, након одмотавања добијамо да важи

$$\begin{aligned} T(n) &= T(n-1) + O(1) \\ &= T(n-2) + O(1) + O(1) \\ &= T(n-3) + O(1) + O(1) + O(1) \\ &= \dots \\ &= T(0) + n \cdot O(1) \\ &= O(1) + n \cdot O(1) \\ &= O(n). \end{aligned}$$



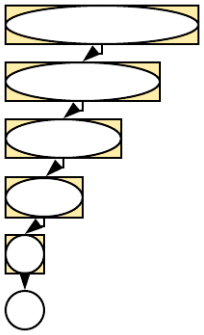
Слика 2.14: Дрво позива у случају $T(n) = T(n-1) + O(1)$, $T(0) = O(1)$ за $n = 4$. Правоугаоник означава димензију улаза, а елипса количину посла који се обавља у том чвору.

2.6.2 Једначина $T(n) = T(n-1) + O(n)$, $T(0) = O(1)$

Код једначине $T(n) = T(n-1) + O(n)$, $T(0) = O(1)$ слично добијамо n сабирака који су сви $O(n)$ тако да је укупна сума $O(n^2)$. Поставља се питање да ли је ова граница егзактна тј. да ли је могуће да је сложеност мања од изведеног горњег ограничења. Претпоставимо да је $T(n) = T(n-1) + cn$ и да је $T(0) = O(1)$. Тада је

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= T(n-2) + (n-1) + cn \\ &= \dots \\ &= T(0) + c(1 + \dots + n) \\ &= O(1) + cn(n+1)/2, \end{aligned}$$

тако да је $T(n) = \Theta(n^2)$.



Слика 2.15: Дрво позива у случају $T(n) = T(n-1) + O(n)$, $T(0) = O(1)$ за $n = 4$

2.6.3 Једначина $T(n) = T(n-1) + O(\log n)$, $T(0) = O(1)$

Покажимо још и шта се дешава са једначином $T(n) = T(n-1) + c\log n$, $T(0) = O(1)$. Одмотавањем добијамо

$$\begin{aligned} T(n) &= T(n-1) + c\log n \\ &= T(n-2) + c\log(n-1) + c\log(n) \\ &= \ldots \\ &= O(1) + c(\log 1 + \ldots + \log n). \end{aligned}$$

Пошто је логаритам растућа функција, сваки од n чланова овог збира ограничен је одозго вредношћу $\log n$. Зато је збир $O(n\log n)$. Докажимо да ово ограничење није превише грубо. Важи да је

$$\log 1 + \log 2 + \ldots + \log n \geq \log(n/2) + \log(n/2+1) + \ldots + \log n,$$

јер је првих $n/2$ чланова који су из суме изостављени сигурно ненегативно. Пошто је логаритам растућа функција (за основу већу од 1), сви сабирци у овом збиру су већи или једнаки $\log(n/2)$, па је

$$\log(n/2) + \log(n/2+1) + \ldots + \log n \geq \log(n/2) + \log(n/2) + \ldots + \log(n/2).$$

Збир на десној страни има $n/2$ истих сабирака и једнак је $(n/2)\cdot\log(n/2)$. Стога је почетни збир логаритама ограничен и одоздо и одозго функцијама које су $\Theta(n\log n)$, па је и сам $\Theta(n\log n)$.

Још један начин да се ово покаже који се често среће у литератури је следећи. Збир логаритама, једнак је логаритму производа, па заправо овде рачунамо вредност $\log 1 \cdot \ldots \cdot n = \log n!$. По Стирлинговој формули $n!$ се понаша као $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$. Зато се $\log n!$ понаша као $n\log n - n + O(\log n)$, па је укупан збир $\Theta(n\log n)$.

Можемо уочити неке правилности. Код свих једначина облика $T(n) = T(n-1) + f(n)$, $T(0) = c$, након одмотавања добијамо да је $T(n) = c + f(1) + f(2) + \ldots + f(n)$, тако да се одређивање асимптотског понашања своди на сумирање.

- Збир n сабирака реда $1 + 2 + \ldots + n$ има вредност $n(n+1)/2$, која је реда $\Theta(n^2)$. То је само дупло мање од вредности збира $n + \ldots + n$, који се састоји од n сабирака и има вредност n^2.
- Збир n сабирака $\log 1 + \ldots + \log n$ има вредност која се понаша као $n\log n - n$, што је асимптотски исто као вредност збира $\log n + \ldots + \log n$ који има n сабирака и вредност $n\log n$.
- Слично, збир $1^2 + \ldots + n^2$ има вредност $n(n+1)(2n+1)/6$, што је $\Theta(n^3)$ и само је око три пута мање од збира $n^2 + \ldots + n^2$ који има n сабирака и вредност n^3.

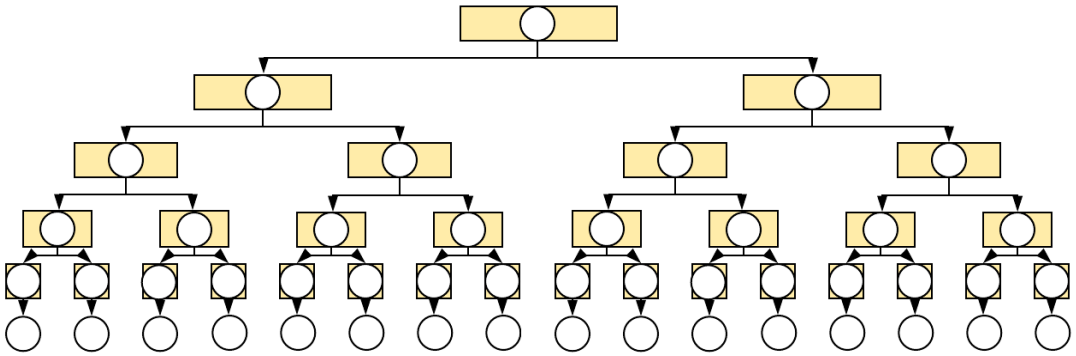
Иако овакве генерализације прете да буду непрецизне, са малом дозом резерве се може проценити да алгоритми у којима се n пута примењује нека операција сложености $\Theta(f(k))$ имају сложеност $\Theta(n \cdot f(n))$, чак и када се операција у сваком кораку примењује над подацима који су се за $O(1)$ повећали у односу на претходни корак и само у крајњој инстанци имамо $\Theta(n)$ података.

2.6.4 Једначина $T(n) = 2T(n-1) + O(1)$, $T(0) = O(1)$

Одмотавањем једначине $T(n) = 2T(n-1) + O(1)$, $T(0) = O(1)$ добијамо

$$\begin{aligned} T(n) &= 2T(n-1) + O(1) \\ &= 2(2T(n-2) + O(1)) + O(1) = 4T(n-2) + 2O(1) + O(1) \\ &= 4(2T(n-3) + O(1)) + 2O(1) + O(1) \\ &= 8T(n-3) + 4O(1) + 2O(1) + O(1) \\ &= \dots \\ &= 2^n T(0) + (2^{n-1} + \dots + 2 + 1) \cdot O(1) \\ &= 2^n \cdot O(1) + (2^n - 1) \cdot O(1) = O(2^n). \end{aligned}$$

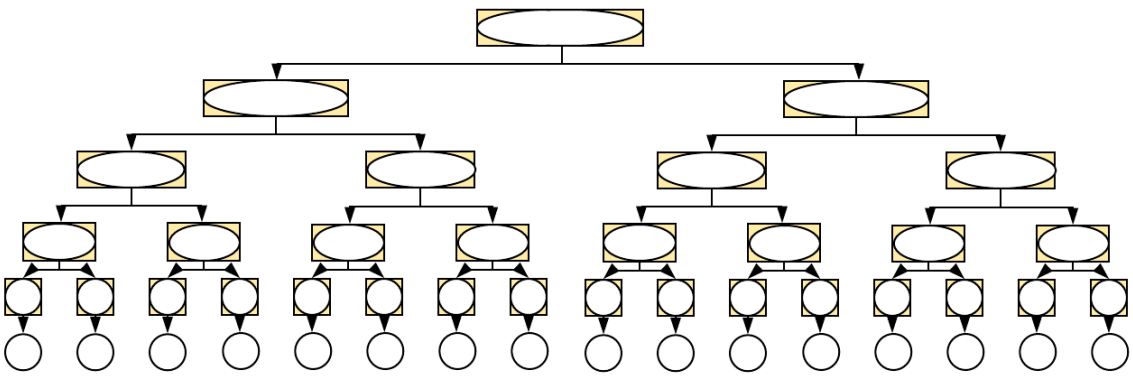
Дакле, иако се у сваком рекурзивном позиву ради мало посла, рекурзивних позива има експоненцијално много, што доводи до изразито неефикасног алгоритма.



Слика 2.16: Дрво позива у случају $T(n) = 2T(n-1) + O(1)$, $T(0) = O(1)$ за $n = 5$

2.6.5 Једначина $T(n) = 2T(n-1) + O(n)$, $T(0) = O(1)$

Функције које задовољавају једначину $T(n) = 2T(n-1) + O(n)$, $T(0) = O(1)$ такође показују експоненцијалну сложеност.



Слика 2.17: Дрво позива у случају $T(n) = 2T(n-1) + O(n)$, $T(0) = O(1)$ за $n = 5$

Одмотавањем једначине $T(n) = 2T(n-1) + c \cdot n$, $T(0) = O(1)$ добијамо

$$\begin{aligned} T(n) &= 2T(n-1) + c \cdot n \\ &= 2(2T(n-2) + c \cdot (n-1)) + c \cdot n = 4T(n-2) + c \cdot (2(n-1) + n) \\ &= 4(2T(n-3) + c \cdot (n-2)) + c \cdot (2(n-1) + n) \\ &= 8T(n-3) + c \cdot (4(n-2) + 2(n-1) + n) \\ &= \ldots \\ &= 2^n T(0) + c(2^{n-1} \cdot 1 + 2^{n-2} \cdot 2 + \ldots + 2(n-1) + n) \cdot O(1) \\ &= 2^n \cdot O(1) + \sum_{k=0}^{n} 2^k (n-k). \end{aligned}$$

Коришћењем раније изведених формула можемо једноставно израчунати и суму

$$\sum_{k=0}^{n} 2^k (n-k) = n + 2(n-1) + 2^2(n-2) + \ldots + 2^{n-1} \cdot 1.$$

Важи да је

$$\begin{aligned} \sum_{k=0}^{n} 2^k (n-k) &= n \sum_{k=0}^{n} 2^k - \sum_{k=0}^{n} k 2^k \\ &= n(2^{n+1} - 1) - ((n-1)2^{n+1} + 2) \\ &= 2^{n+1} - n - 2 \end{aligned}$$

Зато је $T(n) = 2^n \cdot O(1) + 2^{n+1} - n - 2$. Дакле, и у овом случају функција показује експоненцијално понашање $O(2^n)$.

2.6.6 Мастер теорема

Једначине засноване на декомпозицији проблема на неколико мањих потпроблема које су облика $T(n) = aT(n/b) + O(n^k)$, $T(0) = O(1)$ се решавају на основу **мастер теореме**.

Теорема: Решење рекурентне релације $T(n) = aT(n/b) + cn^k$, где су a и b целобројне константе такве да важи $a \geq 1$ и $b \geq 1$, и c и k су позитивне реалне константе је

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & \text{ако је } \log_b a > k \\ \Theta(n^k \log n), & \text{ако је } \log_b a = k \\ \Theta(n^k), & \text{ако је } \log_b a < k \end{cases}$$

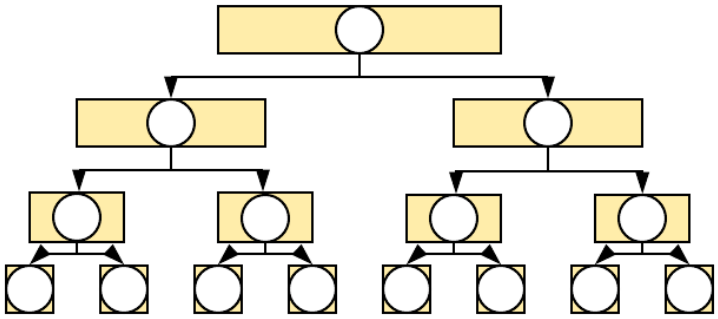
Нећемо давати доказ ове теореме, али покушајмо опет да дамо неко интуитивно јасно објашњење.

2.6.7 Једначина $T(n) = 2 \cdot T(n/2) + O(1)$, $T(1) = O(1)$

У првом случају се добија дрво рекурзивних позива чији број чворова доминира послом који се ради у сваком чвору. Размотримо, на пример, једначину $T(n) = 2 \cdot T(n/2) + O(1)$, $T(1) = O(1)$. Дрво ће садржати $O(n)$ чворова, а у сваком чвору ће се вршити посао који захтева $O(1)$ операција. Одмотавањем рекурентне једначине, добијамо

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + O(1) \\ &= 4 \cdot T(n/4) + 2 \cdot O(1) + O(1) \\ &= 8 \cdot T(n/8) + 4 \cdot O(1) + 2 \cdot O(1) + O(1) \\ &= 2^k \cdot T(n/2^k) + (2^{k-1} + \ldots + 2 + 1) \cdot O(1). \end{aligned}$$

Ако је $n = 2^k$ добијамо да је $n/2^k = 1$, па пошто је на основу формуле за збир геометријског низа $2^{k-1} + \ldots + 2 + 1 = 2^k - 1$, сложеност је $\Theta(n)$. И када n није степен двојке, добија се исто асимптотско понашање (што се може доказати ограничавањем одозго и одоздо степенима двојке).



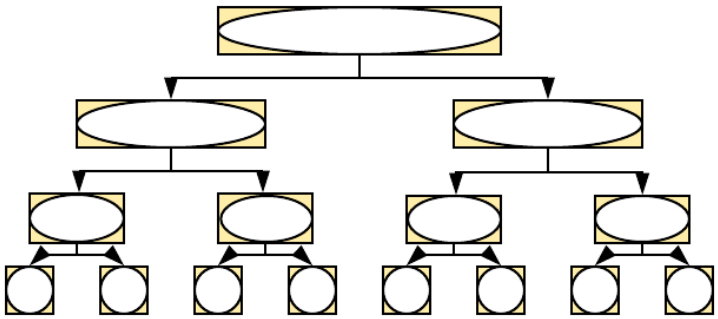
Слика 2.18: Дрво позива у случају $T(n) = 2T(n/2) + O(1), T(1) = O(1)$ за $n = 8$

2.6.8 Једначина $T(n) = 2 \cdot T(n/2) + c \cdot n$, $T(1) = O(1)$

У другом случају су број чворова и посао који се ради на неки начин уравнотежени. Размотримо, на пример, једначину $T(n) = 2 \cdot T(n/2) + c \cdot n, T(1) = O(1)$ и поново покушајмо да је одмотамо.

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + c \cdot n \\ &= 2 \cdot (2 \cdot T(n/4) + c \cdot n/2) + c \cdot n \\ &= 4T(n/4) + c \cdot n + c \cdot n \\ &= 4(2T(n/8) + c \cdot n/4) + 2 \cdot c \cdot n \\ &= 8T(n/8) + 3 \cdot c \cdot n \\ &= \ldots \\ &= 2^k \cdot T(n/2^k) + k \cdot c \cdot n. \end{aligned}$$

Ако је $n = 2^k$ после $k = \log_2 n$ корака $n/2^k$ ће достићи вредност 1 тако да ће збир бити реда величине $n \cdot O(1) + \log_2 n \cdot c \cdot n = \Theta(n \log n)$. Исто важи и када n није степен двојке.



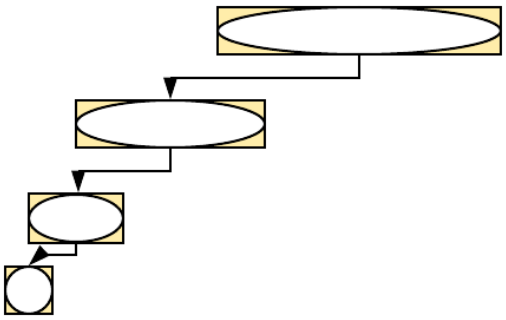
Слика 2.19: Дрво позива у случају $T(n) = 2T(n/2) + O(n), T(1) = O(1)$ за $n = 8$

2.6.9 Једначина $T(n) = T(n/2) + cn$, $T(1) = O(1)$

У трећем случају посао који се ради у чворовима доминира бројем чворова. Размотримо једначину $T(n) = T(n/2) + cn, T(1) = O(1)$. Њеним одмотавањем добијамо да је

$$\begin{aligned} T(n) &= T(n/2) + cn \\ &= T(n/4) + cn/2 + cn \\ &= T(n/8) + cn/4 + cn/2 + cn \\ &= \ldots \\ &= T(n/2^k) + cn(1/2^{k-1} + \ldots + 1/2 + 1). \end{aligned}$$

Поново, ако је $n = 2^k$, тада је први члан једнак $O(1)$ и пошто је на основу формуле за збир геометријског низа $1/2^{k-1} + \ldots + 1/2 + 1 = (1 - (1/2)^k)/(1 - (1/2)) = 2 - 2/n$ збир је једнак $O(1) + cn(2 - 2/n) = \Theta(n)$.



Слика 2.20: Дрво позива у случају $T(n) = T(n/2) + O(n)$, $T(1) = O(1)$ за $n = 8$

2.6.10 Остали типови једначина

Прокоментаришимо да се у неким проблемима добијају једначине које нису баш у сваком рекурзивном позиву идентичне овим наведеним. На пример, приликом анализе алгоритма QuickSort, ако је пивот тачно на средини низа, важи да је $T(n) = 2T(n/2) + O(n)$ и $T(1) = O(1)$. Када би се то стално догађало, решење би било $T(n) = O(n \log n)$, међутим, вероватноћа да се то догоди је страшно мала, јер у већини случајева пивот не дели низ на два дела потпуно исте димензије и зато треба бити обазрив. Ако би се десило да пивот стално завршавао на једном крају низа, једначина би била $T(n) = T(n-1) + O(n)$, $T(1) = O(1)$, што би довело до сложености $O(n^2)$, што и јесте сложеност најгорег случаја. Анализом коју ћемо приказати касније се може утврдити да је просечна сложеност $O(n \log n)$ тј. да иако пивот није стално на средини, да је у довољном процену случајева негде близу ње (рецимо између 25% и 75% дужине низа). Слична анализа важи и за проблем проналажења медијане.

Међутим, постоје и алгоритми код којих ствари стоје другачије. Приликом обиласка бинарног дрвета, балансираност нема утицаја. Наиме, ако је дрво потпуно, тада је једначина $T(n) = 2T(n/2) + O(1), T(1) = O(1)$, чије је решење $O(n)$. Међутим, чак и када је дрво издегенерисано у листу, једначина је $T(n) = T(n-1) + O(1), T(1) = O(1)$, чије је решење опет $O(n)$. Какав год да је однос броја чворова у левом и десном поддрвету решење ће бити $O(n)$. То се може описати једначином $T(n) = T(k) + T(n-k-1) + O(1), T(1) = O(1)$, за $0 \leq k \leq n-1$, чије ће решење бити $O(n)$, без обзира на то какво се k појављује у разним рекурзивним позивима.

2.7 Анализа просечне сложености

2.7.1 Анализа просечне сложености алгоритма QuickSort

Упечатљиво својство алгоритма QuickSort је ефикасност у пракси насупрот квадратној сложености најгорег случаја. Ово запажање сугерише да су најгори случајеви ретки и да је просечна сложеност овог алгоритма осетно повољнија.

Претпоставимо да је једнака вероватноћа да ће произвољни елемент бити изабран за пивот и да је једнака вероватноћа да пивот након партиционисања заврши на било којој позицији од 0 до $n-1$. Ако бројимо само

упоређивања (број замена је мањи или једнак броју упоређивања), сложеност партиционисања је $n-1$. Тада просечна сложеност задовољава наредну рекурентну једначину.

$$T(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1))$$

Први сабирак се креће од $T(0)$ до $T(n-1)$, а други од $T(n-1)$ до $T(0)$, тако да се свако $T(i)$ јавља тачно два пута. Зато за $n \geq 1$ важи

$$T(n) = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} T(i).$$

Ово је такозвана *једначина са потпуном историјом*, јер се вредност $T(n)$ израчунава преко свих претходних вредности $T(i)$. Један начин да се историја елиминише је да се посматрају разлике између суседних чланова низа чиме се добија једначина која описује везу између два суседна члана. У овом случају се сваки од сабирака $T(i)$ дели са n, те пре одузимања сваки од узастопних чланова треба помножити са одговарајућим фактором. Тако се добија

$$\begin{aligned} nT(n) - (n-1)T(n-1) = &\left(n(n-1) + 2\sum_{i=0}^{n-1} T(i)\right) - \\ &\left((n-1)(n-2) + 2\sum_{i=0}^{n-2} T(i)\right) \\ = &\, 2(n-1) + 2T(n-1) \end{aligned}$$

Зато је

$$T(n) = \frac{2(n-1)}{n} + \frac{n+1}{n}T(n-1)$$

Иако је ова једначина линеарна рекурентна једначина која повезује само два узастопна члана низа, она није са константним коефицијентима и потребно је мало инвентивности да бисмо је решили. Дељење са $n+1$ нам даје погоднији облик.

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

Наиме, сада се види да су два члана која укључују непознату функцију $T(n)$ истог облика, па једначину можемо једноставно одмотати и коришћењем чињенице да је $T(0) = 0$ добити

$$\frac{T(n)}{n+1} = \frac{2(n-1)}{n(n+1)} + \frac{2(n-2)}{(n-1)n} + \ldots + \frac{2\cdot(1-1)}{1(1+1)} = 2\sum_{i=1}^{n}\frac{i-1}{i(i+1)}$$

Централно питање постаје како израчунати збир

$$\sum_{i=1}^{n}\frac{i-1}{i(i+1)}$$

Томе помаже раздвајање сабирака на парцијалне разломке. Из једначине

$$\frac{i-1}{i(i+1)} = \frac{A}{i} + \frac{B}{i+1}$$

следи да је $Ai + A + Bi = i - 1$, па је $A = -1$, $B = 2$ и важи

$$\frac{i-1}{i(i+1)} = \frac{2}{i+1} - \frac{1}{i}$$

Зато је

$$\sum_{i=1}^{n} \frac{i-1}{i(i+1)} = \frac{2}{1+1} - \frac{1}{1} + \frac{2}{2+1} - \frac{1}{2} + \frac{2}{3+1} - \frac{1}{3} + \ldots + \frac{2}{n+1} - \frac{1}{n}$$

Можемо груписати разломке са истим имениоцем и добити

$$\sum_{i=1}^{n} \frac{i-1}{i(i+1)} = \frac{1}{2} + \ldots + \frac{1}{n} - 1 + \frac{2}{n+1}$$

Зато је

$$T(n) = 2(n+1) \cdot \left(1 + \frac{1}{2} + \ldots + \frac{1}{n}\right) - 4n.$$

Знамо да се хармонијски збир $1 + 1/2 + \ldots + 1/n$ асимптотски понаша као $\log n + \gamma$, где је γ Ојлер-Маскеронијева константа $\gamma \approx 0,57722$ и зато је

$$T(n) = \Theta\left(2(n+1)(\log n + \gamma) - 4n\right) = \Theta(n \log n).$$

2.8 Амортизована анализа сложености

У неким ситуацијама се извесне операције понављају пуно пута током извршавања програма. У многим ситуацијама можемо да допустимо да појединачно извршавање неке операције траје и мало дуже, ако смо сигурни да више извршавања те операције у збиру неће трајати предуго (ако постоји довољно извршавања те операције која ће трајати кратко). Анализа укупне дужине трајања већег броја операција назива се *амортизована анализа сложености*. Амортизована цена извршавања n операција подразумева количник њихове укупне дужине извршавања и броја n. Илуструјмо је на једном примеру.

2.8.1 Динамички низ

Једна од најчешће употребљаваних структура података је динамички низ. Размотримо колико је потребно времена да се у њега смести елемената. Када у низу нема довољно простора да се смести наредни елемент, низ се динамички реалоцира. У најгорем случају ово подразумева копирање старог садржаја низа на нову локацију, што је операција сложености $O(m)$, где је m број елемената уписаних у низ. Поставља се питање како приликом реалокација одређивати број елемената проширеног низа.

2.8.1.1 Аритметичка стратегија

Једна стратегија може бити аритметичка и она подразумева да се кренувши од празног низа приликом сваке реалокације величина низа повећа за неки број k (ништа се суштински не би променило у анализи и да је иницијални капацитет уместо 0 неки број m_0). Избројмо колико је пута потребно извршити упис елемента у низ (та операција је обично најспорија), при чему ту рачунамо уписе нових елемената и уписе настале током померања постојећих елемената током реалокације. У првом кораку примене аритметичке стратегије алоцирамо k елемената и затим у k наредних корака вршимо упис по једног елемента. Онда вршимо реалокацију на величину $2k$ и притом преписујемо првих k елемената низа. Након тога уписујемо наредних k елемената, а онда приликом реалокације преписујемо $2k$ елемената. Сличан поступак се наставља све док се не упише елемент n. Дакле, број операција је онда једнак $k + k + k + 2k + k + 3k + \ldots$. Да би се сместило n елемената, реалокацију је потребно вршити око n/k пута, па ће укупан број операција бити отприлике једнак

$$\frac{n}{k}k + k \cdot (1 + 2 + \ldots + \frac{n}{k}) = n + k\frac{\frac{n}{k}(\frac{n}{k}+1)}{2} = \frac{n^2}{2k} + \frac{3n}{2}$$

Дакле, укупан број уписа асимптотски је једнак $\frac{n^2}{2k}$ тј. $O(n^2)$ и стога је амортизована цена једне операције асимптотски једнака $\frac{n}{2k}$, што је $O(n)$, додуше, за доста малу вредност константе уз n (што је веће k, то је реалокација мање, па је цена операције мања, али се цена плаћа кроз веће заузеће меморије и мању попуњеност алоцираног простора).

2.8.1.2 Геометријска стратегија

Друга стратегија може бити геометријска и она подразумева да се сваки пут величина низа повећа q пута за неки фактор $q > 1$. Претпоставимо да је почетна величина низа m_0. Дакле, након почетне алокације можемо да упишемо m_0 елемената. Након тога се врши прва реалокација у којој се величина низа повећава на qm_0 елемената и при чему се преписује m_0 елемената. Након тога се врши упис наредних $qm_0 - m_0$ елемената. У наредној реалокацији величина низа се повећава на q^2m_0 и притом се преписује qm_0 елемената. Након тога се уписује преосталих $q^2m_0 - qm_0$ елеманата. Поступак се даље наставља по истом принципу. Дакле, укупан број уписа у низ једнак је

$$m_0 + m_0 + (qm_0 - m_0) + qm_0 + (q^2m_0 - qm_0) + \ldots = m_0 + qm_0 + q^2m_0 + \ldots$$

После r реалокација укупан број уписа једнак је

$$m_0(1 + \ldots + q^r) = m_0\frac{q^{r+1} - 1}{q - 1}.$$

Ако претпоставимо да је цео низ попуњен после r реалокација, тј. да је $n = m_0q^r$, онда је укупан број операција потребних за попуњавање низа једнак

$$m_0\frac{q^{\frac{n}{m_0}} - 1}{q - 1} = \frac{qn - m_0}{q - 1}.$$

Асимптотски је, дакле, укупна цена извођења свих операција једнака $O(n)$, а амортизована цена извођења једне операције додавања у овакав низ је $O(1)$. Константан фактор једнак је $\frac{q}{q-1}$ и он је све мањи како q расте. Заиста, са повећањем q врши се све мање реалокација, али се цена плаћа већим ангажовањем меморије тј. мањом попуњеношћу низа.

Амортизована анализа сложености нам показује да са становишта времена извршавања геометријска стратегија реалокације даје значајно боље резултате него аритметичка.

2.9 Савети за побољшање сложености

Кључни савет за побољшање сложености је то да рачунар ради само оно што је неопходно да би се добио коначан резултат. Када се та идеја мало детаљније разради, добијамо следећи низ савета који нас често доводе до алгоритама мање сложености:

- Немој терати рачунар да врши дуготрајна израчунавања која се могу извршити и "пешке", применом математике.
- Немој терати рачунар да више пута израчунава једно те исто – упамти потребне резултате израчунавања у меморији, да их не би рачунао више пута.
- Немој терати рачунар да израчунава ствари које нису потребне за добијање коначног решења проблема.
- Немој терати рачунар да испитује случајеве за које унапред можеш закључити да не могу бити тражено решење проблема.
- Ако је то могуће, припреми податке тако да се касније могу ефикасније обрадити.
- Користи ефикасније структуре података.
- …

У наставку овог поглавља приказаћемо низ задатака које ћемо решити различитим алгоритмима, анализираћемо њихову асимптотску сложеност најгорег случаја и приказаћемо како се на бази приказаних савета могу изградити значајно ефикаснији алгоритми.

2.10 Замена итерације формулом

Један од важних савета за побољшање сложености алгоритама је тај да не терамо рачунар да врши дуготрајна израчунавања која се могу извршити и "пешке", применом математике. Наиме, често се одређене вредности израчунавају применом итеративних поступака. На пример, збир елемената неког низа израчунавамо додавањем једног по једног елемента. То даје тражени резултат, али и одузима неко време. Постоје ситуације када су елементи који се обрађују правилни и када се коначан резултат може добити применом неке задате формуле, без примене итеративног поступка. На пример, ако знамо да треба сабрати првих n елемената низа $1, 2, 3, \ldots, n$, нема потребе да примењујемо итеративни поступак сабирања чија је сложеност $O(n)$, већ је довољно да у времену $O(1)$ применимо Гаусову формулу на основу које знамо да је

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

Још неке формуле се често могу употребити за смањење сложености (али и за саму анализу сложености), па ћемо их у наставку навести.

2.10.1 Аритметички и геометријски низ

Сличне формуле које су нам корисне су формуле за n-ти члан и збир првих n елемената аритметичког низа $a_0, a_0 + d, a_0 + 2d, \ldots$.

$$a_i = a_0 + (i-1)d, \qquad \sum_{i=0}^{n} a_i = \frac{n(a_0 + a_n)}{2},$$

као и за n-ти члан и збир првих n елемената геометријског низа $a_0, {}_0 \cdot q, a_0 \cdot q^2, \ldots$

$$a_i = a_0 \cdot q^i, \qquad \sum_{i=0}^{n} a_i = a_0 \frac{1 - q^{n+1}}{1 - q}.$$

2.10.2 Збирови степена

$$1^2 + 2^2 + \ldots + n^2 = \sum_{k=1}^{n} k^2 = \frac{n \cdot (n + \frac{1}{2}) \cdot (n+1)}{3}$$

$$1^3 + 2^3 + \ldots + n^3 = \sum_{k=1}^{n} k^3 = \left(\frac{n(n+1)}{2} \right)^2.$$

2.10.3 Комбинаторика

- Број начина да се из скупа од n различитих бројева извуку два броја $a < b$ је $\binom{n}{2} = \frac{n(n-1)}{2}$.
- Број начина да се из скупа од n различитих бројева извуку три броја $a < b < c$ је $\binom{n}{3} = \frac{n(n-1)(n-2)}{3 \cdot 2}$.

Задатак: Број подстрингова који почињу и завршавају са 1

Дат је бинарни стринг (ниска карактера која се састоји од карактера 0 и 1). Написати програм којим се одређује број сегмената (подстринг узастопних елемената), дужине најмање 2, који почињу и завршавају са 1.

Улаз: Прва и једина линија стандардног улаза садржи бинарни стринг (састављен од 0 и 1).

Излаз: На стандардном излазу приказати у једној линији тражени број сегмената.

Пример

Улаз	*Излаз*
`010001001`	`3`

Објашњење

То су подстрингови `10001`, `10001001` и `1001`.

Решење

Анализа свих сегмената

Број свих сегмената који почињу и завршавају са 1 можемо једноставно одредити анализирајући све сегменте. У спољашњој петљи анализирамо један по један карактер. Сваку јединицу на коју наиђемо (за свако i такво да је s_i једнако 1), разматрамо као почетак сегмента и у унутрашњој петљи (бројачем j од $i+1$ до краја стринга) тражимо јединицу којом се сегмент завршава. За сваку јединицу пронађену у унутрашњој петљи (за свако j такво да је s_j једнако 1) увећавамо број сегмената.

```
int broj1x1Podstringova(const string& s) {
  int n = s.length();
  int br = 0;
  for (int i = 0; i < n - 1; i++)
    if (s[i] == '1')
      for (int j = i + 1; j < n; j++)
        if (s[j] == '1')
          br++;
  return br;
}
```

Анализа сложености. Приметимо да на овај начин исте карактере стринга непотребно анализирамо велики број пута. Сложеност алгоритма одговара укупном броју свих сегмената и једнака је $O(n^2)$.

Бројање јединица

Сваки сегмент који почиње и који се завршава јединицом дефинисан је позицијама две јединице у стрингу, па је укупан број тражених сегмената једнак броју начина да се изаберу две различите јединице у стрингу. Ако је укупан број јединица у стрингу b онда две јединице можемо изабрати на $\frac{b \cdot (b-1)}{2}$ начина.

```
int broj1x1Podstringova(const string& s) {
  int brojJedinica = 0;
  for (char c : s)
    if (c == '1')
       brojJedinica++;
  return brojJedinica * (brojJedinica - 1) / 2;
}
```

Анализа сложености. Пошто јединице пребројавамо само једним проласком кроз стринг, сложеност овог алгоритма је $O(n)$.

Задатак: Недостајући број

У низу бројева од 0 до n тачно један број је изостављен. Напиши програм који, без памћења елемената низа, учитава бројеве са улаза и ефикасно одређује који број недостаје.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^9$), а затим и описани низ бројева (бројеви су наведени у једном реду, раздвојени са по једним размаком).

Излаз: На стандардни излаз исписати елемент који недостаје.

Пример

Улаз	*Излаз*
`5` `0 4 2 5 1`	`3`

Решење

Линеарна претрага

Једно решење може бити засновано на линеарној претрази свих кандидата. Оно подразумева да су сви елементи учитани у низ (што, додуше, нарушава услов који је дат у поставци задатка). За све бројеве од 0 до n проверавамо да ли су садржани у низу.

У језику C++ линеарну претрагу можемо реализовати библиотечком функцијом `find` којој се прослеђују итератори на део низа који се претражује и вредност која се тражи. Функција ће вратити итератор на пронађено прво појављивање елемента или итератор који указује непосредно иза краја претраженог распона, ако елемент не постоји.

```
int n;
cin >> n;
vector<int> a(n);
for (int i = 0; i < n; i++)
  cin >> a[i];

for (int x = 0; x <= n; x++)
  if (find(begin(a), end(a), x) == end(a)) {
    cout << x << endl;
    break;
  }
```

Анализа сложености. Пошто су елементи учитани у низ дужине n, меморијска сложеност је $O(n)$.

Линеарна претрага низа од n елемената у најгорем случају захтева $O(n)$ корака, па пошто се тражи $n + 1$ елемент, сложеност је $O(n^2)$. Може се приметити и да није могуће да се у сваком кораку линеарне претраге догоди најгори случај. Најгори случај се дешава када се елемент не налази у низу, а сви елементи који се траже (сем једног) су присутни. Заиста, када тражимо елемент који се налази на позицији i та претрага ће се завршити у i корака. Прецизније, елемент на позицији 1 ће се пронаћи у једном кораку, елемент на позицији 2 у два корака, елемент на позицији 3 у три корака и тако даље. Укупан број корака је зато $1 + 2 + \ldots + n$, што је опет $O(n^2)$.

Нагласимо да не треба да нас завара случај када се линеарна претрага имплементира коришћењем библиотечке функције. Иако тада у главном делу програма постоји само једна петља, у њој се позива функција линеарне сложености, па је укупна сложеност квадратна (кажемо да се појављује скривена сложеност).

Разним техникама временска сложеност се може свести на $O(n)$, а циљ нам је и да меморијску сложеност сведемо на $O(1)$.

Збир елемената

Збир свих елемената из скупа $\{0, 1, 2, \ldots, n\}$ је $0 + 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$. То је збир елемената који се налазе у низу и недостајућег елемента. Недостајући елемент је, дакле, једнак, разлици између $\frac{n(n+1)}{2}$ и збира свих елемената низа. Потребно је обратити пажњу на то да је за израчунавање збира елемената потребно користити бар 64-битни целобројни тип.

```
int n;
cin >> n;
long long zbir = 0;
for (int i = 0; i < n; i++) {
  int x; cin >> x;
  zbir += x;
}
long long zbir_svih = ((long long)n) * (n+1) / 2;
int nedostajuci = zbir_svih - zbir;
cout << nedostajuci << endl;
```

Анализа сложености. Збир свих елемената лако можемо израчунати у времену $O(n)$, приликом учитавања (без смештања елемената у низ, па је меморијска сложеност $O(1)$).

Напомена. Ово решење би се могло уопштити и на случај када знамо да недостају два броја од 0 до n. Тада бисмо уз збир израчунали и збир квадрата, а затим решавањем система две једначине са две непознате могли

израчунати бројеве који недостају.

Задатак: Максимални принос

Фармер поседује њиву димензије $a \times b$ метара. Да би лакше парцелисао њиву, бројеви a и b су цели. На основу субвенције добио је могућности да продужи странице своје њиве укупно за c метара (али тако да њива остане целобројних димензија). Он жели да то уради тако да након продужења површина буде што већа, тако да може да оствари што већи укупан принос. Напиши програм који одређује највећу могућу површину њиве након продужења страница.

Улаз: Са стандардног улаза се учитавају природни бројеви $a, b, c \leq 10^9$, раздвојени са по једним размаком.

Излаз: На стандардни излаз исписати максималну површину након продужења страница.

Пример 1

Улаз	*Излаз*
`5 10 3`	`80`

Објашњење

Димензија након проширења ће бити 8×10.

Пример 2

Улаз

```
9 10 4
```

Излаз

```
132
```

Објашњење

Димензија након проширења ће бити 11×12.

Пример 3

Улаз

```
14 17 5
```

Излаз

```
324
```

Објашњење

Димензија након проширења ће бити 18×18.

Решење

Груба сила

Задатак може бити решен грубом силом, тј. испробавањем свих могућности расподеле додатне дужине. За сваку дужину i између 0 и c, додајемо i страници a и $c - i$ страници b, израчунавамо површину и тражимо максимум тако добијених површина.

```
long long maksimalniPrinos(long long a, long long b, long long c) {
  long long maks = a * (b + c);
  for (long long i = 1; i <= c; i++)
    maks = max(maks, (a+i)*(b+c-i));
  return maks;
}
```

Анализа сложености. Сложеност овог решења је $O(c)$.

Израчунавање максималног приноса

Од свих правоугаоника датог фиксираног обима, највећу површину има квадрат. Заиста, ако је познат обим правоугаоника $O = 2(a+b)$, тада је познат и полуобим $a+b = s$. Површина $a \cdot b = a \cdot (s-a) = a \cdot s - a \cdot a = s^2/4 - (a - s/2)^2$. Пошто је $(a - s/2)^2 \geq 0$, површина не може бити већа од $s^2/4$, а једнака је тој вредности када је $a = b = s/2$. Зато увећање треба направити тако да се добије облик који је што сличнији квадрату.

Нека је $a \leq b$. Ако је $a + c \leq b$, тада се целокупан износ увећања c може додати на мању страницу a. У супротном се прво краћа страница a продужи тако да постане једнака дужој страници b, а затим се преостали износ увећања $(c - (b - a))$ подели што равномерније могуће (ако је то паран број може се добити квадрат, а ако није, тада се добија правоугаоник код којег је једна страница за један дужа од друге). У имплементацији дужине нових страница можемо израчунати тако што дужу страницу b увећамо за $\left\lfloor \frac{c-(b-a)}{2} \right\rfloor$ и за $\left\lceil \frac{c-(b-a)}{2} \right\rceil = \left\lfloor \frac{c-(b-a)+1}{2} \right\rfloor$.

```
long long maksimalniPrinos(long long a, long long b, long long c) {
  if (a > b) swap(a, b);
  if (c <= b - a)
    a += c;
  else {
    long long preostalo = c - (b - a);
    a = b + preostalo / 2;
    b = b + (preostalo + 1) / 2;
  }
  return a*b;
}
```

Анализа сложености. Сложеност овог решења је $O(1)$.

Задатак: Број дељивих у интервалу

Напиши програм који одређује колико у интервалу $[a, b]$ постоји бројева дељивих бројем k.

Улаз: Са стандардног улаза уносе се три цела броја, сваки у посебном реду.

- a $(0 \leq a \leq 10^9)$
- b $(a \leq b \leq 10^9)$
- k $(1 \leq k \leq 10^9)$

Излаз: На стандардни излаз исписати тражени цео број.

Пример

Улаз	*Излаз*
30 53 5	5

Објашњење

Бројеви су 30, 35, 40, 45 и 50.

Решење

Линеарна претрага

Могуће је направити решење засновано на претрази грубом силом, које укључује коришћење петљи. То решење је најједноставније за разумевање и имплементацију, међутим може бити неефикасно за велику разлику између бројева a и b. Потребно је у петљи, редом проћи кроз све бројеве од a до b, проверити за сваки да ли је дељив бројем k и за сваки који јесте, увећати бројач пронађених бројева. Дакле, овај алгоритам врши бројање елемената филтриране серије узастопних природних бројева из интервала $[a, b]$, а филтрирање се врши на основу провере дељивости која се своди на поређење остатака при дељењу са нулом.

```
// broj brojeva u intervalu [a, b] deljivih brojem k
int brojDeljivih(int a, int b, int k) {
  int broj = 0;
```

```
  for (int i = a; i <= b; i++)
    if (i % k == 0)
      broj++;
  return broj;
}
```

Анализа сложености. Сложеност овог решења је линеарна у односу на број елемената у интревалу тј. $O(b-a)$.

Израчунавање броја дељивих бројева

Да би број x био дељив бројем k потребно је да постоји неко n тако да је $x = n \cdot k$. Пошто x мора бити у интервалу $[a, b]$, мора да важи да је $a \leq n \cdot k$ и $n \cdot k \leq b$. Најмање n које задовољава прву неједначину једнако је $n_l = \lceil \frac{a}{k} \rceil$, највеће n које задовољава другу неједначину једнако је $n_d = \lfloor \frac{b}{k} \rfloor$. Било који број из интервала $[n_l, n_d]$ задовољава обе неједнакости и представља количник неког броја из интервала $[a, b]$ бројем k. Слично, било који број из интервала $[a, b]$ дељив бројем k даје неки количник из интервала $[n_l, n_d]$. Зато је тражени број бројева из интервала $[a, b]$ који су дељиви бројем k једнак броју бројева у интервалу $[n_l, n_d]$ а то је $n_d - n_l + 1$ ако је $n_d \geq n_l$, тј. 0 ако је тај интервал празан тј. ако је $n_d < n_l$. Бројеви n_l и n_d се могу одредити, на пример, гранањем.

```
// broj brojeva u intervalu [a, b] deljivih brojem k
int brojDeljivih(int a, int b, int k) {
  int l = a % k == 0 ? a/k : a/k + 1; // ceil(a/k)
  int d = b / k;                      // floor(b/k)
  int broj = d >= l ? d-l+1 : 0;
  return broj;
}
```

Анализа сложености. Сложеност овог решења је, јасно, константна тј. $O(1)$.

Задатак може ефикасно да се реши тако што се одреди најмањи број већи или једнак броју a који је дељив бројем k и највећи број мањи или једнак броју b који је дељив бројем k. Њиховим дељењем са k добијају се бројеви n_l и n_d описани у претходном решењу и задатак се даље решава на исти начин.

```
// broj brojeva u intervalu [a, b] deljivih brojem k
int brojDeljivih(int a, int b, int k) {
  int l = a % k == 0 ? a : (a/k + 1)*k;
  int d = b % k == 0 ? b : (b/k)*k; // moze i samo d = (b/k)*k;
  int broj = d >= l ? (d/k - l/k + 1) : 0;
  return broj;
}
```

Анализа сложености. Сложеност овог решења је, јасно, константна тј. $O(1)$.

Задатак: Растављања на збир узастопних

Напиши програм који одређује на колико се начина дати природни број n може представити као збир два или више узастопна природна броја (већа или једнака 1).

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^9$).

Излаз: На стандардни излаз исписати тражени број начина.

Пример

Улаз	*Излаз*
15	3

Објашњење

$$15 = 1 + 2 + 3 + 4 + 5 = 4 + 5 + 6 = 7 + 8$$

Решење

Испробавање свих могућности за први члан, па за дужину

Први покушај може бити решавање проблема грубом силом, тј. испробавање свих могућих првих чланова збира. Најмањи могући први члан је $a_0 = 1$. Пошто збир мора да буде бар двочлан, највећи могући први члан је онај број a_0 такав да је $a_0 + (a_0 + 1) \leq n$. Када смо одредили први члан, одређујемо колико сабирака треба узети да би се добио збир n. Крећемо од двочланог низа и затим додајемо један по један наредни сабирак све док збир не достигне или не престигне збир n. Бројач увећавамо ако је након петље збир једнак вредности n (тада је успешно нађено једно решење).

```
int brojNacina(int n) {
  int broj = 0;
  for (int a0 = 1; a0 + (a0+1) <= n; a0++) {
    int zbir = a0 + (a0+1);
    for (int ai = a0 + 2; zbir < n; ai++)
      zbir += ai;
    if (zbir == n)
      broj++;
  }
  return broj;
}
```

Анализа сложености. Спољашња петља се извршава око $\frac{n}{2}$ пута. Број извршавања унутрашње петље је теже проценити. Питамо се који је број сабирака m потребан, тако да је $a_0 + (a_0 + 1) + \ldots + (a_0 + (m - 1)) \geq n$. Ако применимо формулу за збир аритметичког низа, видимо да је тај збир једнак $m \cdot a_0 + \frac{m(m-1)}{2}$. Веома груба процена када је a_0 мало даје нам процену за m око $\sqrt{2n}$. Додуше, чим a_0 крене да расте, овај број крене да опада. Веома грубо, сложеност можемо ограничити одозго као $O(n\sqrt{n})$.

Испробавање свих могућности за дужину, па за први члан

Редослед петљи може бити другачији. Спољном петљом можемо одређивати број сабирака m, а унутрашњом испробавати вредности почетног сабирка. Крећемо од два сабирка. Највећи могући број сабирака наступа када је $a_0 = 1$, и пошто је $a_0 + (a_0 + 1) + \ldots + (a_0 + (m - 1)) = m \cdot a_0 + \frac{m(m-1)}{2}$, да би збир могао да евентуално буде n мора да важи да је $m + \frac{m(m-1)}{2} \leq n$.

```
int brojNacina(int n) {
  int broj = 0;
  for (int m = 2; m + m*(m-1)/2 <= n; m++) {
    int a0 = 1;
    int zbir = a0*m + m*(m-1)/2;
    while (zbir < n) {
      a0++;
      zbir = a0*m + m*(m-1)/2;
    }
    if (zbir == n)
      broj++;
  }
  return broj;
}
```

Анализа сложености. Сложеност је идентична као у претходном приступу и може се грубо проценити са $O(n\sqrt{n})$.

Испробавање свих могућности за дужину и израчунавање првог члана

Кључна оптимизација наступа када увидимо да нам унутрашња петља није потребна. Наиме, нема потребе испробавати различите вредности a_0, већ се a_0 може израчунати на основу m и n. Ако је $m \cdot a_0 + \frac{m(m-1)}{2} = n$, тада је $a_0 = \frac{n - \frac{m(m-1)}{2}}{m}$. Збир са m сабирака постоји ако и само ако је ово цео број (што се може проверити испитивањем остатка при дељењу бројева $n - \frac{m(m-1)}{2}$ и m). Услов $m + \frac{m(m-1)}{2} \leq n$ гарантује да је $a_0 \geq 1$.

Напоменимо да је редослед провера био веома важан, јер је једначина линеарна по a_0, а квадратна по m, тако да је за фиксирано m, a_0 прилично једноставно израчунати, док за фиксирано a_0 није једноставно израчунати

m.

```
int brojNacina(int n) {
  int broj = 0;
  for (int m = 2; m + m*(m-1)/2 <= n; m++)
    if ((n - m*(m-1)/2) % m == 0)
      broj++;
  return broj;
}
```

Анализа сложености. Сложеност једине петље, па и целог програма се може грубо оценити са $O(\sqrt{n})$ (у њеном телу се провера постојања броја a_0 врши у сложености $O(1)$).

Задатак: Највећи заједнички делилац

Мрави, пчеле и комарци организују спортски турнир и желе да се поделе у тимове, тако да се сваки тим састоји само од једне врсте инсеката, да сви тимови имају исти број чланова (да би се након рунде квалификација унутар сваке врсте могли своје представнике да пошаљу на заједнички турнир) и да је сваки инсект укључен тачно у један тим. Ако се зна број инсеката сваке од три дате врсте, напиши програм који одређује највећи могући број чланова сваког тима.

Улаз: Са стандардног улаза се уносе три броја из интервала $[1, 2 \cdot 10^9]$, сваки у посебном реду: број мрава, пчела и комараца.

Излаз: На стандардни излаз исписати један цео број - тражену величину тима.

Пример

Улаз	*Излаз*
20	10
30	
40	

Решење

Ако са a, b и c обележимо број сваке од три врсте инсеката, а са t величину сваког тима, бројеви a, b и c морају бити дељиви са t (јер сваки инсект мора бити укључен тачно у један тим). Дакле, тражимо највећи број t који дели бројеве a, b и c, а то је њихов највећи заједнички делилац (НЗД).

НЗД три броја се може одредити као НЗД од НЗД-а прва два и трећег броја, тј. важи $nzd(a, b, c) = nzd(nzd(a, b), c)$. Довољно је, дакле, да опишемо поступак одређивања НЗД два броја.

Еуклидов алгоритам

За одређивање НЗД најбоље је употребити чувени Еуклидов алгоритам. Оригинална формулација Еуклидовог алгоритма заснована је на одузимању.

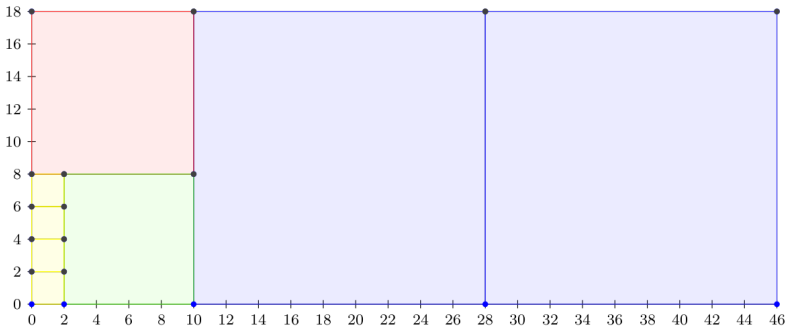
Ако су два броја једнака тј. ако је $a = b$, тада им је и њихов НЗД једнак.

У супротном се проблем може свести на проналажење НЗД мањег од два броја и разлике два броја. На пример, ако је $a > b$, тада је $nzd(a, b) = nzd(a - b, b)$.

Алгоритам се може илустровати и геометријски (и у директној је вези са проблемом одређивања максималне димензије квадрата којима може да се поплоча правоугаоно поље димензија $a \times b$.

Пример. Претпоставимо да је дат правоугаоник чије су дужине страница a и b и да је потребно одредити највећу дужину странице квадрата таква да се правоугаоник може поплочати квадратима те димензије. Ако је полазни правоугаоник димензије $a = 46$ и $b = 18$, тада се прво из њега могу исећи два квадрата димензије 18 пута 18 и остаће нам правоугаоник димензије 18 пута 10. Јасно је да ако неким мањим квадратима успемо да поплочамо тај преостали правоугаоник, да ћемо тим квадратима успети да поплочамо и ове квадрате димензије 18 пута 18 (јер ће димензија тих малих квадрата делити број 18), па ћемо самим тим моћи поплочати и цео полазни правоугаоник димензија 46 пута 18. Од правоугаоника димензије 18 пута 10 можемо исећи квадрат димензије 10 пута 10 и преостаће нам правоугаоник димензије 10 пута 8. Поново, квадратићи којима ће се моћи поплочати тај преостали правоугаоник ће бити такви да се њима може поплочати и исечени квадрат димензије 10 пута 10. Од тог правоугаоника исецамо квадрат 8 пута 8 и добијамо правоугаоник димензије 8

пута 2. Њега можемо разложити на четири квадрата димензије 2 пута 2 и то је највећа димензија квадрата којим се може поплочати полазни правоугаоник.



Слика 2.21: Поплочавање правоугаоника квадратима

Имплементација може бити рекурзивна, која директно прати претходну дефиницију. Могуће је једноставно направити и итеративну имплементацију, у којој се у сваком кораку вредност већег од два броја мења њиховом разликом.

```
// izracunavanje najveceg zajednickog delioca brojeva a i b
int nzd(int a, int b) {
  // Euklidov algoritam sa oduzimanjem
  while (a != b) {
    if (a > b)
      a -= b;
    else
      b -= a;
  }
  return a;
}

// izracunavanje najveceg zajednickog delioca brojeva a, b i c
int nzd(int a, int b, int c) {
  return nzd(nzd(a, b), c);
}
```

Доказ коректности. Ако неки број d дели бројеве a и b, тада он дели и њихову разлику. Дакле, $d = nzd(a, b)$ сигурно дели и b и $a - b$. Ако он не би био НЗД бројева $a - b$ и b, тада би постојао неки већи број d' који би делио и $a - b$ и b. Међутим, тада би d' делио и збир $a = (a - b) + b$, па би био делилац бројева a и b, који је већи од d, што је контрадикција са тим да је d НЗД бројева a и b.

Ако је $a < b$, тада се веома слично може доказати да је $nzd(a, b) = nzd(a, b - a)$.

Анализа сложености. Најгори случај наступа када је разлика између два броја јако велика (ако је $a = 1$, он ће се одузимати од b све док он не постане 1, за шта је потребно $b - 1$ корака). Може се показати да је сложеност линеарна у односу на већи од два броја, што се може записати као $O(\max(a, b))$ или $O(a + b)$. Ако се сложеност рачуна у односу на број цифара броја (што је величина улаза), онда је она заправо експоненцијална.

Поправљање сложености коришћењем дељења

Пример. Размотримо одређивање НЗД на једном примеру.

$nzd(279, 45) = nzd(234, 45) = nzd(189, 45) = nzd(144, 45) = nzd(99, 45) = nzd(54, 45) = nzd(9, 45) = nzd(9, 36) = nzd(9, 27) = nzd(9, 18) = nzd(9, 9) = 9$.

Можемо приметити дугачак низ корака у којима се мало по мало од броја 279 одузима број 45, све док се не дође до броја који је мањи од броја 45. За тим нема потребе, јер знамо да ће после тог дугог низа поступак зауставити када нам један аргумент буде баш 45, а други буде једнак остатку при дељењу броја 279 бројем 45, а то је број 9. Дакле, уместо да итеративно, тај остатак рачунамо узастопним одузимања, бољи приступ

је да применимо дељење и у једном кораку га израчунамо као остатак при дељењу. Ако сличан принцип применимо на бројеве 9 и 45, доћи ћемо до тога да ће нам остати број 9 и остатак при дељењу та два броја, што је нула. То није баш у потпуности једнако као у случају одузимања, где смо се зауставили код пара $(9, 9)$, међутим, сасвим је исправно и може се сматрати продужењем претходног поступка у ком би се пре пријављивања резултата урадило још једно одузимање и дошло се до тога да је један од бројева једнак нули, када је НЗД једнак другом броју.

Бржи Еуклидов алгоритам, у ком се користи дељење, заснован је на следећим тврђењима.

- НЗД било ког броја a и нуле је тај број a (он дели и нулу и самог себе и највећи је такав број јер није могуће да неки број већи од a дели број a).
- НЗД бројева a и b, када b није нула, једнак је НЗД броја b и остатка при дељењу a бројем b, тј. $nzd(a, b) = nzd(b, a \bmod b)$.

Приметимо да нема потребе анализирати који је број мањи, а који је већи. Ако је $a < b$, тада ће важити $a \bmod b = a$, па ће се у првом кораку добити да је $nzd(a, b) = nzd(b, a)$. Пошто је $a \bmod b < b$, једном када је први аргумент већи од другог, то ће тако остати до краја.

Еуклидов алгоритам, дакле, допушта веома једноставну рекурзивну карактеризацију.

Имплементацију Еуклидовог алгоритма можемо извршити итеративно, тако што у петљи која се извршава све док је број b већи од нуле пар променљивих (a, b) мењамо вредностима $(b, a \bmod b)$. Наивни покушај да се то уради на следећи начин:

```
a = b;
b = a % b;
```

није коректан, јер се приликом израчунавања остатка користи промењена вредност променљиве `a`. Зато је неопходно употребити помоћну променљиву. На пример:

```
tmp = b;
b = a % b;
a = tmp;
```

На крају петље вредност b једнака је нули, тако да као резултат можемо пријавити текућу вредност броја a.

```
// izracunavanje najveceg zajednickog delioca brojeva a i b
int nzd(int a, int b) {
  // Euklidov algoritam
  while (b > 0) { // dok b ne postane nula
    int tmp = b;  // par (a, b) menjamo parom (b, a % b)
    b = a % b;    // jer je nzd(a, b) = nzd(b, a % b)
    a = tmp;
  }
  return a;       // nzd(a, 0) = a
}

// izracunavanje najveceg zajednickog delioca brojeva a, b i c
int nzd(int a, int b, int c) {
  return nzd(nzd(a, b), c);
}
```

Доказ коректности. Докажимо формално коректност алгоритма. На основу дефиниције целобројног дељења важи да је $a = (a \operatorname{div} b) \cdot b + (a \bmod b)$. Обележимо са d НЗД бројева b и $a \bmod b$. Да бисмо доказали да је он уједно НЗД бројева a и b довољно је доказати да он дели та два броја и да сваки број који дели та два броја дели њега.

- Пошто број d дели бројеве b и $a \bmod b$, он дели оба сабирка на десној страни, па зато дели и њихов збир који је једнак a и зато дели и a и b.
- Даље, ако неки број d' дели бројеве a и b он мора делити и број $a \bmod b$ (јер се он може исказати као разлика два броја дељивих бројем d'), па пошто је d' делилац бројева b и $a \bmod b$, он мора делити и њихов НЗД тј. мора делити број d.

Еуклидов алгоритам који је заснован на дељењу можемо представити и на следећи начин:

$$\begin{array}{rcll} r_0 & = & a & \\ r_1 & = & b & \\ r_2 & = & r_0 - q_1 r_1, & 0 < r_2 < r_1 \\ \ldots & & & \\ r_{i+1} & = & r_{i-1} - q_i r_i, & 0 < r_{i+1} < r_i \\ \ldots & & & \\ r_k & = & r_{k-2} - q_{k-1} r_{k-1}, & 0 < r_k < r_{k-1} \\ r_{k+1} & = & r_{k-1} - q_k r_k, & 0 = r_{k+1} \end{array}$$

Вредност r_k је НЗД бројева a и b. Заиста, пошто је $r_{k+1} = 0$, важи да је $r_{k-1} = q_k r_k$, па број r_k дели r_{k-1}. Међутим, пошто је $r_{k-2} = q_{k-1} r_{k-1} + r_k$, он дели и r_{k-2}. Сличним резоновањем, уназад, може се закључити да r_k дели и r_1 и r_0 тј. a и b. Обратно, ако неки број дели и a и b онда он дели и r_0 и r_1, а пошто је $r_2 = r_0 - q_1 r_1$, он дели и r_2. Сличним резоновањем, унапред, може се закључити да тај број мора делити и r_k. Стога је r_k НЗД бројева a и b.

У сваком кораку алгоритма одржавамо две узастопне вредности низа r_i. У почетку, то су чланови r_0 и r_1 тј. оригиналне вредности a и b. Важи да је $q_1 = a \operatorname{div} b$, а $r_2 = a \bmod b$. У другом кораку промењиве a и b треба да имају вредности чланова r_1 и r_2, што значи да се пар промењивих a, b мења вредностима b и $a \bmod b$, што је управо оно што смо радили у претходно описаном коду. Поступак се наставља све док пар узастопних вредности не постане r_k, r_{k+1}, тј., пошто пар узастопних вредности одржавамо у промењивама a и b, док b не постане нула и тада је НЗД који је једнак r_k садржан у промењивој a.

Анализа сложености. Приметимо да се после највише једног корака осигурава да је $a > b$ (јер се у сваком кораку пар (a, b) мења паром $(b, a \bmod b)$, а увек важи да је $a \bmod b < b$). После било које две итерације се од пара (a, b) долази до пара $(a \bmod b, b \bmod (a \bmod b))$ (наравно, под претпоставком да је $b \neq 0$ и да је $a \bmod b \neq 0$). Докажимо да је $a \bmod b < a/2$. Ако је $b \leq a/2$, тада је $a \bmod b < b \leq a/2$. У супротном, за $b > a/2$ важи да је $a \bmod b = a - b < a/2$. Зато се први аргумент после свака два корака смањи бар двоструко. До вредности 1 први аргумент стигне у логаритамском броју корака у односу на већи од полазна два броја и тада други број сигурно достиже нулу (јер је строго мањи од првог) и поступак се завршава. Дакле, сложеност је логаритамска у односу на већи од два броја, што може да се запише и као $O(\log(a + b))$. Ако се сложеност рачуна у односу на број цифара броја (што је величина улаза), онда је она заправо линеарна.

2.11 Одсецање

Један од основних принципа за добијање ефикаснијих алгоритама и програма је да рачунар не треба да израчунава ствари за које се унапред може проценити да нису потребне за добијање коначног решења проблема. Важан пример овог принципа се јавља код алгоритама претраге. Претрагу елемената не треба експлицитно вршити међу елементима за које се може унапред утврдити да не могу да задовоље услов претраге. Када прескочимо проверу таквих елемената, кажемо да смо учинили **одсецање у претрази**. Сличан принцип се примењује и када се врши оптимизација (тражи најмањи односно највећи елемент), када се може прескочити анализа елемената за које се унапред може доказати да су мањи (односно већи) од траженог максимума (односно минимума).

Да би се осигурала коректност алгоритама у којима се врши одсецање увек је потребно веома пажљиво утврдити да је одсецање оправдано и да се у делу простора претраге који се не испитује заиста не може налазити решење проблема.

У наставку овог поглавља ћемо кроз одређен број примера приказати како се одсецањем постиже асимптотски ефикаснији алгоритам. Један од најзначајнијих примера одсецања представља *бинарна претрага*, која ће, због свог значаја бити анализирана у посебном поглављу. Одсецање се примењује и у другим облицима претраге (бектрекинг претрази, претрази у дубину, претрази у ширину), о чему ће више бити речи у каснијим поглављима.

Задатак: Прост број

Напиши програм који испитује да ли је унети природан број прост (већи је од 1 и нема других делилаца осим 1 и самог себе).

Улаз: Са стандардног улаза се уноси природан број n ($1 \leq n \leq 10^9$).

Излаз: На стандардни излаз исписати `DA` ако је број n прост тј. `NE` ако није.

Пример		**Пример 2**	
Улаз	*Излаз*	*Улаз*	*Излаз*
`17`	`DA`	`903543481`	`NE`

Решење

Линеарна претрага свих потенцијалних делилаца

Природан број је прост ако је већи од 1 и ако није дељив ни са једним бројем осим са један и са самим собом. По дефиницији број 1 није прост. Дакле, број већи од 1 је прост ако нема ни једног правог делиоца. Потребно је дакле извршити претрагу скупа потенцијалних делилаца и проверити да ли неки од њих стварно дели број n. Имплементација се заснива на алгоритму линеарне претраге. Скуп потенцијалних делилаца је скуп свих природних бројева од 2 до $n - 1$ и наивна имплементација их све проверава.

```
// funkcija koja proverava da li je dati broj prost
bool prost(int n) {
  if (n == 1) return false;
  for (int i = 2; i < n; i++)
    if (n % i == 0)
      return false;
  return true;
}
```

Анализа сложености. Пошто се провера сваког делиоца извршава израчунавањем једног остатка при дељењу, сложеност овог приступа одговара броју делилаца и једнака је $O(n)$.

Одсецање у претрази

Делиоци броја се увек јављају у пару. На пример, делиоци броја 100 организовани по паровима су (1, 100), (2, 50), (4, 25) (5, 20) и (10, 10). Ако је i делилац броја n, делилац је и број $\frac{n}{i}$. При том, ако је $i \geq \sqrt{n}$, тада је $\frac{n}{i} \leq \sqrt{n}$. Дакле, важи следећа теорема.

Теорема. Природан број $n \geq 2$ има праве делиоце који су већи или једнаки вредности $\sqrt{n}$ ако и само ако има делиоце који су мањи или једнаки вредности $\sqrt{n}$.

Ова теорема нам даје могућност да претрагу потенцијалних делилаца редукујемо само на интервал $[2, \sqrt{n}]$, јер ако број нема делилаца мањих или једнаких вредности $\sqrt{n}$, онда не може да има делилаца већих или једнаких тој вредности, тј. нема правих делилаца и прост је. Ово је пример алгоритма у ком се ефикасност значајно поправља тако што је елиминисан (одсечен) значајан део простора претраге за који можемо да докажемо да га није неопходно проверавати.

Сама имплементација је једноставна и заснива се на алгоритму линеарне претраге. У посебној функцији на почетку проверавамо специјалан случај броја 1 (ако је n једнако 1, враћамо вредност `false`). Након тога, у петљи проверавамо потенцијалне делиоце од 2 до $\sqrt{n}$. Један начин да одредимо горњу границу је да употребимо библиотечку функцију `sqrt`. Међутим, рад са реалним бројевима је могуће у потпуности избећи тако што се уместо услова $i \leq \sqrt{n}$ употреби услов $i \cdot i \leq n$. За сваку вредност i проверава се да ли је делилац броја i (израчунавањем остатка при дељењу). Чим се утврди да је i делилац броја n функција може да врати `false` (тиме се уједно прекида извршавање петље). На крају петље, функција може да врати `true`, јер није пронађен ниједан делиоц мањи или једнак од $\sqrt{n}$, па на основу теореме које смо доказали не може постојати ни један делилац изнад те вредности и број је прост.

```
// funkcija koja proverava da li je dati broj prost
bool prost(int n) {
  if (n == 1) return false;
  for (int i = 2; i*i <= n; i++)
    if (n % i == 0)
      return false;
  return true;
}
```

Анализа сложености. Сложеност овог алгоритма је $O(\sqrt{n})$. Обратите пажњу на то да је ово скраћивање интервала претраге веома значајно (ако је највећи број око 10^9 тј. око милијарду, уместо милијарду делилаца потребно је проверавати само њих корен из милијарду, што је тек нешто изнад тридесет хиљада).

Могуће је правити и другачије имплементације истог алгоритма.

```
// funkcija koja proverava da li je dati broj prost
bool prost(int n) {
  int i = 2;
  while (i*i <= n && n % i != 0)
    i++;
  return n > 1 && i*i > n;
}
```

Провера само непарних бројева

Још једна могућа оптимизација је да се на почетку провери да ли је број паран а да се након тога проверавају само непарни делиоци, међутим, та оптимизација не доноси превише (обилазак до корена смањује број потенцијалих кандидата са милијарде на тек тридесетак хиљада, а провера само непарних делилаца тај број смањује на петнаестак хиљада, што је сразмерно знатно мања уштеда).

```
// funkcija koja proverava da li je dati broj prost
bool prost(int n) {
  if (n == 1) return false;     // broj 1 nije prost
  if (n == 2) return true;      // broj 2 jeste prost
  if (n % 2 == 0) return false; // ostali parni brojevi nisu prosti
  // proveravamo neparne delioce od 3 do korena iz n
  for (int i = 3; i*i <= n; i += 2)
    if (n % i == 0)
      return false;
  // nismo nasli delioca - broj jeste prost
  return true;
}
```

Анализа сложености. Сложеност овог алгоритма је $O(\sqrt{n})$, али се провером само непарних бројева константни фактор смањио два пута.

Провера само бројева облика 6k-1 и 6k+1

Програм се још мало може убрзати ако се примети да су сви прости бројеви већи од 2 и 3 облика $6k-1$ или $6k+1$, за $k \geq 1$ (наравно, обратно не важи). Заиста, бројеви облика $6k$, $6k+2$ и $6k+4$ су сигурно парни тј. дељиви са 2, бројеви облика $6k+3$ су дељиви са 3, тако да су једини преостали $6k+1$ и $6k+5$, при чему су ови други сигурно облика $6k'-1$ (за $k'=k+1$). Дакле, уместо да проверавамо дељивост са свим непарним бројевима мањим од корена, можемо проверавати дељивост са свим бројевима облика $6k-1$ или $6k+1$, чиме избегавамо проверу са једним на свака три непарна броја и програм убрзамо сходно томе.

```
// funkcija koja proverava da li je dati broj prost
bool prost(int n) {
  if (n == 1 ||
      (n % 2 == 0 && n != 2) ||
      (n % 3 == 0 && n != 3))
    return false;
  for (int k = 1; (6*k - 1) * (6*k - 1) <= n; k++)
    if (n % (6 * k + 1) == 0 || n % (6 * k - 1) == 0)
      return false;
  return true;
}
```

Анализа сложености. Сложеност овог алгоритма је $O(\sqrt{n})$, али се провером само бројева облика 6k-1 и 6k+1 константни фактор смањио три пута у односу на први алгоритам ове сложености.

Ако је потребно за више бројева одједном проверити да ли су прости, уместо проверавања сваког појединачног, боље је употребити Ератостеново сито. Тај алгоритам је описан у задатаку Ератостеново сито.

Задатак: Ератостеново сито

Напиши програм који одређује број простих бројева у интервалу $[a, b]$ и њихов збир (ако збир има више од 6 цифара, исписати само остатак при дељењу са 1000000).

Улаз: Са стандардног улаза уносе се бројеви a и b ($1 \leq a \leq b \leq 10^7$), сваки у посебној линији.

Излаз: На стандардном излазу приказати у једној линији, одвојени једним бланко знаком, број простих бројева из интервала $[a, b]$ и тражени збир.

Пример

Улаз	*Излаз*
`1` `1000`	`168 76127`

Решење

Појединачне провере простих бројева

Очигледан алгоритам за одређивање свих простих бројева из неког интервала јесте да се за сваки број из тог интервала појединачно провери да ли је прост. Ово може бити урађено уз помоћ алгоритма тј. функције коју смо описали у задатку Прост број.

На основу спецификације задатка потребно је одредити највише 6 последњих цифара збира свих простих бројева из интервала $[a, b]$, што, је еквивалентно одређивању збира тих бројева по модулу 10^6. Наиме, важи $(a + b) \bmod m = (a \bmod m + b \bmod m) \bmod m$. У петљи пролазимо кроз све бројеве од a до b, вршимо филтрирање на основу услова да је број прост и вршимо бројање и сабирање добијене филтриране серије.

Напоменимо да се збир рачуна тако што се на почетку иницијализује на нулу, а затим се у сваком кораку израчунава сабирање збира и текућег простог броја по модулу 10^6 (`zbir = (zbir % 1000000 + p % 100000) % 100000`). Пошто ће у сваком кораку збир бити мањи од 10^6, и пошто не постоји опасност од прекорачења када се у обзир узме максимална вредност простих бројева који се сабирају, претходни корак се може заменити кораком `zbir = (zbir + p) % 1000000`.

```
// funkcija koja proverava da li je dati broj prost
bool prost(int n) {
  if (n == 1) return false;     // broj 1 nije prost
  if (n == 2) return true;      // broj 2 jeste prost
  if (n % 2 == 0) return false; // ostali parni brojevi nisu prosti
  // proveravamo neparne delioce od 3 do korena iz n
  for (int i = 3; i*i <= n; i += 2)
    if (n % i == 0)
      return false;
  // nismo nasli delioca - broj jeste prost
  return true;
}

// funkcija odredjuje broj i zbir po modulu 1000000 prostih brojeva iz
// intervala [a, b]
void prostiUIntevalu(int a, int b, int& broj, int& zbir) {
  zbir = 0, broj = 0;
  for (int i = a; i <= b; i++)
    if (prost(i)) {
      zbir = (zbir + i) % 1000000;
      broj++;
    }
}
```

Анализа сложености. Ако се провера да ли је дати број k прост врши у сложености $O(\sqrt{k})$, тада је овај алгоритам сложености $O((b - a)\sqrt{b})$. Ако је интервал облика $[0, n]$, сложеност је $O(n\sqrt{n})$.

Ератостеново сито

Бољи резултат од испитивања за сваки број појединачно да ли је прост може се добити применом алгоритма познатог као Ератостеново сито. Основна идеја алгоритма је да се прво напишу сви бројеви од 1 до датог броја n, затим да се прецрта број 1 (јер он по дефиницији није прост), након њега сви умношци броја 2 (нису прости зато што су дељиви са 2, док број 2 остаје непрецртан јер је он прост), затим умношци броја 3 (нису прости јер су дељиви бројем 3), затим умношци броја 5 (нису прости зато што су дељиви бројем 5) и тако даље.

Ефикасна имплементација овог алгоритма подразумева неколико одсецања (којима се избегава понављање истих операција више пута и асимптотски убрзава алгоритам).

Прво, умношке сложених бројева нема потребе посебно прецртавати јер су они већ прецртани током прецртавања умножака неког од њихових простих фактора (на пример, нема потребе посебно прецртавати умношке броја 4 јер су они већ прецртани током прецртавања умножака броја 2).

Друго, приликом прецртавања умножака броја d довољно је кренути од $d \cdot d$ јер су мањи умношци већ прецртани раније (сви имају праве факторе мање од d). Зато је потребно је да се поступак понавља само док се не прецртају умношци свих простих бројева који (прости бројеви, а не умношци) нису већи од корена броја n. За бројеве веће од корена од n прецртавање би кренуло од њиховог квадрата који је већи од n, па је јасно да се ни за један од њих ништа додатно не би прецртало.

Бројеви који су остали непрецртани су прости (јер знамо да немају правих делилаца мањих или једнаких корену од n, па самим тим и мањих или једнаких свом корену, а пошто немају делилаца испод вредности корена, немају правих делилаца ни изнад вредности корена). Та теорема је доказана у задатку Прост број.

Пример. Прикажимо како се овим алгоритмом одређују сви прости бројеви од 2 до 50. Крећемо од пуне табеле у којој су уписани сви бројеви од 2 до 50.

```
 .  2  3  4  5  6  7  8  9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
```

У првом кораку прецртавамо све умношке броја 2 (осим самог броја 2).

```
 .  2  3  .  5  .  7  .  9  .
11  . 13  . 15  . 17  . 19  .
21  . 23  . 25  . 27  . 29  .
31  . 33  . 35  . 37  . 39  .
41  . 43  . 45  . 47  . 49  .
```

У наредном кораку прецртавамо све умношке броја 3, кренувши од његовог квадрата тј. од 9 (број 6 је већ прецртан као умножак броја 2).

```
 .  2  3  .  5  .  7  .  .  .
11  . 13  .  .  . 17  . 19  .
 .  . 23  . 25  .  .  . 29  .
31  .  .  . 35  . 37  .  .  .
41  . 43  .  .  . 47  . 49  .
```

Умношке броја 4 не прецртавамо, јер је он прецртан (па самим тим и сви његови умношци).

У наредном кораку прецртавамо све умношке броја 5, кренувши од његовог квадрата тј. броја 25 (умношци $2 \cdot 5$, $3 \cdot 5$ и $4 \cdot 5$ су већ прецртани).

```
 .  2  3  .  5  .  7  .  .  .
11  . 13  .  .  . 17  . 19  .
 .  . 23  .  .  .  .  . 29  .
31  .  .  .  .  . 37  .  .  .
41  . 43  .  .  . 47  . 49  .
```

Умношке броја 6 не прецртавамо, јер је он прецртан (па самим тим и сви његови умношци).

Прецртавамо умношке броја 7, кренувши од његовог квадрата тј. броја 49.

```
 .  2  3  .  5  .  7  .  .  .
11  . 13  .  .  . 17  . 19  .
 .  . 23  .  .  .  .  . 29  .
31  .  .  .  .  . 37  .  .  .
41  . 43  .  .  . 47  . 49  .
```

Умношке броја 8 не прецртавамо, јер је он прецртан (па самим тим и сви његови умношци). Међутим, прецртавање би кренуло од његовог квадрата. И за све наредне бројеве прецртавање креће од њиховог квадрата, међутим, тај квадрат је већ ван табеле (јер је већи од 50), па се поступак може завршити. Преостали бројеви су прости.

Прецртавање бројева моделоваћемо низом (или вектором) који садржи логичке вредности (вредности типа `bool`) и прецртане бројеве означаваћемо са `false`, а непрецртане са `true`. Одређивање простих бројева (помоћу поменутог низа тј. вектора) реализоваћемо у засебној функцији, јер та функција може бити корисна и у многим наредним задацима.

Рецимо и да је без обзира на то што су нама потребни само бројеви из интервала од a до b, у Ератостеновом ситу потребно вршити анализу свих бројева из интервала од 0 до b (јер се прецртавање мора вршити и бројевима мањим од a).

```
// funkcija koja popunjava logicki niz podacima o prostim brojevima iz
// intervala [0, n]
void Eratosten(vector<bool>& prost, int n) {
  // alociramo potreban prostor
  prost.resize(n + 1, true);
  prost[0] = prost[1] = false; // 0 i 1 po definiciji nisu prosti
  // brojevi ciji se umnosci precrtavaju
  for (int i = 2; i * i <= n; i++)
    // nema potrebe precrtavati umnoske slozenih brojeva
    if (prost[i]) {
      // precrtavamo umnoske broja i i to krenuvsi od i*i
      for (int j = i * i; j <= n; j += i)
        prost[j] = false;
    }
}

// funkcija odredjuje broj i zbir po modulu 1000000 prostih brojeva iz
// intervala [a, b]
void prostiUIntevalu(int a, int b, int& broj, int& zbir) {
  // odredjujemo proste brojeve u intervalu [0, b]
  vector<bool> prost;
  Eratosten(prost, b);

  // analiziramo jedan po jedan broj u intervalu
  zbir = 0; broj = 0;
  for (int i = a; i <= b; i++)
    if (prost[i]) {
      zbir = (zbir + i) % 1000000;
      broj++;
    }
}
```

Анализа сложености. Анализа сложености је компликованија и захтева одређено (додуше веома елементарно) познавање теорије бројева. Проценимо број извршавања тела унутрашње петље. У почетном кораку спољне петље прецртава се око $\frac{n}{2}$ елемената. У наредном, око $\frac{n}{3}$. У наредном кораку је број 4 већ прецртан, па се не прецртава ништа. У наредном се прецртава око $\frac{n}{5}$, након тога опет ништа, затим $\frac{n}{7}$ итд. У последњем кораку се прецртава око $\frac{n}{\sqrt{n}}$ елемената. Дакле, број прецртавања је највише

$$\frac{n}{2}+\frac{n}{3}+\frac{n}{5}+\ldots+\frac{n}{\sqrt{n}}=n\cdot\left(\sum_{\substack{d\text{ prost},\\ d\leq\sqrt{n}}}\frac{1}{d}\right)$$

Број је заправо и мањи, јер приликом прецртавања у унутрашњој петљи прецртавање не крећемо од d, већ од d^2, али за потребе лакшег одређивања горње границе сложености користићемо претходну оцену.

Још је велики Ојлер открио да се збир $H(m)=1+1/2+1/3+\ldots+1/m=\sum_{d\leq m}\frac{1}{d}$ (такозвани хармонијски збир) асимптотски понаша слично функцији $\log m$ (разлика између ове две функције тежи такозваној Ојлер-Маскеронијевој константи $\gamma\approx 0.5772156649$), па самим тим знамо да тај збир дивергира. Такође, открио је да када се сабирање врши само по простим бројевима, тада се збир понаша као логаритам хармонијског збира, тј. као $\log\log m$ (па је и он дивергентан). Дакле, у нашем примеру можемо закључити да је број прецртавања једнак $n\cdot\log\log\sqrt{n}$. Пошто је $\log\log\sqrt{n}=\log\log n^{\frac{1}{2}}=\log\left(\frac{1}{2}\log n\right)=\log\frac{1}{2}+\log\log n$, под претпоставком да је сабирање бројева (које се користи у имплементацији петљи) константне сложености, важи да је сложеност Ератостеновог сита $O(n\cdot\log\log n)$. Иако није линеарна, функција $\log\log n$ толико споро расте, да се за све практичне потребе Ератостаново сито може сматрати линеарним у односу на n (што је доста спорије само од испитивања да ли је број n прост, што има сложеност $O(\sqrt{n})$, али је брже од проверавања сваког броја појединачно које је сложености $O(n\sqrt{n})$).

Задатак: Најдужа серија победа

Кошаркашки тим је играо пуно утакмица у сезони. У свакој утакмици остварио је или победу или пораз. Напиши програм који одређује дужину најдуже серије победа у узастопним мечевима током сезоне.

Улаз: Са стандардног улаза се уноси природан број N ($5\leq N\leq 50000$), а затим и N бројева -1 (што означава пораз) или 1 (што означава победу).

Излаз: На стандардни излаз исписати један природан број који представља тражену дужину најдуже серије узастопних победа.

Пример

Улаз	*Излаз*
8 1 1 -1 1 1 1 -1 -1	3

Решење

Груба сила

Постоји неколико начина да се наивно приступи решавању овог проблема и сви претпостављају да су у програму резултати утакмица смештени у низ. Врло је вероватно да би сваки програмер са имало искуства избегао оваква решења, али приказаћемо их, да бисмо систематично илустровали неке технике поступне оптимизације кода.

Једно веома наивно решење је да анализирамо све могуће сегменте низа одређене свим могућим вредностима променљивих $0\leq i\leq j<n$. Њих можемо набројати угнежђеним петљама.За сваки сегмент можемо применом линеарне претраге проверити да ли садржи само победе и ако садржи, ажурирати максимум у складу са тим.

```
// izracunava duzinu najduze serije pobeda za dati niz rezultata utakmica
int najduzaSerijaPobeda(const vector<int>& a) {
  // broj utakmica
  int N = a.size();
```

```
  // duzina najduze serije pobeda
  int maxDuzina = 0;
  // analiziramo sve segmente a[i, j]
  for (int i = 0; i < N; i++) {
    for (int j = i; j < N; j++) {
      // proveravamo da li su u segmentu a[i, j] samo pobede
      bool samo_pobede = true;
      for (int k = i; k <= j; k++)
        if (a[k] != 1) {
          samo_pobede = false;
          break;
        }
      // ako jesu, azuriramo maksimum u odnosu na duzinu segmenta [i, j]
      if (samo_pobede)
        maxDuzina = max(maxDuzina, j - i + 1);
    }
  }

  return maxDuzina;
}
```

Анализа сложености. Пошто се резултати утакмица смештају у помоћни низ, меморијска сложеност је $O(n)$. Ово решење је изразито неефикасно (временска сложеност оваквог приступа је чак кубна тј. $O(n^3)$. Веома грубо се та сложеност може проценити тако што се примети да се обрађује $O(n^2)$ сегмената, сваки у линеарној сложености. Детално, прецизије извођење те сложености, приказано је у задатку Сегмент датог збира у низу целих бројева. Са друге стране коректност овог решења је заиста тривијално образложити (оно се потпуно директно изводи из саме формулације задатка).

Најдужа серија за сваки леви крај

Мало бољи приступ је да за сваку партију i одредимо најдужи сегмент победа који почиње на тој позицији. То се може урадити тако што се сегмент који почиње на позицији i проширује од позиције i надесно, све док се у њему налазе само победе. Важна уштеда на овом месту је то што ако знамо да су све победе у неком интервалу $[i, j]$ и да је победа и у утакмици $j + 1$, аутоматски знамо да су све победе и у интервалу $[i, j + 1]$ (кажемо да проверу вршимо инкрементално).

Када се наиђе на први пораз (или на крај), нађен је најдужи сегмент победа који почиње на позицији i, јер ће сва продужавања сегмента надесно, ако их има, садржати и тај пораз и неће више представљати серије победа. Дакле, на овом месту вршимо одсецање претраге прескачући многе сегменте за које се унапред зна да не могу задовољити наметнути услов. Такође, поређење са максималном дужином вршимо тек када максимално проширимо текући сегмент, јер унапред знамо да су сви подсегменти тог максимално проширеног сегмента краћи од њега (и овде заправо вршимо одређено одсецање).

```
// izracunava duzinu najduze serije pobeda za dati niz rezultata utakmica
int najduzaSerijaPobeda(const vector<int>& a) {
  // broj utakmica
  int N = a.size();

  // duzina najduze serije pobeda
  int maxDuzina = 0;
  // za svaku poziciju i odredjujemo duzinu najduze serije pobeda koja
  // pocinje na poziciji i
  for (int i = 0; i < N; i++) {
    int duzina = 0;
    for (int j = i; j < N && a[j] == 1; j++)
      duzina++;
    // ako je duzina serije koja se zavrsava na poziciji i veca od
    // maksimalne do tada vidjene duzine, azuriramo maksimalnu duzinu
    if (duzina > maxDuzina)
      maxDuzina = duzina;
```

```
  }

  return maxDuzina;
}
```

Анализа сложености. Ово решење је сложености $O(n^2)$, што је боље од првог, међутим и даље субоптимално. Пошто се резултати утакмица смештају у помоћни низ, меморијска сложеност је $O(n)$.

Одсецање непотребних израчунавања

Програм се може додатно значајно убрзати даљим одсецањем. У претходном решењу за сваку позицију i одређујемо дужину најдужег сегмента победа који почиње на позицији i. Једном када одредимо да је то сегмент $[i, j]$, време значајно можемо уштедети тако што приметимо да ни један сегмент победа који почиње на позицијама након i, а закључно са j не може бити дужи од сегмента који почиње са i (јер ако позиција j није последња у низу, на позицији иза ње се налази нула). Зато је након ширења сегмента који почиње на позицији i надесно и одређивања серије победа који почињу на позицији i могуће директно прећи на израчунавање најдужег сегмента победа који почиње на позицији $j + 1$ (ако таква постоји). Ово заправо одговара томе да цео низ изделимо на сегменте победа који се надовезују (пресечени сегментима пораза). Сложеност таквог приступа је $O(n)$, јер се границе сегмената само увећавају и никада не смањују.

Овај алгоритам је заправо и врло интуитиван и вероватно је први алгоритам који би програмер са мало искуства имплементирао: крећемо од почетка, проналазимо серију победа који почиње на почетку, након тога тражимо серију победа након те прве серије, затим серију након те друге и тако даље. Дакле, цео низ делимо на мање сегменте који се надовезују један иза другог, при чему је подела таква да је сваки од тих сегмената оптималан у смислу да га није могуће продужити (ни на лево, ни на десно).

Можемо приметити да нам током имплементације није више неопходно да памтимо све резултате у низу истовремено. У једној петљи ћемо читати резултате мечева и у сваком тренутку одржавати дужину текуће и дужину најдуже до тада обрађене серије (сегмента) победа. Пошто на почетку нисмо видели још ни један резултат, обе променљиве иницијализујемо на нулу. Затим, у петљи, учитавамо и обрађујемо резултате утакмица. Ако је тим победио, тада текућа утакмица продужава текућу серију победа и њену дужину увећавамо за један. Ако је тим изгубио, тада се прекида евентуална серија победа и текућа серија победа има дужину 0 (јер је утакмица којом та серија почиње изгубљена).

Након завршетка читања сваке серије победа потребно је ажурирати дужину најдуже серије (ако је дужина текуће серије дужа од до тада најдуже, потребно је дужину најдуже поставити на дужину текуће). То се дешава или када се наиђе на утакмицу у којој је тим поражен или након петље, када је последња евентуална серија победа завршена. Треба бити веома обазрив да се не заборави на последњу серију победу, тј. да се не заборави поређење текуће и најдуже серије након завршетка петље.

```
// broj utakmica
int N;
cin >> N;
// duzina tekuce serije pobeda
int duzinaTekuce = 0;
// duzina najduze do sada vidjene serije pobeda
int duzinaNajduze = 0;
// ucitavamo podatke o svim utakmicama
for (int i = 0; i < N; i++) {
  // rezultat utakmice
  int rezultat;
  cin >> rezultat;
  if (rezultat == 1) {
    // ako je tim pobedio, to produzava tekucu seriju pobeda
    duzinaTekuce++;
  } else {
    // ako je upravo prekinuta serija pobeda duza od najduze,
    // azuriramo duzinu najduze
    if (duzinaTekuce > duzinaNajduze)
      duzinaNajduze = duzinaTekuce;
    // procitali smo poraz, pa u tekucoj seriji nema pobeda
    duzinaTekuce = 0;
```

```
  }
}
// vrsimo proveru i za poslednju seriju
if (duzinaTekuce > duzinaNajduze)
  duzinaNajduze = duzinaTekuce;

// ispisujemo konacan rezultat
cout << duzinaNajduze << endl;
```

Анализа сложености. Сложеност овог алгоритма је $O(n)$. Меморијска сложеност је $O(1)$.

Задатак: Број растућих сегмената

Дат је низ a целих бројева, дужине n. Написати програм којим се одређује на колико начина можемо изабрати растуће сегменте у низу. Растући сегмент чине узастопни елементи низа $a_p < a_{p+1} < \ldots < a_q, 0 \leq p < q < n$.

Улаз: Прва линија стандардног улаза садржи природан број n ($2 \leq n \leq 10000$), број елемената низа. У свакој од n наредних линија стандардног улаза, налази по један члан низа.

Излаз: На стандардном излазу приказати у једној линији број растућих сегмената датог низа.

Пример

Улаз	*Излаз*
5 1 3 4 -2 10	4

Објашњење

То су низови [1, 3], [1, 3, 4], [3, 4], [-2, 10].

Решење

Груба сила

Задатак можемо решити анализирајући све сегменте датог низа a и за сваки сегмент $a_i, a_{i+1}, ..., a_j$ где је $0 \leq i < j < n$ проверити да ли је растући и у складу са тим увећати бројач растућих сегмената.

Анализа сложености. Пошто сегмената има $O(n^2)$ и сваком се провера монотоности врши у линеарној сложености, ово решење је сложености $O(n^3)$ (наравно, нису сви сегменти исте дужине и прецизна анализа би требало да у обзир узме дужине свих сегмената, међутим, и на тај начин би се израчунало да је алгоритам кубне сложености). Елементи су смештени у низ, па је меморијска сложеност $O(n)$.

Број растућих сегмената за сваки леви крај

Ефикасније решење се може добити ако се провера монотоности врши инкрементално (да би се проверило да је сегмент $[i, j]$ растући довољно је да је $a_j > a_{j-1}$ и да је сегмент $[i, j-1]$ растући или једночлан).

Време извршавања можемо унапредити и одсецањем. Приметимо да ако сегмент који чине елементи на позицијама $[i, j]$ није растући, онда нису растући ни сегменти $[i, j']$ за $j \leq j' < n$, па за те сегменте не треба вршити проверу, што значи да је бројање растућих сегмената који почињу на позицији i могуће прекинути чим се пронађе неки сегмент који почиње на тој позицији и није растући.

Дакле, за сваку позицију i у низу (за свако $0 \leq i < n-1$) анализирамо један по један сегмент $[i, j]$ који на тој позицији почиње све док су ти сегменти растући и за сваки растући сегмент увећавамо бројач растућих сегмената за 1. Чим наиђемо на сегмент који није растући (тј. на елемент који је мањи од претходног), прелазимо на наредну позицију i.

Пример. На пример у низу $[1, 3, 4, 5, 2, 6]$ анализирамо сегменте који почињу првим елементом низа тј. елементом a_0 све док су сегменти растући, при томе пребројимо растуће сегменте $[1, 3]$; $[1, 3, 4]$ и $[1, 3, 4, 5]$.

Слично полазећи од другог елемента низа пребројимо растуће сегменте $[3, 4]$ и $[3, 4, 5]$. Настављајући исти поступак за остале елементе низа пребројимо и растући сегмент $[4, 5]$, а затим и $[2, 6]$.

Анализа сложености. Сложеност овог алгоритма је $O(n^2)$ и њиме се најефикасније могуће експлицитно набрајају сви растући сегменти. Међутим, у задатку је потребно израчунати само њихов број (а не и набројати их експлицитно), а то се може урадити и ефикасније (у сложености $O(n)$). Меморијска сложеност је $O(n)$.

```
long long brojRastucihSegmenata(const vector<int>& a) {
  // velicina niza
  int n = a.size();
  // ukupan broj rastucih serija
  long long brojRastucih = 0;
  // za svaku poziciju u nizu
  for (int i = 0; i < n - 1; i++) {
    // pronalazimo sve rastuce serije koje pocinju na toj poziciji
    // proveru da li je serija odredjena pozicijama [i, j] rastuca
    // odredjujemo inkrementalno
    // postupak prekidamo cim se naidje na seriju koja nije rastuca
    for (int j = i + 1; j < n; j++)
      if (a[j] > a[j-1])
        brojRastucih++;
      else
        break;
  }
  return brojRastucih;
}
```

Максимални растући сегменти

Приметимо да у претходном примеру анализирајући растући сегмент који почиње од елемента a_0 пролазимо и по растућим сегментима низа који почињу са a_1 и a_2. Ако је сегмент $[a_i, a_{i+1}, ..., a_j]$ растући, онда унапред знамо да су растући и сегменти $[a_i, a_{i+1}]$, $[a_i, a_{i+1}, a_{i+2}]$, ..., $[a_i, a_{i+1}, \ldots, a_j]$, затим $[a_{i+1}, a_{i+2}]$, ..., $[a_{i+1}, a_{i+2}, \ldots, a_j]$, па све до $[a_{j-1}, a_j]$, а да сегменти $[a_i, a_{i+1}, \ldots, a_{j+1}]$, ..., $[a_i, a_{i+1}, \ldots, a_{n-1}]$, затим $[a_{i+1}, a_{i+2}, \ldots, a_{j+1}]$, ..., $[a_{i+1}, a_{i+2}, \ldots, a_{n-1}]$ итд., закључно са $[a_j, \ldots, a_{n-1}]$ нису растући. Дакле, за сваку позицију из интервала $[i, j]$ тачно знамо све растуће сегменте који на њој почињу.

Растући сегмент $[a_i, a_{i+1}, \ldots, a_{i+k-1}]$ дужине k у себи садржи:

- $k - 1$ растућих сегмената који почињу са a_i
- $k - 2$ растућих сегмената који почињу са a_{i-1}
- ...
- 1 растући сегменат који почиње са a_{i+k-2}

укупно $(k - 1) + (k - 2) + ... + 1$ растућих сегмената што износи $\frac{k \cdot (k-1)}{2}$.

До истог закључка можемо доћи и на следећи начин: сваки подсегмент $a_p, a_{p+1}, \ldots, a_q$ где је $i \leq p < q \leq i + k - 1$ растућег сегмента $a_i, a_{i+1}, ..., a_{i+k-1}$ је растући, почетак p и крај q подсегмента можемо изабрати на $\frac{k \cdot (k-1)}{2}$ начина.

Дакле, потребно је пронаћи дужине максималних растућих сегмената (оних који се не могу продужити додатним елементом да и даље остају растући), а затим извршити одсецање тако што нећемо засебно анализирати сваки леви крај (јер након налажења неког максималног растућег сегмента $[i, j]$, унапред, без провере, знамо број растућих сегмената који почињу на свим позицијама између i и j).

Према томе, у петљи анализирамо низ члан по члан почев од првог члана ($i = 0$) и одређујемо дужину t текућег растућег сегмента (ако је $a_i < a_{i+1}$ увећавамо t за 1). Када дођемо до краја текућег растућег сегмента увећамо укупан број растућих сегмената br за $\frac{t \cdot (t-1)}{2}$ и почињемо анализу следећег растућег сегмента ($t = 1$). До краја максималног растућег сегмента се може стићи на два начина: или када је $a_i \geq a_{i+1}$ или када се дође до краја низа. Напоменимо да за тај последњи растући сегмент увећавамо укупан број растућих сегмената изван петље (честа грешка је да се то заборави).

У сваком тренутку је довољно поредити само суседна члана низа, тако да није неопходно цео низ памтити у меморији, већ само два суседна члана низа (претходни и текући).

Анализа сложености. На претходно описан начин добијамо решење једним проласком по низу и временска сложеност му је $O(n)$. Меморијска сложеност је $O(1)$.

```
int n;
cin >> n;
int prethodni;
cin >> prethodni;
// ukupan broj rastucih serija
long long brojRastucih = 0;
// duzina tekuce rastuce serije
long long duzinaTekuceRastuce = 1;
for(int i = 1; i < n; i++) {
  int tekuci;
  cin >> tekuci;
  if (tekuci > prethodni)
    // tekuci element produzava tekucu rastucu seriju
    duzinaTekuceRastuce++;
  else {
    // tekuci element zapocinje novu rastucu seriju
    // dodajemo sve rastuce serije koje su podserije rastuce serije
    // koja se zavrsila sa prethodnim elementom
    brojRastucih += (duzinaTekuceRastuce - 1) * duzinaTekuceRastuce / 2;
    duzinaTekuceRastuce = 1;
  }
  prethodni = tekuci;
}
// dodajemo sve rastuce serije koje su podserije poslednje rastuce
// serije
brojRastucih += (duzinaTekuceRastuce - 1)*duzinaTekuceRastuce/2;
cout << brojRastucih << endl;
```

Види другачија решења овог задатка.

Задатак: Максимални збир сегмента

Напиши програм који одређује највећи збир неког сегмента (подниза узастопних елемената) датог низа.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 50\,000$), а затим n целих бројева између -10 и 10, сваки број у посебном реду.

Излаз: На стандардни излаз испиши тражени збир.

Пример

Улаз	*Излаз*
6 2 -3 4 -1 3 -2	6

Решење

Груба сила

Најдиректнији могући начин да се задатак реши је да се израчуна збир сваког сегмента. Збир сваког сегмента можемо израчунавати засебно (у петљи или библиотечком функцијом), међутим, ефикасније решење добијамо ако сегменте набрајамо редом (угнежђеним петљама, где спољашња петља набраја редом леве, а

унутрашња десне крајеве сегмената) и збир наредног сегмента израчунавамо инкрементално, на основу збира претходног сегмента. Та техника је објашњена у задатку Највећи збир префикса.

Анализа сложености. Ако збир сваког сегмента рачунамо независно, сабирањем његових елемената (било у петљи, било помоћу библиотечке функције), сложеност решења је $O(n^3)$. Ако збирове сегмената рачунамо инкрементално, добијамо алгоритам сложености $O(n^2)$. У оба случаја елементе учитавамо у низ и меморијска сложеност је $O(n)$.

```
int maksZbirSegmenta(const vector<int>& a) {
  int n = a.size();
  int max = 0;
  for (int i = 0; i < n; i++) {
    int z = 0;
    for (int j = i; j < n; j++) {
      z += a[j];
      if (z > max)
        max = z;
    }
  }
  return max;
}
```

Одсецање непотребних провера

Алгоритам заснован на провери свих сегмената се слободно може назвати тривијалним, јер се до њега долази прилично директно и веома једноставно му се и доказује коректност и анализира сложеност. Међутим, он је прилично неефикасан за решавање овог проблема, чак и када се збирови рачунају инкрементално. Значајно унапређење можемо добити када приметимо да велики број сегмената уопште не морамо да обрађујемо, јер из неких других разлога знамо да њихов збир не може бити максималан.

Посматрајмо низ `-2 3 2 -3 -3 4 -2 5 -8 3` и збирове свих његових непразних сегмената.

```
-2  1   3   0  -3   1  -1   4  -4  -1
    3   5   2  -1   3   1   6  -2   1
        2  -1  -4   0  -2   3  -5  -2
           -3  -6  -2  -4   1  -7  -4
               -3   1  -1   4  -4  -1
                    4   2   7  -1   2
                       -2   3  -5  -2
                            5  -3   0
                               -8  -5
                                    3
```

Прекид после негативног збира

Размотримо било који низ који почиње негативним бројем. Ниједан сегмент који почиње од тог броја, не може бити сегмент максималног збира, пошто се изостављањем првог броја добија већи збир. Ово својство је и општије. Уколико низ почиње префиксом негативног збира, из истог разлога, ниједан сегмент чији је он префикс не може бити сегмент максималног збира. Отуд, при инкременталном проширивању интервала удесно, чим се установи да је текући збир негативан, могуће је прекинути даље проширивање.

На пример, чим видимо да је први елемент првог сегмента `-2`, можемо прекинути даљу обраду елемената прве врсте, јер ће сви елементи друге врсте сигурно бити за два већи него одговарајући елементи прве врсте (`3` је веће од `1`, `5` је веће од `3`, `2` је веће од `0` итд.).

Слично, када се приликом проширивања сегмента који почиње на позицији `1` (од елемента `3`) дође до тога да је парцијални збир `-1` (што се дешава када се израчуна збир $3 + 2 - 3 - 3 = -1$, можемо прекинути са обрадом даљих сегмената који почињу на тој позицији, јер смо сигурни да ће за сваки од њих касније већи бити онај који се добија изостављање префикса `3 2 -3 -3`, чији је збир `-1`. Заиста од преосталих збирова `3 1 6 -2 1` у другој врсти за један су већи збирови `4 2 7 -1 2` у шестој врсти који су добијени изостављањем тог префикса. Обратимо пажњу на то да прекид унутрашње петље на овај начин узрокује да се максимална вредност у текућој врсти не мора уопште наћи. Петља која обрађује другу врсту ће бити прекинута чим се наиђе на збир `-1`, када је текућа вредност максимума `5` иако је максимум те врсте `6`. Сигурни смо да ће у некој

наредној врсти постојати већа вредност од те највеће (заиста, у шестој врсти се јавља 7), па нам налажење стварног максимума у текућој врсти уопште није неопходно.

Анализа сложености. Иако се на овај начин може прескочити разматрање неких сегмената, у најгорем случају сложеност није смањена. На пример, у случају да су елементи низа строго позитивни, збир никад не постаје негативан и сложеност након овог исецања је и даље квадратне сложености тј. $O(n^2)$.

Одсецање провере почетака унутар позитивног сегмента

Ако су сви елементи позитивни, максималан збир бива нађен за $i = 0$ и $j = n - 1$. Након тога се, увећавањем индекса i, збир смањује пошто се сваким скраћивањем сегмента слева изоставља неки позитиван број који доприноси збиру. И ово запажање се може уопштити. Не само што је непожељно скратити интервал слева за неки позитиван број, већ је непожељно скратити га за било који префикс чији је збир позитиван. Питање је докле такви префикси сежу? Бар до елемента чијим обухватањем добијамо први негативан префикс. Отуд сегмент максималног збира не може почињати ни на једној позицији између текуће почетне позиције и прве позиције на којој збир постаје негативан.

На пример, у наведеном примеру максимални сегмент не може почињати на позицији `2`, јер се проширивањем налево и додавањем елемента `3` са позиције `1` добијају сигурно збирови који су већи за три. Дакле, сви елементи друге врсте (која одговара позицији `1` у низу) су за `3` већи од одговарајућих елемената треће врсте (која одговара позицији `2` у низу). Заиста, `5` је веће од `2`, `2` од `-1` итд. Слично, ти елементи су за `5` већи од одговарајућих елемената четврте врсте (која одговара позицији `3` у низу). Заиста, `2` је веће од `-3`, `-1` од `-6` итд. Они су за `2` већи од одговарајући елемената пете врсте (која одговара позицији `4` у низу). Заиста, `-1` је веће од `-3`, `-3` је веће од `-5` итд. Зато те три врсте уопште нема потребе разматрати.

Захваљујући овом запажању, при завршетку обраде једне врсте и преласку на наредну, није неопходно увећавати променљиву i за један, већ је могуће наставити иза елемента чијим је укључивањем збир постао негативан.

Пошто се сваки елемент обрађује само једном, приликом имплементације није неопходно све елементе памтити у низу.

Пример. Прикажимо рад алгоритма на примеру низа `-2 3 2 -3 -3 -2 4 5 -8 3`. У таблици попуњавамо вредности променљивих током обраде елемената низа.

```
i    j   max    z     a_j
0         0
     0          0
     0         -2    -2
1
     1          0
     1    3     3     3
     2    5     5     2
     3    5     2    -3
     4         -1    -3
5
     5          0
     5         -2    -2
6
     6          0
     6    5     4     4
     7    9     9     5
     8    9     1    -8
     9    9     4     3
    10
11
```

Анализа сложености. Пошто обе променљиве пролазе кроз распон од 0 до n и крећу се само у једном смеру (вредност им се само повећава и никада не смањује), сложеност овог решења је линеарна тј. $O(n)$. У приказаној имплементацији елементи се чувају у низу па је и меморијска сложеност линеарна тј. $O(n)$, међутим, пошто се сваки елемент анализира само једном, за тим нема потребе и могуће је направити и имплементацију константне меморијске сложености.

```
int maksZbirSegmenta(const vector<int>& a) {
  int n = a.size();
  int max = 0;
  int i = 0;
  while (i < n) {
    int z = 0;
    int j;
    for (j = i; j < n; j++) {
      z += a[j];
      if (z < 0)
        break;
      if (z > max)
        max = z;
    }
    i = j + 1;
  }
  return max;
}
```

Види другачија решења овог задатка.

2.12 Инкременталност

Једна од основних техника за избегавање лоше сложености алгоритама је да се избегне израчунавање истих ствари више пута у истом програму. Често је потребно израчунати неку вредност за различите вредности неког параметра. Рећи ћемо да је израчунавање **инкрементално** ако се резултујућа вредност за наредну вредност параметра израчунава коришћењем већ израчунатих вредности за претходну (или неколико претходних) вредности параметра. Дакле, на мале измене улазних података реагујемо малим изменама резултата (уместо да поново вршимо велико израчунавање свега, из почетка).

Веома једноставан пример принципа инкременталности је израчунавање парцијалних збирова (збирови префикса) елемената неког низа. На пример, ако је дат низ $1, 2, 3, 4, 5$, његови парцијални збирови су редом $0, 1, 3, 6, 10, 15$. Веома једноставно се примећује да се израчунавање наредног парцијалног збира не мора вршити сабирањем свих елемената од почетка, већ се може добити сабирањем претходног парцијалног збира са текућим елементом низа (на пример, збир $1 + 2 + 3 + 4 = 10$, се добија сабирањем претходног збира $1 + 2 + 3 = 6$ и текућег елемента 4). Ако збир првих k елемената означимо са Z_k, тада важи да је $Z_0 = 0$ и да је $Z_{k+1} = Z_k + a_k$. Овим смо добили серију бројева у којој се наредни елемент израчунава на основу претходног (или неколико претходних). За такве серије кажемо да су *рекурентне серије*. Сваки наредни члан се израчунава у сложености $O(1)$, па се израчунавање свих парцијалних збирова низа дужине n врши у сложености $O(n)$. Када би се сваки парцијални збир рачунао сабирањем елемената низа из почетка, тада би израчунавање k-тог збира било сложености $O(k)$, а израчунавање свих збирова сложености $O(n^2)$.

Принцип инкременталности је у тесној вези са индуктивно/рекурзивном конструкцијом алгоритама и лежи у основи великог броја основних алгоритама. Већ само израчунавање збира свих елемената низа заправо почива на постепеном, инкременталном израчунавању збирова префикса, све док се не израчуна збир свих елемената низа. Слично је и са израчунавањем минумума, максимума, линеарном претрагом и другим фундаменталним алгоритмима. У свим овим примерима крећемо од неке почетне вредности у низу резултата, а затим наредну вредност у том низу израчунавамо на основу претходне или неколико претходних, што директно одговара индуктивном поступку израчунавања. Слична техника (добијања наредних резултата на основу претходних) примењује се у склопу технике динамичког програмирања навише, о чему ће више речи бити касније.

Поред парцијалних збирова, инкрементално се могу израчунавати и парцијални производи, парцијални минимуми и максимуми и слично.

2.12.1 Инкременталност збира и производа

Једна од најчешће коришћених статистика је збир елемената неког низа. Парцијални збирови серије елемената се могу рачунати инкрементално, што често убрзава алгоритам. Веома сличан збиру је и производ, и парцијалне производе је такође могуће рачунати инкрементално.

Задатак: Највећи збир префикса

Сваког дана током неког периода на банковни рачун је вршена тачно једна трансакција (уплата или исплата новца). Ако је почетно стање на рачуну нула, напиши програм који одређује највеће стање на рачуну током тог периода.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 100000$), а затим и n целих бројева (сваки у посебној линији) који представљају трансакције (позитиван број означава уплату, а негативан исплату).

Излаз: На стандардни излаз исписати један цео број који представља највеће стање на рачуну у неком тренутку.

Пример

Улаз	*Излаз*
5 4 2 -3 5 -4	8

Решење

Засебно израчунавање сваког салда

Под претпоставком да су подаци о свим трансакцијама учитане у низ, директно, наивно решење овог задатка би било да се за сваки дан k (од 0 до $n-1$) одреди збир Z_k елемената од првог дана закључно са даном k (алгоритмом сабирања серије бројева) и да се онда одреди највећи од тих збирова (алгоритмом одређивања минимума тј. максимума описаним).

```
int najveciZbirPrefiksa(const vector<int>& a) {
  // prvog dana je zbir 0
  // najveci zbir prefiksa
  int maxZbir = 0;
  for (int i = 0; i < a.size(); i++) {
    // racunamo zbir prefiksa od pocetka do pozicije i (ukljucujuci i nju)
    int zbir = 0;
    for (int j = 0; j <= i; j++)
      zbir += a[j];
    // azuriramo maksimum ako je potrebno
    if (zbir >= maxZbir)
      maxZbir = zbir;
  }
  return maxZbir;
}
```

Анализа сложености. Пошто се збир префикса дужине k може израчунати у k корака, укупно бисмо извршавали око $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$ сабирања (и линеарни број ажурирања максимума) и временска сложеност овог наивног алгоритма била би квадратна у односу на број елемената низа тј. $O(n^2)$.

Скривена сложеност

За сабирање је могуће употребити и библиотечку функцију. У језику C++ то је функција `accumulate` која сабира све елементе у делу низа ограниченом са два итератора, при чему се вредност збира у почетку иницијализује на вредност прослеђену као трећи параметар те функције.

```
int najveciZbirPrefiksa(const vector<int>& a) {
  // prvog dana je zbir 0
  // najveci zbir prefiksa
  int maxZbir = 0;
  // analiziramo sve pozicije u nizu
  for (auto it = begin(a); it != end(a); it++)
    // racunamo zbir prefiksa od pocetka do ispred pozicije it
    // i azuriramo maksimum ako je potrebno
```

```
    maxZbir = max(maxZbir, accumulate(begin(a), it, 0));
  return maxZbir;
}
```

Анализа сложености. Иако се тада у програму види једна петља, због скривене линеарне сложености библиотечких функција укупна сложеност програма остаје квадратна тј. $O(n^2)$.

Напомена. Скривена сложеност у задацима овог типа је нарочито чест проблем у језицима у којима се збир дела низа израчунава јако једноставно и елегантно. На пример, у језику Python:

```
maxZbir = 0
for i in range(n):
    maxZbir = max(maxZbir, sum(a[0:i+1]))
```

Инкрементално израчунавање салда

Много ефикасније решење се може добити ако се примети прилично очигледна чињеница да се салдо у следећем дану може јако једноставно израчунати ако је познат салдо у претходном дану тако што се тај салдо увећа за износ трансакције у следећем дану, што омогућава *инкременталност* израчунавања. Збир (текући салдо) израчунавамо индуктивном конструкцијом. Означимо са Z_k збир трансакција у првих k дана. Важи да је $Z_0 = 0$ и да је $Z_{k+1} = Z_k + a_k$. Збир зато иницијализујемо на нулу, а онда у петљи учитавамо трансакцију по трансакцију, увећавамо тренутни збир за износ трансакције, проверавамо да ли је текући износ већи од до тада највећег и ако јесте, ажурирамо вредност највећег износа.

```
// ucitavamo broj elemenata niza
int n;
cin >> n;

// prvog dana je saldo 0
// najveci zbir prefiksa
int maxZbir = 0;
// tekuci zbir prefiksa
int zbir = 0;
for (int i = 0; i < n; i++) {
  // ucitavamo tekuci element niza
  int x; cin >> x;
  // azuriramo zbir prefiksa
  zbir += x;
  // azuriramo maksimum ako je potrebno
  if (zbir >= maxZbir)
    maxZbir = zbir;
}

// ispisujemo resenje
cout << maxZbir << endl;
```

Анализа сложености. У сваком кораку петље вршимо константан број додатних операција (ажурирање збира и минимума) и сложеност овог алгоритма је линеарна тј. $O(n)$. Приметимо и да је меморијска сложеност овог решења $O(1)$, јер током инкременталне обраде нема потребе памтити вредности свих трансакција у низу.

Задатак: Сегмент датог збира у низу целих бројева

Напиши програм који за дати низ целих бројева одређује број непразних сегмената узастопних елемената низа чији је збир једнак датом броју.

Улаз: Са стандардног улаза се у првој линији уноси тражена вредност збира z (цео број -10000 и 10000), затим, у наредној линији димензија низа n ($3 \leq n \leq 50000$) и затим у наредној линији елементи низа (цели бројеви између -100 и 100, раздвојени размаком).

Излаз: На стандардни излаз испиши број сегмената чији је збир једнак z.

Пример

Улаз	*Излаз*
11 10 1 2 3 5 1 -1 1 5 3 2	7

Решење

Груба сила

Директно решење грубом силом подразумевало би да се провере сви сегменти узастопних елемената, да се за сваки израчуна збир и да се провери да ли је тај збир једнак траженом. Сви сегменти се могу набројати угнежђеним петљама, где спољна петља пролази кроз леве крајеве сегмента (i узима вредности од 0 па до $n-1$), а унутрашња петља пролази кроз десне крајеве сегмената (од вредности i па до $n-1$). Збир можемо рачунати алгоритмом сабирања или применом библиотечких функција.

```
int brojSegmenataDatogZbira(const vector<int>& a, int trazeniZbir) {
  // broj elemenata niza
  int n = a.size();
  // broj segmenata trazenog zbira
  int broj = 0;

  // prolazimo sve segmente
  for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
      // izracunavamo zbir segmenta [i, j]
      int zbir = 0;
      for (int k = i; k <= j; k++)
        zbir += a[k];
      // proveravamo da li je zbir tog segmenta jednak trazenom
      if (zbir == trazeniZbir)
        broj++;
    }
  }

  return broj;
}
```

Анализа сложености. Сложеност овог приступа је кубна у односу на димензију n тј. $O(n^3)$. Наиме, постоји квадратни број сегмената и за сабирање елемената сваког од њих потребно је линеарно време.

Претходна процена је извршена прилично грубо, јер иако је сложеност сабирања сегмента линеарна, нису сви сегменти дужине n, међутим, и прецизнија анализа ће довести до истог резултата. Унутрашња петља се извршава $j-i+1$ пута толико пута се врши операција сабирања), што одговара дужини сегмента. Спољне петље које набрајају све сегменте набрајају један сегмент дужине n, два сегмента дужине $n-1$, три сегмента дужине $n-2$, … и n сегмената дужине 1. Значи да је број корака једнак $1 \cdot n + 2 \cdot (n-1) + 3 \cdot (n-2) + \ldots + (n-1) \cdot 2 + n \cdot 1$. Дакле, сегмената дужине k има $n-k+1$, па важи да је претходни збир једнак

$$\sum_{k=1}^{n} k(n-k+1) = (n+1) \cdot \sum_{k=1}^{n} k - \sum_{k=1}^{n} k^2 = (n+1) \cdot \frac{n(n+1)}{2} - \frac{n(n+1)(2n+1)}{6}$$

Ово је реда величине $\frac{n^3}{2} - \frac{2n^3}{6} = \frac{n^3}{6}$, што је $O(n^3)$.

Инкрементално рачунање збира сегмената

Алгоритам грубе силе који посебно рачуна збир сваког сегмента се може унапредити ако се збирови рачунају инкрементално тј. ако се искористи чињеница да се збир сваког сегмента који се добија проширивањем претходног сегмента једним елементом може лако израчунати на основу збира претходног сегмента, тако што се збир претходног сегмента увећа за текући елемент низа. Ту идеју смо видели, на пример, у задатку Највећи збир префикса.

Дакле, опет можемо набрајати све сегменте угнежђеним петљама, на почетку тела спољашње петље збир иницијализујемо на нулу, у унутрашњој петљи збир увећавамо за елемент a_j и, ако је он једнак траженом, исписујемо индексе интервала $[i, j]$.

```
int brojSegmenataDatogZbira(const vector<int>& a, int trazeniZbir) {
  // broj elemenata niza
  int n = a.size();
  // broj segmenata trazenog zbira
  int broj = 0;

  // prolazimo sve segmente
  for (int i = 0; i < n; i++) {
    // zbir segmenta [i, j]
    int zbir = 0;
    for (int j = i; j < n; j++) {
      // izracunavamo zbir segmenta [i, j] na osnovu zbira segmenta [i, j-1]
      zbir += a[j];
      // proveravamo da li je zbir tog segmenta jednak trazenom
      if (zbir == trazeniZbir)
        broj++;
    }
  }

  return broj;
}
```

Анализа сложености. Овим је избегнута линеарна сложеност за израчунавање збира тренутног интервала тј. да се збир сваког наредног интервала добија у константном времену, тако да је сложеност целог алгоритма редукована на квадратну тј. $O(n^2)$.

Опет би се прецизнијом анализом добио исти резултат. Наиме, сви збирови сегмената који почињу на позицији 0 рачунају се помоћу n сабирања, збирови сегмената који почињу на позицији 1 рачунају се помоћу $n-1$ сабирања итд. Укупан број сабирања је, дакле, $1 + \ldots + n = \frac{n(n+1)}{2}$, што је $O(n^2)$.

Задатак се, може решити и доста ефикасније од овога.

Види другачија решења овог задатка.

Задатак: Сума реда

Другарице су кренуле на клизање. Изнајмиле су клизаљке, мало се клизале, а онда одлучиле да се окрепе уз топли чај. Изуле су се, ставиле клизаљке на гомилу, али су заборавиле да обележе које клизаљке су чије. Пошто све имају ногу веома сличне величине, договориле су се да није ни важно, него да свака може да узме било који пар клизаљки са гомиле. Израчунати колика је вероватноћа да ниједна од њих није добила исте клизаљке које је користила пре паузе за чај, ако се зна да се та вероватноћа може израчунати као $1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} \ldots + \frac{(-1)^n}{n!}$, где је n број другарица.

Улаз: Са стандардног улаза се учитава број другарица n ($2 \leq n \leq 20$).

Излаз: На стандардни излаз исписати тражену вероватноћу, заокружену на 14 децимала.

Пример 1

Улаз	*Излаз*
2	0.50000000000000

Пример 2

Улаз	*Излаз*
10	0.367879464285714

Решење

Израчунавање сваког сабирка засебно

Директан начин да се израчуна тражена вероватноћа је да се израчуна збир серије бројева која се добија тако што се за свако k од 0 до n израчуна вредност $\frac{(-1)^k}{k!}$. Степен $(-1)^k$ се може израчунати функцијом `pow` или се може одредити гранањем, пошто је $(-1)^k = 1$ за парне вредности k и $(-1)^k = -1$ за непарне вредности k. Факторијел $k!$ се може израчунати множењем серије бројева од 1 до k.

```
// faktorijel broja n
double faktorijel(int n) {
  double p = 1.0;
  for (int i = 2; i <= n; i++)
    p *= i;
  return p;
}

// verovatnoca da nijedna devojcica nije uzela iste klizaljke:
// zbir 1 - 1/1! + 1/2! + ... + (-1)^n/n!
double verovatnoca(int n) {
  double p = 0.0;
  for (int k = 0; k <= n; k++)
    p += pow(-1, k) / faktorijel(k);
  return p;
}
```

Анализа сложености. Израчунавање k-тог сабирка захтева $O(k)$ операција, па је за сабирање n сабирака потребно време $O(n^2)$.

Инкрементално израчунавање сабирака

Ефикасније решење се може добити ако се уочи да бројеви $1 = \frac{(-1)^0}{0!}$, $-1 = \frac{(-1)^1}{1!}$, $\frac{1}{2} = \frac{(-1)^2}{2!}$, $-\frac{1}{6} = \frac{(-1)^3}{3!}$ итд. чине веома правилну серију у којој се сваки наредни члан може добити инкрементално, множењем претходног члана вредношћу $-\frac{1}{k}$. Дакле, у овом задатку се користи како инкременталност серије парцијалних збирова (у склопу алгоритма сабирања), тако и инкременталност серије самих сабирака, који су заправо парцијални производи правилне серије $-\frac{1}{1}, -\frac{1}{2}, -\frac{1}{3}, -\frac{1}{4}, \ldots$ Ако са x_k обележимо сабирак k, тада је $x_0 = \frac{(-1)^0}{0!} = 1$, док је $x_{k+1} = -\frac{1}{k} \cdot x_k$.

Током имплементације ћемо одржавати две променљиве. Прва ће да представља збир до сада сабраних сабирака, а друга текући сабирак. Збир и текући члан иницијализоваћемо на вредност 1 (то је вредност $\frac{(-1)^0}{0!}$). У сваком кораку петље у којој k узима вредности од 1 до n текући члан ћемо ажурирати множењем са вредношћу $-\frac{1}{k}$ и додаваћемо га на збир. Након завршетка петље, збир ће садржати тражену вероватноћу коју ћемо исписати са траженим бројем децимала.

Напомена. Добар савет приликом израчунавања збирова овог типа је да се израчуна количник два суседна сабирка и да се провери да ли се он можда веома једноставно израчунава у функцији од k (у овом случају тај количник је $\frac{-1}{k}$). Уместо количника, могуће је користити и разлику два узастопна сабирка.

```
// verovatnoca da nijedna devojcica nije uzela iste klizaljke:
// zbir 1 - 1/1! + 1/2! + ... + (-1)^n/n!
double verovatnoca(int n) {
  double p = 1.0;
  // tekuci element zbira (-1)^k/k!
  double xk = 1.0;
  for (int k = 1; k <= n; k++) {
    // izracunavamo sledeci clan mnozenjem prethodnog sa -1/k
    xk *= -1.0/k;
    // dodajemo ga na z
    p += xk;
  }
  return p;
}
```

Анализа сложености. Израчунавање k-тог сабирка на основу претходног захтева само $O(1)$ операција, па се израчунавање збира n сабирака врши у времену $O(n)$. Додуше, у овом задатку n је веома мали број, па се на овај начин не постиже значајно убрзање, али у другим сличним задацима оптимизација на основу инременталности може бити веома значајна.

Доказ коректности. Ко жели да зна више?

На крају, образложимо и како се дошло до формуле која је у задатку дата. Пермутације у којима ни један

елемент није на свом месту се називају деранжмани. Њихов број је могуће одредити на основу формуле укључивања-искључивања. Размотримо скуп пермутација од 4 елемента. Њега чине пермутације 1234, 1243, 1324, 1342, 1423, 1432, 2134, 2143, 2314, 2341, 2413, 2431, 3124, 3142, 3214, 3241, 3412, 3421, 4123, 4132, 4213, 4231, 4312, 4231 од чега су деранжмани 2143, 2341, 2413, 3142, 3412, 3421, 4123, 4312 и 4321. Укупно пермутација има $4!$. Од тог броја треба одузети број пермутација којима се бар један елемент налази на свом месту. Можемо разматрати пермутације у којима се 1 налази на свом месту, а остали бројеви су произвољно испермутовани (то су 1234, 1243, 1324, 1342, 1423, 1432), у којима се број 2 налази на свом месту, а остали бројеви су произвољно испермутовани (то су 1234, 1243, 3214, 3241, 4213, 4231), у којима се број 3 налази на свом месту, а остали бројеви су произвољно испермутовани (то су 1234, 1432, 2134, 2431, 4132, 4231) и оне у којима је 4 на свом месту, а остали бројеви су произвољно испермутовани (то су 1234, 1324, 2134, 2314, 3124, 2314). Њих има $4 \cdot 3!$. Преостале пермутације су деранжмани и број деранжмана се може добити тако што се од укупног броја пермутација одузме број оних пермутација које фиксирају неки елемент. Проблем настаје због тога што су неке пермутације бројане више пута (на пример, пермутација 1243 је бројана и у оквиру пермутација које елемент 1 остављају на свом месту и у оквиру пермутација које остављају број 2 на свом месту). Пермутације које су бројане бар два пута су оне које остављају нека два елемента на свом месту. Ако су то елементи 1 и 2, то су пермутације 1234 и 1243, ако су то елементи 1 и 3 то су пермутације 1234 и 1432, ако су то елементи 1 и 4 то су пермутације 1234 и 1324, ако су то елементи 2 и 3 то су пермутације 1234 и 4231, ако су то елементи 2 и 4 то су пермутације 1234 и ако су то елементи 3 и 4 то су пермутације 1234 и 2134. Парова елемената има $\binom{4}{2} = 6$, а сваки од њих даје две пермутације. Њихов број можемо одузети од броја пермутација са фиксираним једним елементом, али је проблем да се међу њима неке пермутације понављају тако да смо онда одузели више него што је потребно. Пермутације које се понављају више пута су оне које фиксирају 3 елемента. Која год да су три елемента у питању то је пермутација 1234. Три елемента можемо одабрати на $\binom{4}{3} = 4$ начина тако да треба додати 4 пермутације. Међутим, тада је пермутација која фиксира сва 4 елемента урачуната два пута, тако да њу на крају треба одузети. Дакле број деранжмана једнак је $4! - 4 \cdot 3! + 6 \cdot 2! - 4 \cdot 1! + 1 = 24 - 24 + 12 - 4 + 1 = 9$.

Овом логиком долазимо до тога да се број деранжмана може израчунати као

$$n! - \binom{n}{1}(n-1)! + \binom{n}{2}(n-2)! - \ldots + (-1)^{n-1}\binom{n}{n-1}1! + (-1)^n$$

Зато је вероватноћа да се насумично одабере деранжман једнака количнику броја деранжмана и укупног броја пермутација. Пошто је $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, добијамо

$$\frac{n! - \frac{n!}{1!(n-1)!}(n-1)! + \frac{n!}{2!(n-2)!}(n-2)! + \ldots + (-1)^{n-1}\frac{n!}{(n-1)!1!}1! + (-1)^n}{n!}$$

тј.

$1 - \frac{1}{1!} + \frac{1}{2!} - \ldots + \frac{(-1)^{n-1}}{(n-1)!} + \frac{(-1)^n}{n!}$

2.12.2 Инкременталност минимума и максимума

И минимум и максимум серије бројева представљају веома важне и често коришћене статистике. Минимум тј. максимум серије префикса тј. серије суфикса низа такође могу да се израчунавају инкрементално, што доводи до ефикаснијих решења.

Задатак: Најбољи "сабмит"

Такмичар је током интергалактичког шампионата у програмирању слао на оцењивање један задатак више пута. Број поена које такмичар добија за задатак се рачуна тако што се одреди највећи број поена од свих појединачних слања (гледа се "најбољи сабмит"). Напиши програм који одређује колико је поена за тај задатак ученик имао након сваког слања. Пошто су интергалактички задаци веома тешки, они носе пуно поена и такмичари их често шаљу велики број пута.

Улаз: У првој линији стандардног улаза налази се природан број n ($n \leq 50000$). У следећих n линија налазе се редом поени које је ученик добио за свако појединачно слање (број између 0 и 100000).

Излаз: На стандардном излазу приказати n линија које приказују колико је поена за тај задатак ученик имао након сваког слања.

Пример

Улаз	*Излаз*
5	3
3	3
2	4
4	4
1	5
5	

Решење

За сваку позицију i у серији бројева је потребно одредити највећи међу свим бројевима од почетка серије, закључно са тренутним елементом $m_i = \max(a_0, \ldots, a_i)$.

Груба сила

Наивни начин да се то уради је да се резултати свих слања упишу у низ, а затим да се за сваку позицију i од 0 до $n-1$ одреди и испише максимум m_i. Одређивање максимума се може урадити уобичајеним алгоритмом одређивања максимума серије бројева.

Анализа сложености. Ово решење је прилично неефикасно (има квадратну сложеност у односу на број слања, тј. сложеност $O(n^2)$).

```
int n;
cin >> n;
vector<int> poeni(n);
for (int i = 0; i < n; i++)
  cin >> poeni[i];

for (int i = 0; i < n; i++) {
  int maxPoena = poeni[0];
  for (int j = 1; j <= i; j++)
    if (poeni[j] > maxPoena)
      maxPoena = poeni[j];

  cout << maxPoena << endl;
}
```

Библиотечке функције - скривена сложеност

Максимум дела низа можемо одредити и библиотечком функцијом. У језику C++ можемо употребити функцију `max_element`.

Анализа сложености. Иако сада програм има само једну петљу, у функцији за израчунавање максимума је скривена линеарна сложеност, па укупна је сложеност $O(n^2)$.

```
for (auto it = next(begin(poeni)); it <= end(poeni); it++)
  cout << *max_element(begin(poeni), it) << endl;
```

Инкременталност серије парцијалних максимума

Кључни увид којим се може доћи до ефикаснијег решења је да се приликом додавања новог елемента максимум проширене серије m_{i+1} може израчунати крајње једноставно ако је познат максимум m_i серије пре проширења. Наиме, ако је познат број поена такмичара пре текућег слања задатка, када се одреди број поена у том слању, максимум се или увећава (ако је у текућем слању остварен највећи број поена до тада) или се не мења тј. $m_{i+1} = \max(m_i, a_{i+1})$. Почетни максимум може бити постављен или на први елемент низа (важи да је $m_0 = a_0$), или, пошто су сви елементи ненегативни, на нулу (важи да је $m_{-1} = 0$). Дакле, принцип инкременталности који важи за парцијалне збирове, важи и за парцијалне максимуме. Инкрементално израчунавање серије парцијалних збирова (збирова префикса) описано је, на пример, у задатку Највећи збир префикса.

Дакле, у задатку се примењује класичан алгоритам одређивања максимума серије бројева, али се у сваком кораку исписује текућа вредност максимума.

Анализа сложености. Иницијализација се врши у времену $O(1)$ и сваки нови максимум се од претходног добија у времену $O(1)$, па се n максимума израчунава у укупном времену $O(n)$. Ово решење не учитава

истовремено све елементе у низ, па је меморијска сложеност $O(1)$. Са друге стране у овом решењу се наизменично учитавају и исписују бројеви, што у језику C++ захтева да се библиотека за улаз и излаз посебно подеси, да би се искључило стално пражњење бафера, које успорава програм (довољно је навести `cin.tie(0)` и уместо `endl` користити `'\n'`.

```
ios_base::sync_with_stdio(false); cin.tie(0);
int n;
cin >> n;
int maxPoena = 0;
for (int i = 0; i < n; i++) {
  int poeni;
  cin >> poeni;
  if (poeni > maxPoena)
    maxPoena = poeni;
  cout << maxPoena << '\n';
}
```

Задатак: Поглед на реку

У једној улици која иде ка реци налазе се разне куће и зграде. Инвеститор бира локацију на којој би изградио вишеспратницу, такву да се са њеног последњег спрата види река. Зато она мора бити виша или бар једнаке висине од свих постојећих зграда од одабране локације до краја улице. Напиши програм који за сваку локацију у тој улици одређује минималну висину нове вишеспратнице.

Улаз: У првој линији стандардног улаза налази се природан број n ($n \leq 50000$). У следећих n линија налазе се редом висине свих зграда и кућа од почетка до краја улице.

Излаз: На стандардном излазу приказати n бројева (сваки у посебном реду) који приказују тражене висине.

Пример

Улаз	*Излаз*
5	23
13	23
23	17
11	17
17	13
13	

Решење

Задатак захтева да се за сваку позицију i у низу одреди максимум свих елемената од те позиције па до краја низа (тј. да се одреди $m_i = \max(a_i, \ldots, a_{n-1})$).

Приметимо да се овде захтева рачунање максимума серије суфикса низа, што је заправо веома слично израчунавању максимуми серије префикса низа. Израчунавање серије максимума префикса низа описано је у задатку Најбољи “сабмит”.

Груба сила

Директан, наивни приступ да се то уради је да се у спољној петљи по i пролази кроз позиције од 0 до $n-1$, а да се у унутрашњој петљи која пролази позиције од i до $n-1$ одређује максимум m_i (применом уобичајеног алгоритма одређивања максимума тј. минимума серије елемената). Максимуме ћемо уписивати у низ (то може бити и оригинални низ) и на крају исписати у траженом редоследу.

Анализа сложености. Пошто се за сваки суфикс дужине од 1 до n одређује максимум у линеарној сложености, укупна сложеност овог приступа је $O(n^2)$.

Инкременталност

И за серију максимума растућих суфикса низа важи принцип инкременталности. Максимум елемената који почињу на позицији i може се лако одредити ако се већ зна максимум елемената који почињу на позицији $i+1$. Наиме, важи да је $m_i = \max(a_i, m_{i+1})$. Иницијализацију можемо извршити било на последњи елемент серије (важи $m_{n-1} = a_{n-1}$), било на нулу, јер су сви елементи ненегативни (важи $m_n = 0$).

Задатак решавамо тако што у петљи пролазимо кроз низ у обрнутом редоследу, крећући се од последњег члана низа (који је индексиран са $n-1$) до првог члана (који је индексиран са 0). Максимум иницијализујемо на нулу. У сваком пролазу кроз петљу, анализира се вредност текућег елемента низа који се пореди са текућим максимумом. Ако је вредност текућег елемента низа већа од текућег максимума (који у ствари представља максималану вредност елемената десно од текућег), нови текући максимум постаје баш тај члан низа. Један проблем са овим решењем је што се максимуми одређују у супротном редоследу од оног који је потребно исписати. Зато је пре исписа потребно упамтити их. Једно решење би било да се они памте у новом низу. Ако не желимо да ангажујемо додатну меморију, можемо приметити да се након одређивања максимума m_i елемент a_i неће више анализирати, па се m_i може уписати на место елемента a_i. На крају, исписаћемо елементе низа у који смо сместили максимуме од почетка.

```
// maksimalni element od pozicije i do kraja niza
int max = 0;
for (int i = n - 1; i >= 0; i--) {
  // ako je tekuci element veci od maksimalnog iza njega, onda je
  // potrebno azurirati maksimum
  if (a[i] > max)
    max = a[i];
  // belezimo vrednost maksimuma (koristimo slobodno prostor niza a,
  // jer nam vrednost a[i] vise nece biti potrebna)
  a[i] = max;
}
```

Анализа сложености. Пошто се сада сваки максимум одређује на основу претходног у константној сложености, укупна сложеност одређивања свих n максимума је линеарна тј. $O(n)$.

Напомена. Приметимо да и збирови суфикса имају својство инкременталности.

2.12.3 Инкременталност - остале статистике

Задатак: Рутер

Дуж једне улице су равномерно распоређене зграде (растојање између сваке две суседне је једнако). За сваку зграду је познат број корисника које нови добављач интернета треба да повеже. Одредити у коју од зграда треба поставити рутер тако да би укупна дужина оптичких каблова којим се сваки од корисника повезује са рутером била минимална (рачунати само дужину каблова од зграде до зграде и занемарити дужине унутар зграда).

Улаз: У првом реду стандардног улаза налази се број n ($1 \leq n \leq 10^5$), а у наредном n природних бројева раздвојених размацима који представљају број корисника у свакој од n зграда.

Излаз: На стандардни излаз исписати минималну дужину каблова.

Пример

Улаз	*Излаз*
6 3 5 1 6 2 4	30

Решење

Груба сила

Наивно решење би подразумевало да се израчуна дужина каблова за сваку могућу позицију рутера и да се одабере најмањи. Да бисмо израчунали дужину каблова, ако је рутер у згради на позицији k, рачунамо заправо збир

$$\sum_{i=0}^{n-1} |k-i| \cdot a_i,$$

где је a_i број корисника у згради i.

```
long long minDuzinaKablova(const vector<int>& broj_stanara) {
  // broj zgrada
  int n = broj_stanara.size();
```

```
  // minimalna duzina kablova
  long long min_duzina_kablova = numeric_limits<long long>::max();
  // obrađujemo sve zgrade od 1 do n-1
  for (int k = 0; k < n; k++) {
    // duzina kablova ako je ruter u zgradi broj k
    long long duzina_kablova = 0;
    for (int i = 0; i < k; i++)
      duzina_kablova += (k - i) * broj_stanara[i];
    for (int i = k+1; i < n; i++)
      duzina_kablova += (i - k) * broj_stanara[i];

    if (duzina_kablova < min_duzina_kablova)
      min_duzina_kablova = duzina_kablova;
  }

  return min_duzina_kablova;
}
```

Анализа сложености. Сваки тежински збир можемо израчунати у времену $O(n)$, па пошто се испитује n позиција, алгоритам је сложености $O(n^2)$.

Решење на основу принципа инкременталности

Много боље решење и линеарни алгоритам можемо добити ако применимо принцип инкременталности и избегнемо рачунање у сваком кораку из почетка. Размотримо како се дужина каблова мења када се рутер помера са зграде k на зграду $k+1$.

Дужину каблова за рутер у згради $k+1$ добијамо од дужине каблова за рутер у згради k тако што ту дужину увећамо за укупан број станара закључно са зградом k и умањимо је за укупан број станара почевши од зграде $k+1$. То је заправо интуитивно прилично јасно и без компликованог математичког извођења. Померањем рутера за дужину једне зграде надесно, сваком станару који живи закључно до зграде k дужина кабла се повећала за једно растојање између зграда, а свим станарима од зграде $k+1$ надесно се та дужина смањује за једно растојање између зграда.

Доказ коректности. Формално, математички, то се може показати на следећи начин. Ако је рутер на позицији k, тада је дужина каблова једнака

$$d_k = \sum_{i=0}^{k-1} (k-i) \cdot a_i + \sum_{i=k+1}^{n-1} (i-k) \cdot a_i.$$

Ако је рутер на позицији $k+1$, тада је дужина каблова једнака

$$d_{k+1} = \sum_{i=0}^{k} (k+1-i) \cdot a_i + \sum_{i=k+2}^{n-1} (i-k-1) \cdot a_i.$$

Разлика између те две суме једнака је

$$\begin{aligned}
d_{k+1} - d_k &= \left(\sum_{i=0}^{k} (k+1-i) \cdot a_i - \sum_{i=0}^{k-1} (k-i) \cdot a_i \right) + \left(\sum_{i=k+2}^{n-1} (i-k-1) \cdot a_i - \sum_{i=k+1}^{n-1} (i-k) \cdot a_i \right) \\
&= \left(\sum_{i=0}^{k-1} ((k+1-i) - (k-i)) \cdot a_i \right) + a_k - a_{k+1} + \left(\sum_{i=k+2}^{n-1} ((i-k-1) - (i-k)) \cdot a_i \right) \\
&= \sum_{i=0}^{k-1} a_i + a_k - a_{k-1} - \sum_{i=k+2}^{n-1} a_i \\
&= \sum_{i=0}^{k} a_i - \sum_{i=k+1}^{n-1} a_i
\end{aligned}$$

Укупне бројеве станара пре и после дате зграде можемо такође рачунати инкрементално (при преласку на наредну зграду, први број се увећава, а други умањује за број станара текуће зграде).

Дакле, у програму можемо да памтимо три ствари: дужину каблова d_k ако је рутер на позицији k, укупан број станара pre_k пре зграде k (не укључујући њу) и укупан број станара $posle_k$ од зграде k (укључујући њу) до краја. На почетку, када је $k = 0$, први број d_0 морамо експлицитно израчунати као $\sum_{i=1}^{n-1} i \cdot a_i$, други број треба иницијализовати на нулу $pre_0 = 0$, а трећи на укупан број свих станара $posle_k = \sum_{i=0}^{n-1} a_i$. Затим за свако k од 1 до $n-1$ рачунамо $pre_k = pre_{k-1} + a_{k-1}$, $posle_k = posle_{k-1} - a_{k-1}$ и затим $d_k = d_{k-1} + pre_k - posle_k$.

Пример. Илуструјмо извршавање овог алгоритма на примеру зграда у којима живи редом $3, 5, 1, 6, 2, 4$ станара.

```
k    dk  pre_k   posle_k
0    53   0       21
1    38   3       18
2    33   8       13
3    30   9       12
4    39  15        6
5    52  17        4
```

Приметимо да је могуће извршити и малу оптимизацију (додуше која неће поправити асимптотску сложеност) на основу монотоности низа d_k и петљу прекинути чим се број d_k први пут повећа. Наиме, вредности у том низу ће опадати до тражене минималне вредности, након чега ће кренути да расту. У претходном примеру, могли смо закључити да је минимална дужина каблова 30, чим се у наредном кораку та вредност повећала на 39.

```
long long minDuzinaKablova(const vector<int>& broj_stanara) {
  // broj zgrada
  int n = broj_stanara.size();
  // krećemo od zgrade 0
  // ukupna dužina kablova ako je ruter u tekućoj zgradi
  long long duzina_kablova = 0;
  for (int i = 0; i < n; i++)
    duzina_kablova += broj_stanara[i] * i;
  // broj stanara pre tekuće zgrade
  long long stanara_pre = 0;
  // broj stanara od tekuće zgrade do kraja
  long long stanara_posle = 0;
  for (int i = 0; i < n; i++)
    stanara_posle += broj_stanara[i];

  // minimalna duzina kablova
  long long min_duzina_kablova = duzina_kablova;

  // obrađujemo sve zgrade od 1 do n-1
  for (int k = 1; k < n; k++) {
    // ažuriramo brojeve stanara
    stanara_pre += broj_stanara[k-1];
    stanara_posle -= broj_stanara[k-1];
    // ažuriramo duz
    duzina_kablova += stanara_pre - stanara_posle;
    if (duzina_kablova < min_duzina_kablova)
      min_duzina_kablova = duzina_kablova;
  }

  return min_duzina_kablova;
}
```

Анализа сложености. Пошто је и за једну и за другу фазу потребно време $O(n)$, то је уједно сложеност овог алгоритма.

Интересантно, укупну почетну дужину каблова можемо израчунати ефикасно и без множења, тако што на збир додамо број станара у последњој згради, затим број станара у последње две зграде, затим број станара у последње три зграде и тако даље, све док не додамо број станара у свим зградама осим прве.

Задатак: Највећи тежински збир после цикличног померања

Дат је низ a целих бројева дужине n. Дозвољена је операција цикличног померања тј. ротације низа улево за једно место, операцију можемо понављати произвољан број пута. Написати програм којим се највећа вредност тежинског збира

$$0 \cdot a_0 + 1 \cdot a_1 + 2 \cdot a_2 + 3 \cdot a_3 + \ldots + (n-1) \cdot a_{n-1},$$

по модулу 1234567 у трансформисаном низу.

Улаз: У првој линији стандардног улаза налази се природан број n ($1 \leq n \leq 50000$). У следећих n линија налазе се редом елементи низа a (цели бројеви из интервала $[0, 100]$).

Излаз: На стандардном излазу у једној линији приказати највећу вредност тежинског збира.

Пример

Улаз	*Излаз*
3 5 4 1	13

Објашњење

Највећи збир се добија након ротације за два места улево (низ је тада 154).

Решење

Груба сила

Задатак можемо решити тако што $n-1$ пут низ ефективно ротирамо за по једно место улево, израчунавајући сваки пут тежински збир изнова, тражећи максимум и позицију максимума тако добијених тежинских збирова. Пошто се тежински збирови у нашем задатку рачунају по датом модулу mod, све операције у претходним формулама треба заменити одговарајућим операцијама по модулу (њих можемо реализовати у засебним функцијама).

Анализа сложености. Пошто број операција потребан за израчунавање тежинског збира линеарно зависи од дужине низа n (сваки тежински збир се израчунава у сложености $O(n)$), овај приступ доводи до квадратне сложености алгоритма (сложености $O(n^2)$). Чак иако бисмо оптимизовали број израчунавања остатака, програм ће бити неефикасан услед квадратне сложености алгоритма и пуно померања елемената низа.

```
// modul dat u tekstu zadatka
const int mod = 1234567;

// zbir brojeva x i y po modulu mod
int zm(int x, int y, int mod) {
  return (x % mod + y % mod) % mod;
}

// ciklicno pomeranje (rotiranje) niza za jedno mesto ulevo
void rotirajUlevo(vector<int>& a, int n) {
  int pom = a[0];
  for (int i = 0; i < n - 1; i++)
    a[i] = a[i + 1];
  a[n - 1] = pom;
  // moglo bi i rotate(begin(a), next(begin(a)), end(a));
}

// suma brojeva a[i]*i, za i od 0 do n-1, po modulu mod
```

```
int tezinskiZbir(const vector<int>& a, int n, int mod) {
  int s = 0;
  for (int i = 0; i < n; i++)
    s = zm(s, i * a[i], mod);
  return s;
}

// najveci tezinski zbir 0*a[0] + ... + (n-1)*a[n-1] posle ciklicnog
// pomeranja niza ulevo
int maksTezinskiZbirNakonRotacije(const vector<int>& a) {
  // kopiramo niz da bismo mogli da ga menjamo
  auto acopy = a;
  // broj elemenata niza
  int n = acopy.size();
  // maksimum inicijalizujemo na tezinsku sumu pocetnog niza
  int maxTezinskiZbir = tezinskiZbir(acopy, n, mod);
  for (int i = 1; i < n; i++) {
    // rotiramo elemente niza a za jedno mesto ulevo
    rotirajUlevo(acopy, n);
    // izracunavamo novu tezinsku sumu
    int tekSuma = tezinskiZbir(acopy, n, mod);
    // azuriramo podatke o maksumumu, ako je to potrebno
    if (tekSuma > maxTezinskiZbir)
      maxTezinskiZbir = tekSuma;
  }
  return maxTezinskiZbir;
}
```

Избегавање ротација низа

Уместо ефективне ротације свих елемената низа, ефекат обиласка низа који је ротиран за k места улево можемо постићи тако што обилазак крећемо од позиције k, а затим у петљи која има n итерација увећавамо бројач за 1, али овај пут по модулу n (када бројач постане n вредност му се враћа на нулу што можемо постићи било експлицитним испитивањем вредности након сваког увећања бројача, било израчунавањем остатка при дељењу са n, што је спорије од гранања, јер је израчунавање остатка при дељењу обично релативно скупа операција).

На основу ограничења датих у тексту задатка, максимална вредност тежинског збира се постиже када низ има 50000 елемената чија је вредност 100. Тада се са 100 множи сваки елемент од 0 до 49999 и максимални тежински збир је једнак 100 пута вредност збира свих природних бројева до 49999 што је једанко $100 \cdot \frac{49999 \cdot (49999+1)}{2}$, што је око $1.25 \cdot 10^{11}$ и стаје у опсег 64-битног типа (у језику C++ то је тип `long long`). Ако тај тип употребимо за чување збира, онда остатак при дељењу можемо израчунати само једном, након целокупног израчунавања збира, чиме се добија на ефикасности.

Анализа сложености. Иако избегавамо ротације, свака од n тежинских сума се израчунава засебно у времену $O(n)$, па сложеност остаје $O(n^2)$.

```
// modul dat u tekstu zadatka
const int mod = 1234567;

// najveci tezinski zbir 0*a[0] + ... + (n-1)*a[n-1] posle ciklicnog
// pomeranja niza ulevo
int maksTezinskiZbirNakonRotacije(const vector<int>& a) {
  int n = a.size();
  long long maxTezinskiZbir = 0;
  for (int i = 0; i < n; i++) {
    // koristimo tip long long, da ne bismo morali da izracunavamo
    // ostatak posle svakog sabiranja, vec samo nakon izracunavanja
    // celog zbira
    long long tezinskiZbir = 0;
    int k = i;
```

```
    for (int j = 0; j < n; j++) {
      tezinskiZbir += a[k] * j;
      if (++k == n) k = 0;
    }
    tezinskiZbir %= mod;
    // azuriramo podatke o maksumumu, ako je to potrebno
    if (tezinskiZbir > maxTezinskiZbir)
      maxTezinskiZbir = tezinskiZbir;
  }
  return maxTezinskiZbir;
}
```

Оптимизација на основу инкременталности

Задатак је могуће решити и много ефикасније ако применимо принцип инкременталности тј. пронађемо начин да тежински збир после ротације ефикасно израчунамо на основу познатог тежинског збира пре ротације. Можемо уочити да је у тежинском збиру после померања улево сваки елемент низа, изузев првог елемента a_0, један пут мање укључен него пре померања, а први елемент је укључен $n-1$ пут. Обележимо са z_i тежински збир добијен приликом померања полазног низа улево i пута, а са z класичан збир свих елемената низа. Према томе важе следеће једнакости:

$$\begin{aligned}
z_0 &= 0 \cdot a_0 + 1 \cdot a_1 + 2 \cdot a_2 + ... + (n-2) \cdot a_{n-2} + (n-1) \cdot a_{n-1} \\
z_1 &= 0 \cdot a_1 + 1 \cdot a_2 + 2 \cdot a_3 + ... + (n-2) \cdot a_{n-1} + (n-1) \cdot a_0 \\
z_2 &= 0 \cdot a_2 + 1 \cdot a_3 + 2 \cdot a_4 + ... + (n-2) \cdot a_0 + (n-1) \cdot a_1 \\
&\dots \\
z_{n-1} &= 0 \cdot a_{n-1} + 1 \cdot a_0 + 2 \cdot a_1 + ... + (n-2) \cdot a_{n-3} + (n-1) \cdot a_{n-2}
\end{aligned}$$

$$z = a_0 + a_1 + ... + a_{n-2} + a_{n-1}$$

Приметимо да важи

$$\begin{aligned}
z_0 - z_1 &= a_1 + a_2 + \ldots + a_{n-1} - (n-1) \cdot a_0 = z - n \cdot a_0 \\
z_1 - z_2 &= a_2 + a_3 + \ldots + a_0 - (n-1) \cdot a_1 = z - n \cdot a_1 \\
&\dots \\
z_{n-2} - z_{n-1} &= a_{n-1} + a_0 + \ldots + a_{n-3} - (n-1) \cdot a_{n-2} = z - n \cdot a_{n-2}
\end{aligned}$$

итд.

Према томе $z_{i-1} - z_i = z - n \cdot a_{i-1}$, тј.

$$z_0 = z, \qquad z_i = z_{i-1} - z + n \cdot a_{i-1}, \text{ за } i > 0.$$

Дакле, тежински збир после померања за једно место улево можемо једноставно израчунати без померања низа на основу тежинског збира низа пре померања и збира свих елемената низа.

Имплементацију можемо извршити на следећи начин. Израчунамо тежински збир полазног низа $z_0 = \sum_{i=0}^{n-1} i \cdot a_i$ и класични збир свих елемената низа $z = \sum_{i=0}^{n-1} a_i$ (једноставним алгоритмом сабирања елемената низа). Од свих збирова треба израчунати највећи и одредити после колико ротација се та највећи збир постиже. Максимум ћемо иницијализовати на први израчунати тежински збир, а број померања ћемо иницијализовати на 0. Затим рачунамо тежинске збирове за низове добијене померањем низа за једно место улево i пута, и то редом за i од 1 до $n-1$. Тежински збир z_i за низ добијен након i померања рачунамо тако што претходни тежински збир z_{i-1} умањимо за збир свих елемената z и увећамо за $n \cdot a_{i-1}$. Проверавамо да ли је добијени тражени збир већи од дотадашњег максимума и ако јесте коригујемо максимум.

Наравно, све аритметичке операције вршимо по датом модулу.

Анализа сложености. Приметимо да време потребно за израчунавање почетних збирова линеарно зависи од дужине низа n, док је за израчунавање сваког наредног збира довољан константан број операција, тако да је укупна временска сложеност алгоритма линеарна тј. $O(n)$.

```
// zbir x i y po modulu mod
int zm(int x, int y, int mod) {
  return (x % mod + y % mod) % mod;
}

// razlika x i y po modulu mod
int rm(int x, int y, int mod) {
  return (x % mod - y % mod + mod) % mod;
}

// najveci tezinski zbir posle ciklicnog pomeranja
int maksTezinskiZbirNakonRotacije(const vector<int>& a) {
  // broj elemenata niza
  int n = a.size();
  // izracunavamo tezinsku sumu i*a[i] i obicnu sumu elemenata a[i] po
  // modulu mod
  int tezinskiZbir = 0;
  int suma = 0;
  for (int i = 0; i < n; i++) {
    tezinskiZbir = zm(tezinskiZbir, i * a[i], mod);
    suma = zm(suma, a[i], mod);
  }

  // najveca do sada vidjena tezinska suma
  int maxTezinskiZbir = tezinskiZbir;
  for (int i = 1; i < n; i++) {
    // rotacija za jedno mesto ulevo na sledeci nacin menja tezinsku sumu
    tezinskiZbir = zm(rm(tezinskiZbir, suma, mod), n * a[i - 1], mod);

    // proveravamo da li je dobijena tezinska suma veca od do tada
    // najvece
    if (tezinskiZbir > maxTezinskiZbir)
      maxTezinskiZbir = tezinskiZbir;
  }
  return maxTezinskiZbir;
}
```

2.12.4 Покретни прозор фиксне ширине

У многим ситуацијама потребно је израчунати одређену статистику свих сегмената низа фиксиране дужине. Тих сегмената је мање него свих сегмената (њихов број линеарно зависи од дужине низа), али ако се статистика сваког сегмента рачуна из почетка, добија се неефикасан алгоритам (квадратне сложености). Сваки наредни сегмент се разликује од претходног за тачно два елемента: приликом преласка са неког сегмента на наредни уклања се први елемент старог сегмента и додаје се последњи елемент новог сегмента. Ово омогућава инкрементално и ефикасно израчунавање многих статистика (оних заснованих на операцијама које имају инверзне, попут сабирања).

Задатак: Сегмент дужине k највећег просека

Дат је низ a реалних бројева дужине n и природан број k. Написати програм којим се у низу a одређује позиција почетка сегмента (подниза узастопних елемената) дужине k са највећим просеком (ако више сегмената има исти просек, пријавити последњи од њих).

Улаз: У првој линији стандардног улаза налази се природан број k ($k \leq 5 \cdot 10^3$). У другој линији налази се природан број n ($n \leq 5 \cdot 10^5$). У следећих n линија налазе се по један реалан број (ти бројеви представљају редом елементе низа a).

Излаз: На стандарном излазу приказати позицију почетка последњег сегмента дужине k низа a чији је просек највећи (позиције у низу се броје од нуле).

Пример

Улаз	*Излаз*
3 5 1.0 5.0 8.0 2.0 7.0	2

Решење

Приметимо да је проблем налажења сегмента дужине k чији је просек највећи, еквивалентан проблему налажења сегмента дужине k највећег збира. Просек сегмента добијамо дељењем суме сегмента са дужином сегмента, која је у овом задатку константа и износи k, тако да је просек највећи када је збир највећи.

Груба сила

Задатак решавамо тако што анализирамо све сегменте дужине k, почев од сегмента који почиње од елемента a_0 до сегмента који почиње од елемента a_{n-k}. Одређујемо збир сваког сегмента и уврђујемо који је последњи сегмент максималног збира. Одређивање позиције на којој почиње последњи сегмент максималног збира вршимо уобичајеним алгоритмом одређивања позиције максимума тј. минимума, при чему, пошто се тражи последњи сегмент, индекс максимума ажурирамо када год се наиђе на сегмент који има збир који је већи или једнак тренутном максимума (када би се тражио први сегмент, индекс би се ажурирао само када се наиђе на сегмент који има збир који је строго већи од тренутно максималног).

Директан приступ би био да се рачунање збира елемената сегмента који почиње од елемента a_i реализује тако што ће се редом сабирати елементи a_j за j од i до $i + k - 1$ (применом класичног алгоритма сабирања елемената низа).

```
// pronalazi indeks pocetka segmenta duzine k ciji je prosek najveci
int pocetakSegmentaNajvecegProseka(const vector<double>& a, int k) {
  // duzina niza
  int n = a.size();

  // trenutna maksimalna suma segmenta i indeks njenog pocetka
  double maxSuma = numeric_limits<double>::min();
  int maxPocetak = 0;

  for (int i = 0; i <= n - k; i++) {
    // izracunavamo sumu segmenta duzine k koji pocinje na poziciji i
    double suma = 0;
    for (int j = i; j < i+k; j++)
      suma += a[j];
    // double suma = accumulate(next(a, i), next(a, i+k), 0.0);

    // ako je potrebno, azuriramo maksimum
    if (suma >= maxSuma) {
      maxSuma = suma;
      maxPocetak = i;
    }
  }

  // vracamo pocetak poslednjeg segmenta sa maksimalnom sumom
  // (ujedno i prosekom)
  return maxPocetak;
}
```

Анализа сложености. Број операција сабирања у овом алгоритму приступу је око $(n-k) \cdot k$ (јер одређивање сваког од $n - k$ сегмената захтева око k операција сабирања), што даје квадратни број операција када је k око $n/2$. Временску сложеност најгорег случаја, дакле, можемо проценити са $O(n^2)$. Пошто све елементе

памтимо у низу, меморијска сложеност је $O(n)$.

Инкременталност - покретни прозор

Међутим, пошто се узастопни сегменти у великој мери преклапају (разликују им се само почетни и завршни елемент) могућ је и бољи приступ, у којем се збир наредног сегмента дужине k рачуна на основу познатог збира претходног сегмента дужине k. Ако са S_i обележимо збир сегмента који почиње од елемента a_i и дужине је k, $S_i = a_i + a_{i+1} + ... + a_{i+k-1}$, лако се може показати да за $i > 0$ важи једнакост $S_i = S_{i-1} - a_{i-1} + a_{i+k-1}$. Према томе можемо израчунати збир S_0 као збир првих k елемената низа (опет уобичајеним алгоритмом сабирања), а затим за свако i од 1 до $n - k$ наредни збир S_i добијамо на основу претходне једнакост. Дакле, ефикасније решење добијамо ако збир рачунамо инкрементално (сваки наредни члан се израчунава на основу претходних).

```
// pronalazi indeks pocetka segmenta duzine k ciji je prosek najveci
int pocetakSegmentaNajvecegProseka(const vector<double>& a, int k) {
  // duzina niza
  int n = a.size();

  // suma pocetnog segmenta duzine k
  double suma = 0;
  for (int i = 0; i < k; i++)
    suma += a[i];

  // trenutna maksimalna suma segmenta i indeks njenog pocetka
  int maxPocetak = 0;
  double maxSuma = suma;

  for (int i = 1; i <= n - k; i++) {
    // izracunavamo sumu segmenta duzine k koji pocinje na poziciji i
    suma = suma - a[i - 1] + a[i + k - 1];

    // ako je potrebno, azuriramo maksimum
    if (suma >= maxSuma) {
      maxSuma = suma;
      maxPocetak = i;
    }
  }

  // vracamo pocetak poslednjeg segmenta sa maksimalnom sumom
  // (ujedno i prosekom)
  return maxPocetak;
}
```

Анализа сложености. Оптимизовани приступ је знатно ефикаснији у односу на директни, јер се у сваком кораку петље врши само константан број операција, па је сложеност алгоритма линеарна тј. $O(n)$. Елементи се чувају у низу дужине n, па је меморијска сложеност $O(n)$.

Види другачија решења овог задатка.

2.13 Збирови префикса и разлике суседних елемената низа

Ако је дат низ елемената збир свих елемената у интервалу позиција $[a, b]$ се може израчунати као разлика између збира свих елемената у интервалу позиција $[0, b]$ и збира елемената у интервалу позиција $[0, a - 1]$.

На пример, размотримо како да израчунамо збир елемената на позицијама из интервала [3, 5] (тј. на позицијама 3, 4 и 5) у низу 4, 2, 3, 1, 5, 6, 9, 2. На тим позицијама се налазе елементи 1, 5 и 6 и збир им је $1 + 5 + 6 = 12$. Збир свих елемената из интевала позиција $[0, 5]$ је $4 + 2 + 3 + 1 + 5 + 6 = 21$, док је збир свих елемената из интервала позиција $[0, 2]$ једнак $4 + 2 + 3 = 9$. Разлика $21 - 9$ управо је једнака 12.

Ова наизглед веома једноставна особина сабирања може значајно помоћи убрзању разних алгоритама у којима су нам потребни збирови узастопних елемената низа. Наиме, ако знамо *збирове свих префикса* низа тј.

збирове на свим интервалима $[0, k)$, за $k = 0$ до n (а њих можемо израчунати током фазе претпроцесирања, инкрементално, у линеарној сложености), тада у константној сложености (једним одузимањем) можемо израчунати збир произвољног сегмента низа.

У језику C++ парцијалне збирове је могуће израчунати и коришћењем библиотечке функције `partial_sum`, која, наравно, ради у линеарној сложености. Функцији се прослеђују два итератора на део низа који се сабира, као и итератор на почетак низа у који се смештају резултати (пошто се унапред зна колико ће елемената бити, тај низ се унапред алоцира).

Дакле, од датог низа, низ збирова префикса можемо израчунати у линеарној сложености, али важи и обратно. Од низа префикса, у линеарној сложености можемо израчунати елементе оригиналног низа. Важи чак и јаче тврђење од тога, јер сваки конкретни елемент низа можемо наћи у константној сложености, одузимањем два суседна збира префикса. Дакле, прелазак са низа на збирове његових префикса можемо сматрати променом репрезентације (нема смисла чувати и једно и друго истовремено у меморији).

Приметимо огромну сличност са интегралним и диференцијалним рачуном. Израчунавање збирова префикса одговара одређеном интеграљењу, разлика збирова префикса одговара Њутн-Лајбницовој формули, док израчунавање разлике суседних елемената одговара диференцирању. Интеграљење и диференцирање су међусобно инверзне операције.

Дуалан приступ збировима префикса је промена репрезентације у којој уместо низа чувамо разлике суседних елемената. Повратак на оргинални низ се онда може извршити у линеарној сложености тако што израчунамо збирове префикса низа разлика. Ова репрезентација нам омогућава да веома ефикасно мењамо сегменте низа тако што све елементе из неког задатог сегмента увећамо или умањимо за неку фиксну вредност.

Задатак: Збирови сегмената

Позната је зарада једног предузећа током одређеног броја дана. Напиши програм који омогућава кориснику да израчунава укупну зараду предузећа у временским периодима одређеним почетним и крајњим даном.

Улаз: Са стандардног улаза се уноси број дана n ($1 \leq n \leq 100000$), а затим у наредном реду n целих бројева између 0 и 100, раздвојених са по једним размаком, који представљају зараде током n дана. Након тога се уноси број упита m ($1 \leq m \leq 100000$) и у наредних m редова се уносе временски периоди одређени редним бројем почетног дана a и крајњег дана b ($0 \leq a \leq b < n$).

Излаз: На стандардни излаз исписати m целих бројева који представљају укупне зараде у сваком од m периода.

Пример

Улаз	*Излаз*
5	15
1 2 3 4 5	9
3	3
0 4	
1 3	
2 2	

Решење

Директно решење

Директно решење подразумева да све бројеве учитамо у низ, а затим да за сваки упит изнова рачунамо збир одговарајућег сегмента низа.

```
// ucitavamo niz
int n;
cin >> n;
vector<int> brojevi(n);
for (int i = 0; i < n; i++)
  cin >> brojevi[i];

// ucitavamo granice segmenata i izracunavamo i ispisujemo njihove zbirove
int m;
cin >> m;
```

```
for (int i = 0; i < m; i++) {
  int a, b;
  cin >> a >> b;
  int zbir = 0;
  for (int j = a; j <= b; j++)
    zbir += brojevi[j];
  cout << zbir << endl;
}
```

Анализа сложености. Сложеност оваквог приступа је $O(nm)$.

Пошто се фаза учитавања и исписа података преплићу, учитавање би и испис података би требало додатно убрзати, но то не може поправити неефикасност овог наивног алгоритма.

Збирови префикса

Једноставно ефикасно решење је засновано на наредној идеји: уместо чувања елемената низа, можемо чувати низ збирова префикса низа. Збир сваког сегмента $[l, d]$ можемо разложити на разлику збира префикса до елемента d и префикса до елемента $l - 1$. Слична техника је употребљена у задатку Аритметички троугао. Ако користимо ознаку $\sum_{k=m}^{n} a_k$ која означава збир $a_m + a_{m+1} + \ldots + a_n$, можемо записати да је

$$\sum_{k=l}^{d} a_k = \sum_{k=0}^{d} a_k - \sum_{k=0}^{l-1} a_k.$$

Збирови свих префикса се могу израчунати и сместити у додатни (а ако је уштеда меморије битна, онда чак и у оригинални) низ. Дакле, током учитавања елемената можемо формирати низ збирова префикса (рачунаћемо их инкрементално, јер се сваки наредни збир префикса добија увећавањем претходног збира префикса за текући елемент низа). Нека z_i означава збир елемената префикса одређеног позицијама из интервала $[0, i)$. Формирамо, дакле, низ $z_i = \sum_{k=0}^{i-1} a_k$ (при чему је $z_0 = 0$, збир празног префикса). Тада збир елемената у сегменту позиција $[l, d]$ израчунавамо као $z_{d+1} - z_l$.

```
ios_base::sync_with_stdio(false); cin.tie(0);
// ucitavamo brojeve i izracunavamo zbirove prefiksa
int n;
cin >> n;
vector<int> zbirovi_prefiksa(n+1);
zbirovi_prefiksa[0] = 0;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  zbirovi_prefiksa[i+1] = zbirovi_prefiksa[i] + x;
}

// ucitavamo granice segmenata i izracunavamo i ispisujemo njihove zbirove
int m;
cin >> m;
for (int i = 0; i < m; i++) {
  int a, b;
  cin >> a >> b;
  cout << zbirovi_prefiksa[b+1] - zbirovi_prefiksa[a] << '\n';
}
```

Анализа сложености. За учитавање бројева и формирање низа збирова префикса потребно нам је $O(n)$ корака. Након оваквог претпроцесирања, збир сваког сегмента се може израчунати у времену $O(1)$, па је укупна сложеност $O(n + m)$.

Пошто се у овом задатку преплићу фаза учитавања и фаза исписа података на стандардни улаз и излаз, потребно је обратити пажњу на неефикасност која настаје због честог пражњења излазног бафера. Потребно је развезати `cin` и `cout` коришћењем `cin.tie(0)` и уместо помоћу `endl` у нови ред прелазити помоћу `\n`. На-

равно, ово има смисла само у случају аутоматске примене програма на велике улазе и излазе - овим изменама програм престаје да ради коректно у интерактивном режиму.

Задатак: Максимални збир сегмента

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Збирови префикса

Један алгоритам којим можемо ефикасно решити овај задатак се заснива на коришћењу збирова префикса. Сличну техника је описана, на пример, у задатку Збирови сегмената. Ако знамо збир сваког префикса низа, онда збир сваког сегмента можемо добити као разлику збирова два префикса. Збир елемената сегмента $[l, d]$ једнак је разлици збира елемената сегмента $[0, d + 1)$ и збира елемената сегмента $[0, l)$, тј. важи да је

$$\sum_{i=l}^{d} a_i = \sum_{i=0}^{d} a_i - \sum_{i=0}^{l-1} a_i$$

Рачунамо да је збир празног сегмента $[0, 0)$ по дефиницији једнак нули.

За сваку позицију у низу одређујемо максимални збир суфикса који се на тој позицији завршава. Највећи од свих максималних збирова суфикса је највећи збир неког сегмента у низу (јер је сваки сегмент заправо суфикс дела низа до оне позиције на којој се тај сегмент завршава).

Користимо индуктивноу конструкцију и низ проширујемо једним по једним елементом. Крећемо од празног низа чији је максимални збир сегмента једнак нули. Приликом сваког проширивања низа новим елементом, претпостављамо да знамо максимум збирова сегмента у непроширеном делу низа и да израчунамо максимални збир суфикса проширеног низа. Максимум збирова сегмента у проширеном низу је већи од та два броја.

Максимални збир суфикса проширеног низа, на основу разлагања на збирове префикса, добија се као разлика збира целог проширеног низа (тј. збира префикса до текуће позиције) и збира неког префикса непроширеног низа (празан суфикс не морамо анализирати, јер је празан сегмент већ обрађен у склопу иницијализације). Пошто је умањеник константан, да бисмо максимизовали разлику потребно да знамо најмањи могући умањилац, тј. да знамо најмањи збир префикса који се завршава на некој позицији испред текуће. И текући и минимални збир префикса можемо одржавати инкрементално. Инкрементално израчунавање збирова префикса већ је описано у задатку Највећи збир префикса.

Када год низ проширимо неким елементом, збир префикса увећавамо за тај елемент, поредимо га са дотадашњим минималним збиром префикса и ако је мањи, ажурирамо минимални збир префикса. Наравно, одржавамо и глобални максимални збир сегмента који ажурирамо сваки пут када наиђемо на суфикс чији је збир већи од дотадашњег максимума.

Приметимо да је у овом решењу је индуктивна хипотеза појачана и претпостављамо да поред сегмента највећег збира у обрађеном делу низа умемо да одредимо и минимални збир префикса обрађеног дела низа.

Анализа сложености. Сложеност овог решења је $O(n)$, па је ово решење оптималне сложености.

```
int zbir_prefiksa = 0;
int min_zbir_prefiksa = zbir_prefiksa;
int max = 0;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  zbir_prefiksa += x;
  int zbir_segmenta = zbir_prefiksa - min_zbir_prefiksa;
  if (zbir_segmenta > max)
    max = zbir_segmenta;
  if (zbir_prefiksa < min_zbir_prefiksa)
    min_zbir_prefiksa = zbir_prefiksa;
```

```
}
cout << max << endl;
```

Види другачија решења овог задатка.

Задатак: Број сегмената чији је збир дељив са k

Дат је низ a природних бројева дужине n и природан број k. Написати програм који одређује број сегмента низа a (непразних поднизова узастопних елемената) чији је збир дељив са k.

Улаз: У првој линији стандардног улаза налази се природан број k ($k \leq 10^5$). Друга линија стандардног улаза садржи природан број n ($n \leq 10^5$). У следећој линији се налази n природних бројева (ти бројеви представљају редом елементе низа a), раздвојених са по једним размаком.

Излаз: На стандардном излазу приказати колико постоји сегмената низа a чији је збир дељив са k.

Пример

Улаз	*Излаз*
3 5 1 8 2 3 4	4

Објашњење: то су сегменти $(1, 8)$, (3), $(2, 3, 4)$ и $(1, 8, 2, 3, 4)$

Решење

Груба сила

Директан начин да се задатак реши је да се угнежђеним петљама наброје сви сегменти, да се за сваки израчуна сума и да се провери да ли је дељива са k, бројећи успут такве сегменте. Број је дељив са k ако и само ако даје остатак 0 при дељењу са k, а знамо да је остатак при дељењу збира са бројем k заправо једнак збиру остатака тј. да важи да је $(a + b) \bmod k = (a \bmod k + b \bmod k) \bmod k$.

Дакле, за сваки сегмент довољно је израчунати збир по модулу k, при чему за фиксирани леви крај сегмента, збир сваког наредног сегмента $[i, j]$ добијамо инкрементално, од збира сегмента $[i, j - 1]$, додајући на њега број a_j и израчунавајући остатак добијеног збира при дељењу са k. Збир префикса смо рачунали инкрементално, на пример, у задатку Највећи збир префикса.

```
// izracunava broj segmenata niza a ciji je zbir deljiv sa k
int brojSegmenataZbiraDeljivogSaK(const vector<int>& a, int k) {
  // duzina niza
  int n = a.size();
  // broj segmenata deljivih sa k
  int broj = 0;
  // obradjujemo sve segmente [i, j]
  for (int i = 0; i < n; i++) {
    // zbir po modulu k segmenta [i, j] inicijalizujemo na nulu
    int s = 0;
    for (int j = i; j < n; j++) {
      // azuriramo zbir po modulu k segmenta [i, j] na osnovu zbira [i, j-1]
      s = (s + a[j]) % k;
      // ako je zbir po modulu k jednak 0, zbir je deljiv sa k
      if (s == 0)
        broj++;
    }
  }
  return broj;
}
```

Анализа сложености. Уз овако инкрементално израчунавање збира, сложеност алгоритма једнака је броју сегмената што је $O(n^2)$.

Остаци збирова префикса

Тражимо број сегмената $a_p, a_{p+1}, \ldots, a_q$, за $0 \leq p \leq q < n$, таквих да је збир $S_{pq} = a_p + a_{p+1} + ... + a_q$ дељив са k.

Обележимо са $S_0 = 0$, $S_1 = a_0$, $S_2 = a_0 + a_1$, итд., тј. обележимо са S_i, $0 < i \leq n$ збир првих i елемената низа ($S_i = a_0 + a_1 + ... + a_{i-1}$). Збир S_{pq} можемо изразити као $S_{q+1} - S_p$. Сличну технику користили смо, на пример, у задатку Сегмент датог збира у низу целих бројева.

На основу особина операција по модулу важи да је $S_{pq} \bmod k = (S_{q+1} \bmod k - S_p \bmod k + k) \bmod k$.

Према томе збир S_{pq} је дељив са k (тј. важи $S_{pq} \bmod k = 0$) акко збирови S_{q+1} и S_p имају исти остатак при дељењу са k тј. ако је $S_{q+1} \bmod k = S_p \bmod k$.

Обележимо са b_r број збирова S_i (за $0 \leq i \leq n$) који при дељењу са k дају остатак r (за свако $0 \leq r < k$). Сваки пар (различитих) збирова префикса који дају исти остатак r одређује тачно један сегмент чији је збир елемената дељив са k. У скупу од m различитих елемената постоји тачно $\frac{m(m-1)}{2}$ различитих парова. Зато је за свако r број сегмената који се добија комбинујући два збира који дају остатак r једнак $\frac{b_r \cdot (b_r - 1)}{2}$. Према томе укупан број сегмената дељивих са k је $\sum_{r=0}^{k-1} \frac{b_r \cdot (b_r - 1)}{2}$.

Остаје још само питање како избројати збирове префикса за сваки дати остатак тј. како израчунати све бројеве b_r. Низом b дужине k памтимо број префикса чији збирови елемената дају остатке редом $0, 1, 2, \ldots, k-1$ тако да је b_r једнак броју префикса чији збир елемената при дељењу са k даје остатак r (што је могуће, с обзиром на дату горњу границу броја k). Збирове префикса, наравно, израчунавамо инкрементално. Слично смо радили, на пример, у задатку Највећи збир префикса.

Полазни збир је $S_0 = 0$, тако да све елементе низа b иницијализујемо на 0, осим вредности на позицији 0 коју иницијализујемо на 1. Збир текућег префикса одржавамо у променљивој S коју иницијализујемо на нулу. Учитавамо члан по члан низа x, и при томе ажурирамо збир префикса (S постављамо на $(S + x) \bmod k$), увећавајући одговарајући бројач (вредност b_S увећавамо за 1).

Пример. Прикажимо рад овог алгоритма на примеру одређивања броја сегмената низа $1, 8, 2, 3, 4$ који су дељиви са 3. Могући остаци су 0, 1 и 2.

```
i  ai Si  b0 b1 b2
      0   1  0  0
0  1  1   1  1  0
1  8  0   2  1  0
2  2  2   2  1  1
3  3  2   2  1  2
4  4  0   3  1  2
```

Зато је коначан резултат $\frac{b_0(b_0-1)}{2} + \frac{b_1(b_1-1)}{2} + \frac{b_1(b_1-1)}{2} = 3 + 0 + 1 = 4$.

```
// izracunava broj segmenata niza a ciji je zbir deljiv sa k
int brojSegmenataZbiraDeljivogSaK(const vector<int>& a, int k) {
  // duzina niza
  int n = a.size();

  // na mestu i u nizu br čuvamo broj segmenata čiji zbir pri
  // deljenju sa k daje ostatak i
  vector<int> br(k, 0);
  br[0] = 1;

  // zbir tekućeg segmenata
  int s = 0;
  for (int i = 0; i < n; i++) {
    // ažuriramo zbir elemenata tekućeg segmenta po modulu k
    s = (s + a[i]) % k;
    br[s]++;
  }

  // izračunavamo ukupan broj segmenata deljivih sa k
```

```
  int broj = 0;
  for(int i = 0; i < k; i++)
    broj += br[i]*(br[i]-1)/2;

  return broj;
}
```

Анализа сложености. Временска сложеност овог алгоритма је, јасно, $O(n)$. Приметимо да у овом решењу није било потребно користити низ за бројеве које уносимо, јер при уносу обрадимо сваки елемент, међутим, користимо помоћни низ дужине k, па је меморијска сложеност $O(k)$. Обратимо пажњу на то да ово може бити ограничавајући фактор, ако k може бити јако велики број (што у овом задатку није случај).

Задатак: Увећавање сегмената

Камион превози терет током N километара пута. На пут креће празан и током пута утоварује и истоварује пакете. Ако се за сваки пакет зна на ком је километру пута утоварен, на ком је километру пута истоварен и колика му је маса, напиши програм који одређује колико је оптерећење камиона на сваком километру пута. Сматрати да се предмет утоварује на почетку, а истоварује на крају датог километра.

Улаз: Са стандардног улаза се уноси број километара N ($10 \leq N10000$), затим, у наредном реду, број предмета M ($0 \leq M \leq 10000$), а након тога, у наредних M редова по три цела броја раздвојена размацима који представљају број километра на чијем је почетку утоварен предмет (цео број између 0 и $N-1$), број километра на чијем крају је истоварен (цео број између 0 и $N-1$) и на крају маса предмета (цео број између 1 и 10).

Излаз: На стандардни излаз исписати масу терета у килограмима на сваком километру пута (иза сваке масе написати по један размак).

Пример

Улаз
```
10
3
1 5 10
3 7 10
2 8 15
```

Излаз
```
0 10 25 35 35 35 25 25 15 0
```

Објашњење

```
        km   0   1  2  3  4  5  9  7  8  9
             0   0  0  0  0  0  0  0  0  0
1 5 10       0  10 10 10 10 10  0  0  0  0
3 7 10       0  10 10 20 20 20 10 10  0  0
2 8 15       0  10 25 35 35 35 25 25 15  0
```

Решење

Директно решење

Директан начин је да се одржава низ M у којем се памти маса на камиону током сваког километра пута. Након учитавања сваког податка о предмету (почетног километра a, завршног километра b и масе m), све вредности у низу M на позицијама од a до b (укључујући и њих) се увећавају за m.

Анализа сложености. Проблем са овим решењем је то што предмети могу путовати велики број километара па се у сваком кораку врши ажурирање великог броја чланова низа (сложеност је у најгорем случају $O(n \cdot m)$).

```
int n;
cin >> n;
vector<int> mase(n, 0);
int m;
cin >> m;
for (int i = 0; i < m; i++) {
  int km_od, km_do, masa;
  cin >> km_od >> km_do >> masa;
```

```
  for (int km = km_od; km <= km_do; km++)
    mase[km] += masa;
}

for (int masa : mase)
  cout << masa << " ";
```

Разлике суседних елемената низа

Задатак можемо решити ефикасније ако уместо да у низу M одржавамо масу у камиону у километру i, одржавамо разлику између масе у километру i и $i-1$ (на позицији 0 се чува маса у камиону у нултом километру). Дакле, уводимо низ R_i такав да је $R_0 = M_0$, а $R_i = M_i - M_{i-1}$, за $1 \leq i < n$. Посматрајмо шта се дешава са низом R када се у низу M сви елементи на позицијама a до b увећају за m.

- Вредност R_a једнака је разлици $M_a - M_{a-1}$ (или евентуално M_0 ако је $a = 0$) и она се увећава за m, јер је M_a увећан за m, док се M_{a-1} не мења.
- Све вредности од R_{a+1} до R_b остају не промењене. Наиме, за све њих важи да је $R_i = M_i - M_{i-1}$, а да су и M_i и M_{i-1} увећани за m.
- На крају, вредност R_{b+1} се умањује за m. Наиме важи да је $R_{b+1} = M_{b+1} - M_b$, да се M_b увећава за m, док се M_{b+1} не мења.

Рецимо да ако је $b = n - 1$, тада не морамо разматрати вредност $R_{b+1} = R_n$ (мада, униформности ради, можемо, што захтева да низ R садржи $n + 1$ елемент). Дакле, приликом сваког учитавања бројева a, b и m потребно је само да увећавамо елемент R_a за m, а да елемент R_{b+1} умањимо за m.

Када знамо елементе низа R елементе низа M можемо једноставно реконструисати сабирањем елемената низа R. Наиме, важи да је $M_0 = R_0$, док је $M_i = M_{i-1} + R_i$, тако да сваки наредни елемент низа M можемо израчунати као збир претходног елемента низа M и њему одговарајућег елемента низа R. Приметимо да је заправо елемент M_i једнак збиру свих елемената од R_0 до R_i, јер је $R_0 + R_1 + \ldots + R_i = M_0 + (M_1 - M_0) + \ldots + (M_i - M_{i-1}) = M_i$.

Анализа сложености. Укупна сложеност овог приступа је $O(n + m)$.

```
int n;
cin >> n;
vector<int> razlika(n+1, 0);
int m;
cin >> m;
for (int i = 0; i < m; i++) {
  int km_od, km_do, masa;
  cin >> km_od >> km_do >> masa;
  razlika[km_od] += masa;
  razlika[km_do+1] -= masa;
}

int masa_km = 0;
for (int km = 0; km < n; km++) {
  masa_km += razlika[km];
  cout << masa_km << " ";
}
```

Напомена. Идеја коришћена у овом задатку донекле је слична (заправо инверзна) техници одређивања збира сегмената као разлике два збира префикса. Ту технику смо, на пример, применили у задатку Збирови сегмената. Може се приметити да се реконструкција низа врши заправо израчунавањем префиксних збирова низа разлика суседних елемената, што указује на дубоку везу између ове две технике. Заправо, разлике суседних елемената представљају одређени дискретни аналогон извода функције, док префиксни збирови представљају аналогију одређеног интеграла. Израчунавање збира сегмента као разлике два збира префикса одговара Њутн-Лајбницовој формули.

2.14 Сортирање

Због својих примена, сортирање је један од најзначајнијих и најпроучаванијих проблема у рачунарству. Током година развијен је велики број алгоритама за сортирање. Најпознатији су *QuickSort*, *MergeSort*, *HeapSort*, *SelectionSort*, *InsertionSort*, *BubbleSort* и *ShellSort*. Неки од њих су једноставнији, али спорије раде (такви су *Selection*, *Insertion* и *Bubble*), док су неки ефикаснији, али компликованији за разумевање и реализацију (такви су *Quick*, *Merge*, *Heap* и *Shell*). Њихова ручна имплементација поучна је као одличан пример за приказ различитих техника конструкције алгоритама.

Ипак, по правилу све библиотеке савремених (а и старијих) програмских језика нуде готове функције за сортирање низова. Коришћење ових функција је увек пожељнија опција у односу на ручну имплементацију (добијају се краћи и разумљивији програми, функције су пажљиво тестиране и скоро извесно потпуно коректне, и имплементиране су на најефикаснији могући начин). Можемо слободно претпоставити да је сложеност и најгорег и просечног случаја код библиотечких функција сортирања квазилинеарна тј. једнака $O(n \log n)$ - у делу посвећеном рекурзији и техници "подели па владај", биће приказани најзначајнији алгоритми сортирања (који се користе и у библиотечким функцијама сортирања) и њихова анализа сложености.

У наставку овог поглавља кроз неколико веома елементарних примера ћемо приказати употребу библиотечких функција сортирања, као и пар елементарних алгоритама сортирања (који се једноставно имплементирају, али су квадратне сложености и стога их никада у пракси не треба употребљавати).

Сортирање подразумева да се елементи ређају у односу на неки поредак. Бројеви се подразумевано ређају по њиховој нумеричкој вредности, ниске се подразумевано ређају лексикографски абецедно (као у речнику), парови и торке се поново ређају лексикографски, по својим компонентама, међутим, често је потребно сортирати низове елемената над којима није дефинисан подразумевани поредак (на пример, низове структура). Библиотечке функције сортирања допуштају навођење различитих критеријума сортирања (задавањем функција поређења), што ће такође бити илустровано кроз неколико елементарних задатака.

Након тога, приказаћемо употребу сортирања као технике претпроцесирања која омогућава да се снизи сложеност решења задатака. Сортирање је толико корисна техника, да у свету конструкције алгоритама често важи правило "Ако не знаш одакле да кренеш да конструишеш алгоритам, ти прво покушај да сортираш податке". Једна од најзначајнијих примена сортирања је то што се сортирани подаци могу брзо претраживати (применом алгоритма бинарне претраге). Стога ће у наредном поглављу посебна пажња бити посвећена управо томе.

Задатак: Сортирање бројева

Напиши програм који уређује (сортира) низ бројева неопадајуће (сваки наредни мора да буде већи или једнак од претходног).

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 5 \cdot 10^4$) а затим и n природних бројева мањих од $2n$, сваки у посебном реду.

Излаз: На стандардни излаз исписати учитане бројеве у сортираном редоследу.

Пример

Улаз	*Излаз*
5	1
3	1
1	3
6	6
8	8
1	

Решење

Сортирање вектора библиотечком функцијом

У језику C++ сортирање је најбоље вршити функцијом `sort`, која прима два итератора - први који указује на почетак дела низа тј. вектора који се сортира, а други указује иза последњег елемента у делу низа који се сортира. Када се сортира цео вектор `a` ти итератори се могу добити помоћу `a.begin()` и `a.end()`.

Анализа сложености. Сложеност овог приступа је $O(n \log n)$.

```
// ucitavanje brojeva
int n;
cin >> n;
vector<int> a(n);
for (int i = 0; i < n; i++)
  cin >> a[i];

// sortiranje
sort(a.begin(), a.end());

// ispis rezultata
for (int i = 0; i < n; i++)
  cout << a[i] << endl;
```

Сортирање низа библиотечком функцијом

Када се сортира низ `a` са `n` елемената почетни итератор је `a`, а завршни се може добити као `next(a, n)` или евентуално `a+n`.

Анализа сложености. Сложеност овог приступа је $O(n \log n)$.

```
// ucitavanje brojeva
int n;
cin >> n;
int a[MAX];
for (int i = 0; i < n; i++)
  cin >> a[i];

// sortiranje
sort(a, next(a, n));

// ispis rezultata
for (int i = 0; i < n; i++)
  cout << a[i] << endl;
```

Види другачија решења овог задатка.

2.14.1 Обрада дупликата (поновљених вредности у низу)

У неким задацима је потребно на неки начин обрадити све поновљене вредности у низу (дупликате). Ефикасна решења се обично добијају након што се низ претпроцесира коришћењем сортирања. Након сортирања низа сви поновљени елементи се налазе један иза другога, што значајно онда олакшава њихову обраду (за сваки елемент је веома једноставно проверити колико пута се јавио у низу, па је самим тим једноставно проверити и да ли је дупликат, уклонити дупликате и слично). Осим сортирањем, обрада дупликата се може вршити и помоћу библиотечких колекција (скупова, мултискупова и мапа тј. речника), о чему ће више речи бити касније.

Задатак: Дупликати

Претпоставимо да су интернет адресе представљене природним бројевима (IP адресе се, на пример, чувају у облику неозначених 32-битних бројева). Претраживач чува списак свих адреса које је корисник посетио током неког претходног периода. Корисник је многе адресе посећивао и више пута. Напиши програм који одређује број различитих адреса које је корисник посетио.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 10^5$), а затим и n природних бројева (мањих од 2^32), сваки у посебном реду.

Излаз: На стандардни излаз исписати број различитих адреса које је корисник посетио.

Пример

Улаз	*Излаз*
8 123456789 234567890 345678901 234567890 456789012 234567890 456789012 234567890	4

Решење

Претрага

Наиван начин да се детектују дупликати се може засновати на алгоритму линеарне претраге. Од свих учитаних елемената низа бројаћемо само оне који се први пут појављују (када се неки елемент појави други, трећи итд. пут, нећемо га бројати). За сваки елемент низа на позицији i (од нула до $n-1$) провераваћемо да ли се тај елемент јавља на некој позицији од 0 до $i-1$. То можемо урадити алгоритмом линеарне претраге. Линеарна претрага се може имплементирати и библиотечком функцијом.

Анализа сложености. Ово решење је неефикасно и сложеност му је квадратна у односу на димензију низа (сложеност му је $O(n^2)$, где је n број елемената низа).

```
// broj razlicitih elemenata niza a
int brojRazlicitih(const vector<unsigned>& a) {
  int broj = 0;
  // za svaki element niza a
  for (int i = 0; i < a.size(); i++) {
    // proveravamo da li se a[i] javlja pre pozicije i
    bool sadrzi = false;
    for (int j = 0; j < i; j++)
      if (a[i] == a[j]) {
        sadrzi = true;
        break;
      }
    // ako se ne pojavljuje, uracunavamo ga
    if (!sadrzi)
      broj++;
  }
  return broj;
}
```

Сортирање

Један од најчешћих начина уклањања дупликата из низа је заснован на сортирању, јер се након сортирања дупликати нађу један до другог. Сортирање можемо најбоље урадити позивом библиотечке функције. Разни начини на које је могуће сортирати низ бројева су приказани у задатку Сортирање бројева. Након сортирања пролазимо редом кроз низ и бројимо први елемент, а затим и све елементе који су различити од њима претходног (то су прва појављивања елемената у сортираном низу).

Често је непожељно да се током неке анализе података подаци трансформишу, јер то може да спречи неке наредне анализе података. На пример, ако сортирамо низ, даље анализе у којима је важан редослед података неће бити могуће, јер се губи информација о полазном редоследу. Због тога је пре сортирања понекад неопходно направити копију низа.

Анализа сложености. Сложеношћу овог приступа доминира сложеност поступка сортирања. Пролаз након сортирања је линеарне сложености, а сортирање се може остварити у сложености $O(n \log n)$, где је n број елемената низа.

```
// broj razlicitih elemenata niza a
int brojRazlicitih(const vector<unsigned>& a) {
  // pravimo kopiju niza, da bi originalni niz ostao nepromenjen
```

```
  auto as = a;
  // sortiramo niz
  sort(as.begin(), as.end());
  // brojimo prvi element i sve elemente koje su razliciti od
  // svojih prethodnika
  int broj = 1;
  for (int i = 1; i < as.size(); i++)
    if (as[i] != as[i-1])
      broj++;
  return broj;
}
```

Библиотечка функција за избацивање дупликата

Избацивање дупликата у сортираном низу може се вршити и библиотечким функцијама. Функција `unique` у језику C++ прима два итератора који ограничавају сегмент сортираног низа тј. вектора (обично `a` и `next(a, n)` тј. `a+n` или `a.begin()` и `a.end()`) и организује тај низ тако да на његов почетак смешта елементе који се не понављају и враћа итератор на позицију иза краја тог дела. На пример, ако је низ $1, 1, 2, 2, 5, 5$, након позива функције `unique` он ће бити реорганизован тако што ће му садржај бити $1, 2, 5, x, x, x$, а итератор ће указивати на позицију три тј. непосредно иза елемента 5. Саржај на позицијама обележеним са x није релевантан. Ако се жели избацивање дупликата, реп низа се може одстранити тако што се позове функција `erase` којој се наведу два итератора - први је онај који је вратила функција `unique` а други је итератор на крај низа (који се обично добија са `next(a, n)`, `a + n` или `a.end()`). Ако је потребно израчунати само број јединствених елемената он се може добити тако што се израчуна растојање између итератора на почетак низа и итератора којег врати функција `unique` (на пример, `distance(a, unique(a, next(a, n)))`).

Анализа сложености. Библиотечке функције за уклањање дупликата из сортиране колекције сложености су $O(n)$, па укупном сложеношћу доминира сортирање сложености $O(n \log n)$.

```
// broj razlicitih elemenata niza a
int brojRazlicitih(const vector<unsigned>& a) {
  // pravimo kopiju niza, da bi originalni niz ostao nepromenjen
  auto as = a;
  // sortiramo niz
  sort(as.begin(), as.end());
  // uklanjamo duplikate
  as.erase(unique(as.begin(), as.end()), as.end());
  return as.size();
  // dovoljno je i:
  // return distance(as.begin(), unique(as.begin(), as.end()));
}
```

Види другачија решења овог задатка.

Задатак: Неупарени елемент

Домаћица матица је на журку позвала своје пријатеље, брачне парове пчеле и трутове. Пошто је гостију пуно, свако је добио број столице. Брачни парови су добили исте бројеве. Који број је добила матица?

Улаз: Са стандардног улаза уноси се број n, а затим и n природних бројева, сваки у посебном реду, од којих се сви осим једног јављају тачно два пута.

Излаз: На стандардни излаз исписати један број - онај који се на улазу јавио тачно једном.

Пример

Улаз	*Излаз*
9 3 2 1 4 2 5 5 3 1	4

Решење

Груба сила

Директно решење задатка подразумева да се за сваки елемент учитаног низа провери да ли у низу постоји још неко његово појављивање. Дакле, користимо угнежђене петље и у спољној петљи пролазимо кроз све позиције i у низу, док у унутрашњој петљи користимо алгоритам линеарне претраге да бисмо проверили да ли у низу постоји позиција j таква да је различита од i, а да је $a_i = a_j$. Разни начини имплементације линеарне претраге описани су у задатку Неупарени елемент. Први пут када се унутрашња петља неуспешно заврши, пријављујемо да је неупарени елемент a_i и прекидамо и спољашњу петљу. Наравно, претрагу је могуће издвојити и у помоћну функцију или употребити библиотечку функцију.

Анализа сложености. Пошто се суштински упоређују сви парови елемената, а њих у низу дужине n има $\frac{n(n-1)}{2}$, сложеност овог решења је $O(n^2)$.

```
int neupareni(const vector<int>& a) {
  // dimenzija niza
  int n = a.size();
  for (int i = 0; i < n; i++) {
    bool uparen = false;
    for (int j = 0; j < n; j++)
      if (i != j && a[i] == a[j]) {
        uparen = true;
        break;
      }
    if (!uparen)
      return a[i];
  }
  // po uslovima zadatka dovde nikada ne bi trebalo da se dodje
  throw "ne postoji neupareni element";
}
```

Линеарна претрага може бити реализована и библиотечком функцијом (за сваки елемент посебно претражујемо део низа испред и део низа иза њега).

```
int neupareni(const vector<int>& a) {
  auto b = begin(a), e = end(a);
  for (auto it = b; it != e; it++)
    if (find(b, it, *it) == it && find(next(it), e, *it) == e)
      return *it;
  // po uslovima zadatka dovde nikada ne bi trebalo da se dodje
  throw "ne postoji neupareni element";
}
```

Сортирање

Задатак се ефикасније може решити ако се низ претходно сортира. Сортирање можемо урадити на разне начине, али најбољи је да се употреби бибилотечка функција за сортирање. Разни начини сортирања низа бројева описани су у задатку Сортирање бројева. Након тога се сви парови налазе један иза другога и потребно је пролазити низ два-по-два елемента проверавајући да ли је други различит од првог (ако јесте, онда је први

елемент тај који се јавља само један пут). Посебну пажњу треба обратити на случај када је тражени елемент последњи у сортираном низу.

Анализа сложености. Сложеношћу доминира сортирање, које је сложености $O(n \log n)$. Провера једнакости узастопних елемената која наступа након сортирања врши се у једном пролазу кроз низ и линеарне је сложености тј. $O(n)$.

```
int neupareni(const vector<int>& a) {
  // dimenzija niza
  int n = a.size();
  // pravimo kopiju niza, da se niz ne bi promenio
  auto as = a;
  // sortiramo niz
  sort(begin(as), end(as));
  // gledamo dva po dva elementa niza
  for (int i = 0; i < n-1; i += 2)
    if (as[i] != as[i+1])
      return as[i];
  if (n % 2 == 1)
    return as[n-1];
  // po uslovima zadatka dovde nikada ne bi trebalo da se dodje
  throw "ne postoji neupareni element";
}
```

Задатак: Фреквенције речи

Милица је куповала разне производе и сваки пут када се вратила из куповине дописивала је све производе које је купила на списак. Напиши програм који помаже Милици да одреди производ који је најчешће куповала током претходне године.

Улаз: Са стандардног улаза учитавају се називи производа, све док се не дође до краја улаза. Сваки назив је реч која садржи највише 10 малих слова енглеског алфабета.

Излаз: На стандардни излаз исписати један ред који садржи назив најчешће купованог производа, један размак и број пута колико је тај производ купљен. Ако је више производа купљено исти број пута, исписати онај који је први по абецедном редоследу.

Пример

Улаз	*Излаз*
jabuka hleb kruska jabuka sljiva hleb mleko jabuka hleb	hleb 3

Решење

Сортирање

Још један могући приступ решавању овог проблема је да се речи сортирају тако да сва појављивања исте речи буду узастопна. Пошто се не зна унапред број речи, најбоље их је учитати у неки динамички низ. У језику C++ то може бити `vector`.

Сортирање је најбоље урадити лексикографски, применом библиотечке функције. Након сортирања, тражимо дужину најдуже узастопне серије једнаких елемената. То можемо слично као у задатку Најдужа серија победа.

```
string s;
vector<string> reci;
while (cin >> s)
```

```
  reci.push_back(s);

sort(begin(reci), end(reci));

int tekucaDuzina = 1;
int maxDuzina = 1; string maxRec = reci[0];
for (int i = 1; i < reci.size(); i++) {
  if (reci[i] == reci[i-1])
    tekucaDuzina++;
  else
    tekucaDuzina = 1;
  if (tekucaDuzina > maxDuzina) {
    maxDuzina = tekucaDuzina;
    maxRec = reci[i];
  }
}

cout << maxRec << " " << maxDuzina << endl;
```

Види другачија решења овог задатка.

2.14.2 Груписање блиских вредности

Још једна примена сортирања долази од тога што се након сортирања низа, елементи блиски по вредности наћи ће један близу другог. Ово нам омогућава да у низу налазимо што ближе елементе као и групе што блискијих елемената (са што мањом разликом између најмањег и највећег елемента у групи).

Задатак: Најближе собе

Два госта су дошла у хотел и желе да одседну у собама које су што ближе једна другој, да би током вечери могли да заједно раде у једној од тих соба. Ако постоји више таквих соба, они бирају да буду што даље од рецепције, тј. у собама са што већим редним бројевима, како им бука не би сметала. Ако је познат списак слободних соба у том тренутку, напиши програм који одређује бројеве соба које гости треба да добију.

Улаз: У првој линији стандардног улаза налази се број n ($1 \leq n \leq 10^5$), а затим се у наредним линијама налазе бројеви слободних соба (у свакој линији по један) - сви бројеви су различити, али је њихов редослед произвољан.

Излаз: На стандардни излаз исписати бројеве соба гостију (прво мањи број, па већи), раздвојене једним размаком.

Пример

Улаз	*Излаз*
7 18 6 25 11 4 1 16	16 18

Решење

Груба сила

Директан приступ решењу би био да се израчунају растојања између сваке две собе и да се пронађе пар са најмањим растојањима.

Анализа сложености. Пошто парова има $\frac{n(n-1)}{2}$, сложеност овог приступа би била $O(n^2)$.

```
void najblizeSobe(const vector<int>& a, int& soba1, int& soba2) {
  // broj soba
  int n = a.size();
```

```
  // dve sobe sa najmanjim rastojanjem
  int min_i = a[0], min_j = a[1];
  // rastojanje izmedju njih
  int d_min = abs(min_i - min_j);
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++) {
      // rastojanje izmedju soba i i j
      int d_ij = abs(a[i] - a[j]);
      // ako je rastojanje manje od do sada najmanjeg ili je jednako,
      // ali su sobe dalje od recepcije
      if (d_ij < d_min || (d_ij == d_min && min(min_i, min_j) < min(a[i], a[j]))) {
        // azuriramo najblize sobe i rastojanje izmedju njih
        min_i = a[i];
        min_j = a[j];
        d_min = d_ij;
      }
    }
  // vracamo sobe u uredjenom redosledu
  soba1 = min(min_i, min_j); soba2 = max(min_i, min_j);
}
```

Сортирање

Боље решење се може добити ако се низ пре тога сортира. Наиме, најближи елемент сваком елементу у сортираном низу је један од њему суседних. Дакле, ако број a_i учествује у пару најближих соба, онда други елемент тог пара може бити или број a_{i-1} који је непосредно испред a_i у сортираном редоследу или број a_{i+1} који је непосредно иза њега (наравно, не постоји елемент испред првог, нити елемент иза последњег елемента низа).

Доказ коректности. Заиста, у неопадајуће сортираном низу важи да из $j' < j < i$ следи $a_i - a_j \leq a_i - a_{j'}$, јер из сортираности и $j' < j$ следи да је $a'_j \leq a_j$. Важи и да из $i < j < j'$ следи да је $a_j - a_i \leq a_{j'} - a_i$, јер из сортираности и $j < j'$ следи $a_j \leq a_{j'}$. Зато за свако $j' < i - 1$ важи да је $a_i - a_{i-1} \leq a_i - a_{j'}$, па елемент на позицији j' није ближи елементу a_i него елемент a_{i-1}. Слично, за свако $j' > i + 1$ важи да је $a_{i+1} - a_i \leq a_{j'} - a_i$, па елемент на позицији j' није ближи елементу a_i него елемент a_{i+1}.

Зато је након сортирања довољно проверити све разлике између суседних елемената и одредити најмању од њих (ако има више истих, одређујемо последњу). За ово користимо алгоритам одређивања најмањег елемента, док сортирање можемо најлакше ефикасно извршити библиотечком функцијом.

Анализа сложености. Сортирање библиотечком функцијом захтева $O(n \log n)$ операција, док је тражење минимума сложености $O(n)$, тако да је укупно време овог поступка $O(n \log n)$.

```
void najblizeSobe(const vector<int>& a, int& soba1, int& soba2) {
  // pravimo duplikat niza koji cemo da sortiramo
  auto as = a;
  sort(begin(as), end(as));
  int min = 1;
  for (int i = 2; i < a.size(); i++)
    if (as[i] - as[i-1] <= as[min] - as[min-1])
      min = i;
  soba1 = as[min-1]; soba2 = as[min];
}
```

Задатак: Праведна подела чоколадица

Дато је n пакета чоколаде и за сваки од њих је познато колико чоколадица садржи. Сваки од k ученика узима тачно један пакет, при чему је циљ да сви ученици имају што приближнији број чоколадица. Колика је најмања могућа разлика између оног ученика који узме пакет са најмање и оног који узме пакет са највише чоколадица.

Улаз: Са стандардног улаза се уноси природан број n ($1 \leq n \leq 50000$) а затим и n природних бројева (између 1 и 10^6, раздвојене са по једним размаком) који представљају број чоколадица у сваком пакету. У

последњем реду се уноси број деце k ($1 \leq k \leq n$).

Излаз: На стандардни излаз исписати вредност најмање разлике.

Пример

Улаз	*Излаз*
8 3 8 1 17 4 7 12 9 4	5

Решење

Сортирање

Најдиректнији начин да се реши задатак је да се испитају сви подскупови од k елемената скупа од n елемената и да се међу њима одабере најбољи. Ово решење је релативно компликовано имплементирати, а уз то је и веома неефикасно (број подскупова је $\binom{n}{k}$, што је $O(n^k)$).

Боље и ефикасније решење се заснива на сортирању. Наиме, када се полазни пакети сортирају по броју чоколадица, ученици треба да узму узастопних k пакета. Претпоставимо да након сортирања имамо низ $a_0, a_1, \ldots, a_{n-1}$. Ученици треба да узму редом пакете од a_i, до a_{i+k-1}, за неко $0 \leq i \leq n-k$.

Доказ коректности. Докажимо претходну чињеницу и формално. Претпоставимо супротно, да пакети који дају најмањи распон не чине узастопан низ и да сваки узастопни низ пакета дужине k има строго већи распон од распона скупа узетих пакета. Нека је први узети пакет a_i. Тада сигурно постоји неки пакет a_j за $i < j < i + k$ који није узет, а уместо њега је узет неки пакет $a_{j'}$ за неко $i + k \leq j' < n$. Нека је j' последњи пакет који је узет. Међутим, када би ученик који је узео пакет $a_{j'}$ заменио тај пакет за a_j распон би се сигурно смањио или бар остао исти (јер би последњи узети пакет тада био неки пакет $a_{j''}$, за $j'' < j'$, а пошто је низ сортиран неопадајуће, важи да је $a_{j''} \leq a_j$, па и $a_{j''} - a_i \leq a_{j'} - a_i$. Даљим заменама истог типа можемо доћи до тога да су сви узети пакети узастопни, а да је распон мањи или једнак полазном, што је у контрадикцији са тим да је распон полазног скупа узетих пакета строго мањи од распона било којег низа k узастопних пакета.

Напомена. Разматрање претходног типа је карактеристично за такозване грамзиве алгоритме, о ком ће више речи бити касније.

На основу претходног јасно је да низ бројева чоколадица у пакетима треба најпре сортирати (најбоље помоћу библиотечке функције `sort` и затим одредити минимум разлика вредности $a_{i+k-1} - a_i$, за $0 \leq i \leq n-k$, коришћењем уобичајеног алгоритма за налажење минимума.

Анализа сложености. Сложеношћу овог алгоритма доминира сложеност корака сортирања, а она је $O(n \log n)$, ако се користе библиотечке имплементације. Након сортирања, минимум се одређује у $n - k$ корака, тј. у линеарној сложености $O(n)$ (када је k мало, број корака може бити веома близак n).

```
// odredjuje najmanju razliku u broju cokoladica unutar odabranih
// paketa ako se bira k paketa medju paketima iz niza a
int minRazlika(vector<int>& a, int k) {
  // broj paketa
  int n = a.size();
  // sortiramo pakete po broju cokoladica
  sort(begin(a), end(a));
  // odredjujemo i najmanju mogucu razliku
  int min = numeric_limits<int>::max();
  for (int i = 0; i + k - 1 < n; i++) {
    int razlika = a[i + k - 1] - a[i];
    if (razlika < min)
      min = razlika;
  }
  return min;
}
```

2.14.3 Свођење на канонски облик

Често имамо потребу да проверимо да ли су два низа елемената једнака, ако се занемари у ком су редоследу елементи наведени. Класичан пример овога је провера да ли се једна реч може добити пермутовањем слова друге (тј. да ли су анаграми). У суштини, ради се о поређењу два мутлискупа елеманата (која су задата низовима). Најбољи начин да се оваква једнакост провери је да се оба низа сведу на неки канонски облик, који неће зависити од редоследа елемената низа. Најједноставнији начин да се такав канонски облик добије је да се елементи низа сортирају пре поређења. Други начин је да се изврши пребројавање свих елемената тј. да се мултискупови представе пресликавањима сваког елемента у његов број појављивања (два таква пресликавања су једнака ако и само ако исти скуп кључева слика у исте вредности).

Задатак: Провера пермутација

Ленка је добила задатак да испрограмира функцију која "меша" дати низ бројева, тј. одређује његову насумичну пермутацију. Ленка је написала своју функцију, покренула је на одређеном броју тест-примера, међутим, када је добила излазне резултате није одмах могла да види да ли је њена функција исправна. Напиши програм који јој помаже тако што учитава почетни низ елемената и низ добијен мешањем и проверава да ли је други низ пермутација првог тј. да ли се могао добити од првог само променом редоследа његових елемената.

Улаз: Са стандардног улаза се уносе два низа природних бројева. За сваки низ се уноси број елемената (највише 50000), а затим и елементи раздвојени са по једним размаком.

Излаз: На стандардни излаз испиши реч `da` ако је други низ добијен мешањем првог, тј. `ne` ако није.

Пример

Улаз	*Излаз*

`5`
`1 3 2 4 3`
`5`
`4 3 2 3 1`
	`da`

Решење

Сортирање

Један од начина да проверимо да ли је један низ пермутација другог је да оба низа доведемо у неку канонску форму, а онда да проверимо да ли су добијене канонске форме једнаке. Најједноставнија канонска форма се добија када се низови сортирају по величини (на пример, неопадајуће). Сортирање се може извршити на неки од начина приказан у задатку Сортирање бројева, најбоље библиотечком функцијом.

Поређење једнакости два низа или вектора се може остварити, пешке, на следећи начин. Прво, да би два низа била једнака потребно је да им број елемената буде једнак. Ову проверу можемо извршити и на самом почетку (пре сортирања). Након тога, линеарном претрагом проверавамо да ли постоји позиција у низовима на којој се налазе два различита елемента. Ако проверу реализујемо у посебној функцији, чим се пронађу два различита елемента функција може да врати вредност `false`. Ако се дође до краја низова, а не нађе се разлика, тада функција може да врати `true`.

Наравно, боље решење је ако се користе библиотечке функције. Поређење једнакости вектора у језику C++ можемо извршити, крајње једноставно, помоћу оператора `==`. Поређење једнакости низова у језику C++ можемо вршити и помоћу функције `equal` која прима итератор на почетак и иза краја првог низа и на почетак другог низа.

Анализа сложености. Низови од n елемената се библиотечком функцијом пореде у времену $O(n \log n)$, док се њихова једнакост проверава у времену $O(n)$. Провером, дакле, доминира време сортирања и алгоритам је сложености $O(n \log n)$.

```
bool jePermutacija(vector<int>& a, vector<int>& b) {
  if (a.size() != b.size())
    return false;
  sort(a.begin(), a.end());
  sort(b.begin(), b.end());
  return a == b;
}
```

Задатак: Анаграм

Дате су две ниске сачињене од малих слова енглеске абецеде, интерпункцијских знакова и размака. Напиши програм који проверава да ли су два дата стринга анаграми, тј. да ли се од прве ниске премештањем слова може добити друга ниска и обрнуто (карактери који нису слова се занемарују).

Улаз: Прва линија стандардног улаза садржи једну ниску, а друга линија садржи другу ниску.

Излаз: На стандардном излазу у једној линији приказати реч `da` ако дате ниске представљају анаграме, у супротном приказати реч `ne`.

Пример1

Улаз

```
panta redovno zakasni
neopravdan izostanak
```

Излаз

```
da
```

Пример2

Улаз

```
oni su skrsili vagu
suvisni kilogrami
```

Излаз

```
ne
```

Решење

Две ниске су анаграми ако се једна може добити пермутовањем редоследа карактера друге, па је централни задатак одредити да ли једна ниска представља пермутацију друге. Провера да ли је један низ пермутација другога описана је у задатку Провера пермутација и у овом задатку се могу применити све технике које су у том задатку описане. Једина разлика у односу на обичну проверу пермутација је то што се приликом провере анаграма не узимају у обзир сви карактери, већ само мала слова.

Сортирање

Један начин да се задатак реши је да се обе ниске сортирају и онда упореде. Сортирање ниски може да се изврши библиотечком функцијом. У језику C++ то је функција `sort`. Међутим, пошто се проверавају само слова, пре сортирања и поређења, из ниски је потребно избацити све карактере који нису слова.

У језику C++ то је могуће урадити функцијом `copy_if`. Она копира елементе из дела низа ограниченог са два итератора у део низа (истог или неког другог), почевши од датог итератора. При томе се претпоставља да ће од позиције тог итератора до краја низа бити довољно алоцираног простора да се прекопирају сви елементи. Међутим, пошто у овом задатку не знамо унапред колико има малих слова, ниску која ће садржати резултат не можемо унапред алоцирати, већ ћемо је проширивати додавањем једног по једног елемената на крај (за шта уместо итератора на почетак алоцираног низа тј. ниске наводимо `back_inserter` направљен од итератора који указује на почетак празне ниске).

Проверу да ли је слово мало вршимо глобалном библиотечком функцијом `islower`.

Анализа сложености. Сложеност овог алгоритма зависи од сложености сортирања и ако се користи библиотечка функција сортирања износи $O(n \log n)$. Наиме, копирање ниски (које је неопходно, да се сортирањем не би покварили оригиналне ниске, али и да би се филтрирала само мала слова) и поређење једнакости након сортирања су линеарне сложености, па сортирање узима највише времена.

```
bool anagram(const string& s1, const string& s2) {
  string slova1, slova2;
  copy_if(begin(s1), end(s1), back_inserter(slova1), ::islower);
  copy_if(begin(s2), end(s2), back_inserter(slova2), ::islower);
  sort(begin(slova1), end(slova1));
  sort(begin(slova2), end(slova2));
```

```
  return slova1 == slova2;
}
```

2.14.4 Остале примене сортирања

Наведени примери сортирања сигурно не исцрпљују све употребе сортирања. У наставку ће бити приказано још неколико задатака који могу ефикасно бити решени захваљујући примени сортирања.

Задатак: Хиршов h-индекс

Рангирање научника врши се помоћу статистике која се назива *Хиршов индекс* (скр. *h-индекс*). H-индекс научника је највећи број h такав да научник има бар h радова од којих сваки има бар h цитата.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 5 \cdot 10^4$) који представља број радова научника а затим и n природних бројева који представљају број цитата (између 0 и 10^6) за сваки од тих n радова.

Излаз: На стандардни излаз исписати један природан број који представља h-индекс научника.

Пример

Улаз	*Излаз*
`8`	`5`
`3 5 12 7 5 9 0 17`	

Објашњење

Постоји тачно 5 радова са бар 5 цитата (5, 12, 9, 7, 17). Преостали радови имају 3, 5 и 0 цитата, тако да не постоји 6 радова са бар 6 цитата.

Решење

Груба сила

H-индекс се може дефинисати као највећи број h такав да постоји бар h радова који имају бар h цитата. Могуће вредности за h индекс су бројеви између 0 и n. Основни начин да одредимо h индекс је да употребимо линеарну претрагу и да за сваку вредност h између 0 и n проверимо да ли постоји бар h радова са бар h цитата тј. да ли је $B_h \geq h$, где B_h представља број радова са h или више цитата. Анализираћемо разне вредности h, на пример, од n па наниже, све док не наиђемо на прву вредност h за коју постоји бар h радова са h цитата тј. за коју је је $B_h \geq h$ и та вредност h представљаће тражени h-индекс.

За сваки број h израчунаћемо вредност B_h. Наиван начин да се то уради је да се да се прође кроз низ бројева цитата c (који не мора бити сортиран) и да се одреди број елемената низа који су већи или једнаки h. У језику C++ пребројавање елемената у низу који задовољавају неки услов можемо урадити библиотечком функцијом `count_if`, којој се прослеђују итератори који ограничавају распон који се претражује и функција (често анонимна, ламбда функција) која проверава да ли појединачни елемент задовољава услов.

Анализа сложености. Пошто се бројање радова за сваку вредност h изводи изнова и захтева линеарну сложеност $O(n)$, а пошто је број различитих вредности h у најгорем случају једнак n, укупна сложеност најгорег случаја овог приступа је $O(n^2)$.

```
int hIndeks(vector<int>& broj_citata) {
  // smanjujemo h-indeks sve dok je broj clanaka koji imaju bar
  // h-indeks citata manji od vrednosti h-indeksa
  int h_indeks = broj_citata.size();
  while (count_if(begin(broj_citata), end(broj_citata),
                  [h_indeks](int c) {
                    return c >= h_indeks;
                  }) < h_indeks)
    h_indeks--;

  return h_indeks;
}
```

Сортирање

Један ефикасан начин одређивања h-индекса може бити заснован на сортирању. На пример, ако сортирамо низ цитата `3 5 12 7 5 9 0 17` нерастуће, добијамо низ `17 12 9 7 5 5 3 0`. Анализирајмо један по један рад у овом низу. Важи да је $c_0 = 17 \geq 1$, па имамо бар један рад са бар једним цитатом, $c_1 = 12 \geq 2$, па имамо бар два рада са бар два цитата, $c_2 = 9 \geq 3$, па имамо бар 3 рада са бар 3 цитата, $c_3 = 7 \geq 4$, па имамо бар 4 рада са бар 4 цитата, $c_4 = 5 \geq 5$, па имамо бар 5 радова са бар 5 цитата, али не и $c_5 = 5 \geq 6$, па немамо бар 6 радова са бар 6 цитата.

Дакле, ако радове сортирамо у нерастући низ c по броју цитата, пошто се позиције броје од 0, за радове који задовољавају h-услов важи да је $c_h \geq (h+1)$ тј. $c_h > h$. H-индекс можемо израчунати као најмањи број h такав да је $c_h \leq h$.

Заиста ако је неки рад на позицији h у сортираном низу и ако он има c_h цитата, знамо да постоји бар $h+1$ рад који има бар c_h цитата (јер он има бар c_h цитата и испред њега постоји тачно h радова који имају бар онолико цитата колико и он, јер је низ нерастући). У нерастуће сортираном низу проверавамо све индексе h од 0 до $n-1$, све док не наиђемо на први за који важи да је $c_h \leq h$. Када се то деси, знамо да постоји тачно h радова који имају бар h цитата, јер је за претходну позицију важило да је $c_{h-1} > h-1$, тј. $c_{h-1} \geq h$, па је постојало тачно h радова са бар c_h цитата, а то је бар h цитата. Такође, не постоји $h+1$ рад са бар $h+1$ цитатом, јер сви радови од позиције h до краја низа имају строго мање од $h+1$ цитата, а испред њих постоји само h радова.

Анализа сложености. Сложеност овог алгоритма долази пре свега од корака сортирања и када се сортира библиотечком функцијом једнака је $O(n \cdot \log n)$ (наиме, након сортирања индекс се одређује једним проласком кроз низ, у највише $O(n)$ додатних корака).

```
int hIndeks(vector<int>& broj_citata) {
  // sortiramo radove na osnovu broju citata, opadajuce
  sort(broj_citata.begin(), broj_citata.end(), greater<int>());

  // Ako je h-indeks jednak h, tada h-ti po redu casopis ima bar h
  // citata. Trazimo najveci broj koji zadovoljava taj uslov.
  int h_indeks = 0;
  while (h_indeks < broj_citata.size() && broj_citata[h_indeks] > h_indeks)
    h_indeks++;

  return h_indeks;
}
```

Види другачија решења овог задатка.

2.15 Бинарна претрага

Ако немамо никакве информације о редоследу елемената у низу, једини начин да проверимо да ли се у њему налази неки елемент је да употребимо линеарну претрагу и да редом проверавамо један по један елемент низа. У најгорем случају сваки елемент низа мора бити прегледан, па је сложеност таквог приступа $O(n)$, где је n број елемената низа.

Ако је низ елемената сортиран, претрагу је могуће извршити много ефикасније, у сложености $O(\log n)$, коришћењем алгоритма **бинарне претраге**, којим се, услед сортираности низа, одсеца значајан део простора претраге и тако добија на ефикасности. Она се може применити на много сродних проблема.

У основној варијанти, бинарна претрага (БП) служи да се провери да ли сортирани низ елемената садржи неку дату вредност.

Поред овога, БП се може употребити да се пронађе први елемент у сортираном низу који је (било строго, било нестрого) већи или мањи од датог.

У свом најопштијем облику бинарна претрага се користи да се у низу пронађе преломна тачка тј. први елемент који задовољава неки услов (под претпоставком да су елементи поређани тако да у низу прво иду сви елементи који тај услов не задовољавају, а затим они који тај услов задовољавају).

Бинарну претрагу ћемо употребљавати и за оптимизацију, тј. да пронађемо најмању или највећу вредност, која задовољава одређени услов.

2.15.1 Бинарна претрага елемента у низу

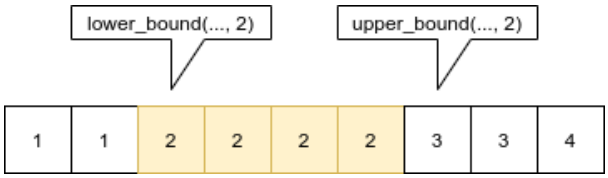
Алгоритам бинарне претраге вредности у низу одговара оном који се може применити у игри погађања непознатог броја и који је заснован на полoвљењу интервала. Основна идеја је да се тражени елемент пореди са средишњим елементом у низу. Ако је тражени елемент мањи од средишњег, пошто је низ сортиран, знаћемо да је мањи и од свих елемената десно од тог средишњег, па тај део низа можемо елиминисати (можемо начинити одсецање у претрази) и претрагу можемо наставити само у левој половини низа. Симетрично, ако је тражени елемент већи од средишњег, због сортираности је већи и од свих елемената лево од средишњег и лева половина низа може бити елиминисана из даље претраге. На крају, ако елемент није ни мањи ни већи од средишњег, онда му је једнак и пронађен је у низу. Приметимо да у основи алгоритма бинарне претраге лежи техника одсецања, која је оправдана тиме што је низ сортиран.

Пошто се у сваком кораку претраге дужина низа дупло смањује, за претрагу целог низа довољно је $O(\log n)$ корака, где је n дужина низа. Наиме, претрага у најгорем случају траје све док се половљењем низ не испразни. Дужина низа након k корака половљења је отприлике $\frac{n}{2^k}$. Низ ће се испразнити када је $\frac{n}{2^k} < 1$, тј. када је $n < 2^k$, тј. када је $k > \log_2 n$.

Слично као и за сортирање, већина програмских језика пружа готове библиотечке функције за бинарну претрагу.

У језику C++ функција `binary_search` проверава да ли дати распон елемената (задат помоћу два итератора) садржи задату вредност (функција враћа `true` ако и само ако се тражени елемент налази унутар задатог распона). Тако се провера да ли се дати елемент `x` налази унутар сортираног вектора `a` може извршити помоћу `binary_search(begin(a), end(a), x)`.

Поред ове, постоје још три функције које врше одређене варијације бинарне претраге. Ако је потребно пронаћи све елементе једнаке датом, можемо користити функцију `equal_range` (са истим параметрима као `binary_search`). Она враћа пар итератора који ограничавају распон елемената једнаких датом (први итератор указује на први елемент једнак траженој вредности, а други на позицију непосредно иза последњег елемента једнаког траженој вредности). Функција `lower_bound` враћа први од та два итератора (тј. први елемент који је већи или једнак од тражене вредности), а функција `upper_bound` враћа други од њих (тј. први елемент који је строго већи од тражене вредности).



Слика 2.22: lower_bound и upper_bound

О њима и њиховој употреби ће бити више речи у наредним поглављима.

У свим библиотечким функцијама за бинарну претрагу, ако се не зада другачије, подразумева се да је низ сортиран у односу на подразумевани поредак елемената (неопадајући нумерички ако су бројеви у питању, тј. неопадајући абецедни лексикографски ако су ниске у питању). Поредак се може задати или променити на сличан начин као код функција за сортирање (о чему ће бити више речи касније).

Задатак: Провера бар-кодова

У продавници се налази пуно врста производа и познати су њихови бар-кодови. Произвођач жели да сазна колико се врста његових производа продаје у тој продавници. Ако је списак свих кодова производа у продавници дат у сортираном облику, а списак свих кодова производа произвођача је достављен несортиран, напиши програм који одређује тражени број.

Улаз: Са стандардног улаза учитава се број n ($1 \leq n \leq 50000$), а n природних бројева (највише шестоцифрених), раздвојених размацима. Ти бројеви представљају бар-кодове производа у продавници и сортирани су растуће. Након тога се до краја улаза учитавају бар-кодови производа које је произвођач доставио (највише шестоцифрени природни бројеви, сваки у посебном реду).

Излаз: На стандардни излаз исписати број производа произвођача који се већ продају у продавници.

Пример

Улаз	*Излаз*
5 1 3 5 6 7 2 3 4 5 8	2

Решење

Линеарна претрага

Наиван начин да проверимо да ли елемент постоји у низу је примена алгоритма линеарне претраге. Линеарна претрага може бити било имплементирана ручно, било реализована помоћу библиотечких функција. У језику C++ функцијом `find` можемо проверити да ли низ садржи дати елемент.

Анализа сложености. Пошто се у најгорем случају линеарном претрагом (било библиотечком, било ручно-имплементираном) анализира сваки елемент низа, сложеност претраге једног елемента је $O(n)$. Ако претпоставимо да се претражује m елемената, укупна сложеност алгоритма је $O(mn)$.

```
// funkcija proverava da li se u datom sortiranom vektoru a nalazi
// element x
bool sadrzi(const vector<int>& a, int x) {
  // proveravamo sve elemente vektora a
  for (int i = 0; i < a.size(); i++)
    // nasli smo element x na poziciji i
    if (a[i] == x)
      return true;
  // element x nije nadjen
  return false;
}

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // brojac pojavljivanja elemenata
  int broj = 0;
  // ucitavamo element po element do kraja ulaza
  int x;
  while (cin >> x) {
    // ako je ucitani element sadrzan u nizu a, uvecavamo brojac
    if (sadrzi(a, x))
      broj++;
  }
  // ispisujemo rezultat
  cout << broj << endl;
  return 0;
}
```

Бинарна претрага

Чињеница да је низ сортиран нам даје начин да задатак решимо много ефикасније, применом бинарне пре-

траге.

Бибилотечке функције

У језику C++ функција `binary_search` изводи бинарну претрагу неког сегмента (низа или вектора). Аргументи су два итератора (један на први елемент сегмента који се претражује, а други непосредно иза последњег елемента) и елемент који се тражи. Функција враћа `bool` чија је вредност `true` ако и само ако елемент постоји у низу.

Анализа сложености. Библиотечка функција гарантује да ће сортиран низ од n елемената бити претражен у сложености $O(\log n)$ (па се m елемената независно претражује у $O(m \log n)$ корака).

```
// broj onih koji postoje u nizu
int broj = 0;
// ucitavamo broj po broj do kraja ulaza
int x;
while (cin >> x) {
  // ako je broj sadrzan u nizu, uvecavamo brojac
  if (binary_search(a.begin(), a.end(), x))
    broj++;
}
// ispisujemo rezultat
cout << broj << endl;
```

Итеративна имплементација

Алгоритам бинарне претраге вредности у низу одговара оном који се може применити у игри погађања непознатог броја и који је заснован на половљењу интервала.

Претпоставимо да желимо проверити да ли се елемент x налази у низу a између позиција l и d тј. да ли се налази у затвореном интервалу позиција $[l, d]$. Ако је интервал празан, он не садржи елемент x. У супротном пронађимо средину овог интервала s.

- Ако је $x < a[s]$, пошто је низ сортиран неопадајуће, важи да је x мањи и од свих елемената који су десно од s. Зато претрагу можемо наставити у интервалу $[l, s-1]$.
- Слично, ако је $x > a[s]$, пошто је низ сортиран неопадајуће, важи да је x веће од свих елемената лево од s тако да претрагу можемо наставити само у интервалу $[s+1, d]$.
- На крају, ако је $x = a[s]$ тада смо елемент пронашли у низу и то на позицији s. Претрага се може прекинути и када се интервал који се претражује испразни (што ће се десити ако је $l > d$).

Алгоритам можемо имплементирати итеративно. Промењиве l и d иницијализујемо на 0 и $n-1$, затим у петљи која се извршава док је $l \leq d$ проналазимо средину s интервала $[l, d]$ и поредимо $a[s]$ са x. Ако је $x < a[s]$ тада d постављамо на $s-1$, ако је $x > a[s]$ тада l постављамо на $s+1$, а ако је $x = a[s]$ прекидамо функцију (и петљу) информишући позиваоца да је елемент нађен на позицији s. Ако се петља заврши, елемент не постоји у низу.

Доказ коректности. Докажимо формално коректност овог алгоритма. Инваријанта петље је да су:

- сви елементи у интервалу $[0, l)$ строго мањи од x,
- сви елементи у интервалу (d, n) строго већи од x.

Уз то важи и $0 \leq l \leq d+1 \leq n$.

Иницијализацијом $l = 0$ и $d = n-1$, инваријанта је задовољена.

Претпоставимо да инваријанта важи при уласку у петљу, након провере услова $l \leq d$. Тада израчунавамо средину интервала s (за њу сигурно важи $l \leq s \leq d$).

- Ако је $x < a_s$, тада је $l' = l$, $d' = s-1$. Пошто је $a_s > x$, услед сортираности низа у неопадајућем поретку, већи од x су и сви елементи на позицијама из интервала $(d', n) = (s-1, n) = [s, n)$. Пошто се вредност промењиве `l` није променила, сви елементи на позицијама $[0, l')$ су сигурно строго мањи од x (што знамо да важи на основу претпоставке).
- Ако је $x > a_s$, тада је $l' = s+1$, $d = d$. Пошто је $a_s < x$, услед сортираности низа у неопадајућем поретку, мањи од x су и сви елементи на позицијама из интервала $[0, l') = [0, s+1) = [0, s]$. Пошто се

вредност променљиве d није променила, сви елементи на позицијама (d', n) су сигурно строго већи од x (што знамо да важи на основу претпоставке).

- Ако је $x = a_s$, пронађен је тражени елемент и функција коректно потврђује да низ садржи елемент x.

Када се петља заврши важи инваријанта, али услов $l \leq d$ није испуњен. Пошто је $l \leq d+1$, важи да је $l = d+1$. Стога су сви елементи у интервалу $[0, l)$ строго мањи од x а сви елементи у интервалу $(d, n) = [d+1, n) = [l, n)$ строго већи од x. Дакле, сви елементи низа су или строго мањи или строго већи од x и стога низ не садржи елемент x.

Алгоритам се сигурно зауставља, јер се у сваком кораку петље ширина интервала $[l, d]$, која је једнака $d-l+1$ строго смањује, све док не достигне нулу.

Анализа сложености. Пошто се у сваком кораку интервал $[l, d]$ полови, пошто иницијално крећемо од интервала $[0, n-1]$ и пошто се претрага завршава када се интервал испразни, сложеност претраге једног елемента је $O(\log n)$ (па се m елемената независно претражује у $O(m \log n)$ корака).

```
// funkcija proverava da li se u datom sortiranom nizu a duzine n
// nalazi element x
bool sadrzi(int a[], int n, int x) {
  // petrazujemo da li se element nalazi u intervalu [l, d]
  int l = 0, d = n - 1;
  // dok god taj interval nije prazan
  while (l <= d) {
    // nalazimo sredinu intervala
    int s = l + (d - l) / 2;

    // ako je x manji od srednjeg on se moze nalaziti samo u intevalu
    // [a, s-1] (jer je niz sortiran)
    if (x < a[s])
      d = s - 1;
    // ako je x veci od srednjeg on se moze nalaziti samo u intevalu
    // [s+1, d] (jer je niz sortiran)
    else if (x > a[s])
      l = s + 1;
    else
      // nasli smo element x na poziciji s
      return true;
  }
  // element ne postoji u nizu
  return false;
}
```

Види другачија решења овог задатка.

Задатак: Број парова датог збира

Дат је цео број s и низ различитих целих бројева. Написати програм којим се одређује број парова у низу који имају збир једнак датом броју s.

Улаз: У првој линији стандардног улаза налази се цео број s (број из интервала $[0, 10^6]$), у другој линији налази се број елемената низа n ($1 \leq n \leq 50000$), а у следећих n линија налази се редом елементи низа (бројеви из интервала $[0, 10^6]$).

Излаз: На стандардном излазу приказати број парова различитих елемената низа чији је збир једнак броју s.

Пример

Улаз	*Излаз*
5 6 1 4 3 6 -1 5	2

Решење

Груба сила

Задатак можемо решити анализирајући збир елемената сваког пара у низу a. Различити парови низа a су облика (a_i, a_j), при чему је $i < j$ (у супротном би се исти пар рачунао два пута). Зато i узима вредности од 0 до $n-2$, а j узима вредности од $i+1$ до $n-1$. Парове можемо набројати помоћу две угнежђене петље, тако да у телу унутрашње петље проверавамо да ли је збир текућа два елемента једнак s и ако јесте, увећавамо бројач парова (пре петљи иницијализован на нулу).

```
int brojParovaDatogZbira(const vector<int>& a, int s) {
  int n = a.size();
  int brojParova = 0;
  for (int i = 0; i < n - 1; i++)
    for(int j = i + 1; j < n; j++)
      if (a[i] + a[j] == s)
        brojParova++;
  return brojParova;
}
```

Анализа сложености. Сложеност овог наивног приступа одговара броју парова наведеног облика, што је $O(n^2)$.

Бинарна претрага

Уместо посебне колекције скупа, скуп елемената може бити репрезентован сортираним низом који претражујемо бинарном претрагом. Тај алгоритам је детаљно објашњен у задатку Провера бар-кодова. Да бисмо могли да применимо овај приступ, потребно је да претходно низ a сортирамо у растућем поретку. Елемент $a_j = s - a_i$ можемо тражити у низу почев од позиције $i+1$, јер на тај начин нећемо исти пар бројати два пута. Такође приметимо да је важан услов, дат у задатку, да су елементи низа различити, јер класичном бинарном претрагом налазимо једно појављивање елемента у низу или установимо да тај елемент не постоји, а нама се тражи број парова па је важно колико пута се неки елемент појављује у низу а не само да ли се појављује.

У језику C++ можемо да проверимо да ли тражена вредност постоји у сортираном низу помоћу функције `binary_search` (која враћа логичку вредност).

```
int brojParovaDatogZbira(const vector<int>& a, int s) {
  // pravimo sortiranu kopiju niza a
  auto as = a;
  sort(begin(as), end(as));
  // za svaki element as[i] trazimo s-as[i] u delu niza iza njega
  int brojParova = 0;
  for (int i = 0; i < as.size() - 1; i++)
    if (binary_search(next(begin(as), i + 1), end(as), s - as[i]))
      brojParova++;
  return brojParova;
}
```

Анализа сложености. Сложеност бинарне претраге је $O(\log n)$, у зависности од дужине дела низа који се претражује, а пошто ми претрагу вршимо за сваки од n елемената низа, може се показати да ће укупна сложеност бити $O(n \log n)$. Пошто се не претражује увек цео низ, већ само део од текућег елемента до краја низа, сложеност провере ће заправо бити $O(\log(n-1) + \log(n-2) + \ldots + \log 1)$, што је $O(\log(n-1)!)$ и за то је могуће показати да је $O(n \log n)$.

Бинарна претрага би се могла унапредити тако што се би се у сваком наредном кораку вршила претрага за вредношћу $s - a_{i+1}$ само до позиције првог елемента који је био већи или једнак вредности $s - a_i$. Наиме, пошто је $a_{i+1} > a_i$, тај елемент и сви елементи иза њега су сигурно строго већи од вредности $s - a_{i+1}$ (која је строго мања од $s - a_i$). Претрагу можемо прекинути када се лева и десна граница претраге сустигну (тј. када се та десна позиција повуче на лево, све до текуће вредности i).

Анализа сложености. Овом модификацијом се не би добило асимптотски значајно убрзање, па је нећемо имплементирати.

Види другачија решења овог задатка.

Задатак: i-ти на месту i

Напиши програм који проверава да ли у строго растућем низу елемената постоји позиција i таква да се на позицији i налази вредност i тј. да важи да је $a_i = i$ (позиције се броје од нуле).

Улаз: Са стандардног улаза се уноси број n ($0 \leq n \leq 10^5$), а затим и строго растући низ од n целих бројева (сваки у посебном реду).

Излаз: На стандардни излаз исписати индекс i такав да је $a_i = i$ или текст `nema` ако такав индекс не постоји у низу. Ако у низу постоји више таквих индекса исписати најмањи од њих.

Пример

Улаз	*Излаз*

`6`
`-3`
`-1`
`1`
`3`
`5`
`7`
	`3`

Решење

Линеарна претрага

Директан начин да се задатак реши је да се употреби линеарна претрага и да се позиције проверавају редом, од 0 до $n - 1$ све док се не пронађе прва позиција која задовољава услов или док се не дође до краја низа.

Анализа сложености. Сложеност линеарне претраге је $O(n)$.

```
int iti_na_mestu_i(const vector<int>& a) {
  for (int i = 0; i < a.size(); i++)
    if (a[i] == i)
      return i;
  return -1;
}
```

Бинарна претрага трансформисаног низа

Размотримо низ `-10 -4 1 3 4 9 11`. Елемент `-10` је мањи од своје позиције 0 за 10. Елемент `-4` је мањи од своје позиције `1` за `5`, елемент `1` је мањи од своје позиције 2 за 1. Елементи `3` и `4` су једнаки својим позицијама. Елемент `9` је већи од своје позиције `5` за `4` док је елемент `11` већи од своје позиције `6` за `5`. Примећујемо одређену монотоност у овом низу, што није случајно. Заиста, ако је $a_i = i$, тада је $a_i - i = 0$. Покажимо да је низ $a_i - i$ неопадајући. Посматарајмо два елемента a_i и a_j на позицијама на којима је $0 \leq i < j$. Пошто је низ a строго растући, важи да је $a_{i+1} > a_i$, па је $a_{i+1} \geq a_i + 1$. Слично је $a_{i+2} > a_{i+1}$, па је $a_{i+2} \geq a_{i+1} + 1 \geq a_i + 2$. Настављањем овог резона важи да је $a_j \geq a_i + (j - i)$. Зато је $a_j - j \geq a_i - i$. Решење, дакле, можемо одредити тако што бинарном претрагом проверимо да ли неопадајући низ $a_i - i$ садржи нулу и ако садржи, тада је решење позиција на којој се та нула налази.

Један од најлакших начина да реализујемо бинарну претрагу је да употребимо библиотечку функцију. Пошто нам је потребна прва позиција нуле у трансформисаном низу, не можемо употребити функцију `binary_search`, већ морамо употребити функцију `lower_bound`.

Анализа сложености. Сложеност бинарне претраге је $O(\log n)$, међутим, временом доминира учитавање и трансформисање низа које захтева $O(n)$ корака.

```
int iti_na_mestu_i(const vector<int>& a) {
  int n = a.size();
  // pripremamo ga za pretragu a[k] = k akko je a[k] - k = 0 tako da
  // umesto niza a, pretrazujemo niz a[i] - i (koji je neopadajuci)
  vector<int> b(n);
  for (int i = 0; i < n; i++)
    b[i] = a[i] - i;

  // trazimo poziciju nule u transformisanom nizu tako sto pronalazimo
  // poziciju prvog elementa koji je >= 0
  auto it = lower_bound(b.begin(), b.end(), 0);

  // ako takav element postoji i ako je jednak nuli
  if (it != b.end() && *it == 0)
    // pronasli smo element i izracunavamo njegovo rastojanje od pocetka niza
    return distance(b.begin(), it);
  else
    // u suprotnom element ne postoji u nizu
    return -1;
}
```

2.15.2 Бинарна претрага преломне тачке

У свом најопштијем облику, бинарна претрага се може формулисати на следећи начин. Размотримо низ елемената такав да су елементи подељени у две групе, на основу неког својства P. Елементи у почетном делу низа су сви такви да немају то својство P, а елементи у завршном делу низа су сви такви да имају својство P. Низ је, дакле, облика `----++++++++`, где су са `-` означени елементи који немају, а са `+` елементи који имају својство P. Могућа је и ситуација у којој је нека од група празна. Својство P може бити сасвим произвољно. На пример, у сортираном низу бројева можемо посматрати својство *већи је или једнак* X за неку дату вредност X. Тада се у првом делу низа налазе елементи који нису већи или једнаки X, тј. строго су мањи од X, док се иза њих налазе елементи који имају својство, тј. већи су или једнаки X. Даље, на пример, низ ученика може бити организован тако да су прво наведени дечаци, а затим девојчице.

Бинарна претрага нам може помоћи да ефикасно одредимо *преломну тачку*, тј. место где престаје једна и почиње друга група елемената. То може бити или позиција последњег елемента који нема својство P или првог елемента који има својство P. Ако сви елементи низа имају својство P, тада претрага за позицијом последњег елемента низа који нема својство треба да врати -1. Ако ниједан елемент низа нема својство P, тада претрага за позицијом првог елемента низа који има својство P треба да врати дужину низа. Познавање преломне тачке нам омогућава и да ефикасно одговоримо на питање колико је елемената у свакој групи (колико елемената низа нема, а колико елемената низа има својство P).

Класична бинарна претрага се лако формулише као претрага преломне тачке. Ако у низу пронађемо позицију првог елемента који је већи или једнак траженој вредности X, тада можемо проверити да ли је та позиција унутар низа (строго мања од дужине низа) и да ли се на њој налази елемент X - ако је то испуњено елемент постоји у низу, а у супротном не постоји.

У наставку ће кроз низ задатака бити приказан алгоритам бинарне претраге преломне тачке у низу. У првим примерима у низу бројева ћемо тражити позицију првог елемента који је (строго или нестрого) већи или мањи од дате вредности, док ћемо у наредним примерима посматрати и другачије низове, подељене у односу на неко својство P.

Позиција преломне тачке може бити пронађена и уз помоћ инвентивне употребе функција `lower_bound` и `upper_bound`.

Задатак: Провера бар-кодова

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Бинарна претрага првог елемента који је већи или једнак од траженог

Један начин да се реализује бинарна претрага којом се проверава да ли елемент постоји у низу је да се пронађе позиција првог елемента који је већи или једнак траженом елементу x. Број x се јавља у низу ако и само у низу постоји елемент који је већи или једнак x и први такав елемент је управо x.

Опишимо имплментацију без употребе библиотечких функција. Први елемент који је већи или једнак од елемента x можемо наћи на следећи начин.

Доказ коректности. Уведимо промењиве l и d и наметнимо услов (инваријанту) да све време током претраге важи $0 \leq l \leq d + 1 \leq n$, и да су

- елементи низа a на позицијама из интервала $[0, l)$ мањи од x,
- елементи на позицијама из интервала (d, n) већи или једнаки x.

Елементима на позицијама из интервала $[l, d]$ статус још није познат. Овај услов ће бити иницијализован ако се промењива l иницијализује на нулу, а промењива d на вредност $n - 1$.

Нека s представља средину интервала $[l, d]$. Важи $l \leq s \leq d$.

- Ако је елемент низа a на позицији s већи или једнак вредности x тада су, услед сортираности низа у неопадајућем поретку, и сви елементи интервала $[s, n)$ су већи или једнаки x. Зато можемо вредност d поставити на $s - 1$ и инваријанта ће остати да важи. Заиста, важи да је $d' = s - 1$, па су елементи из интервала $(d', n) = (s - 1, n) = [s, n)$ већи или једнаки x, док су сви елементи из интервала $[0, l') = [0, l)$ строго мањи од x, што знамо на основу претпоставке. Услов $0 \leq l' \leq d' + 1 \leq n$, еквивалентан је услову $0 \leq l \leq s \leq n$, што сигурно важи, јер је $l \leq s \leq d$ и $0 \leq l \leq d + 1 \leq n$.
- Ако је елемент низа a на позицији s строго мањи од вредности x такви су, услед сортираности низа у неопадајућем поретку, и сви елементи из интервала $[0, s]$. Зато можемо вредност l поставити на $s + 1$ и инваријанта ће остати да важи. Заиста, важи да је $l' = s + 1$, па су сви елементи из интервала $[0, l') = [0, s + 1) = [0, s]$ строго мањи од x, док су сви елементи из интервала $(d', n) = (d, n)$ већи или једнаки од x, што знамо на основу претпоставке. Услов $0 \leq l' \leq d' + 1 \leq n$, еквивалентан је услову $0 \leq s + 1 \leq d + 1 \leq n$, што сигурно важи, јер је $l \leq s \leq d$ и $0 \leq l \leq d + 1 \leq n$.

Претрагу вршимо све док постоје елементи непознатог статус, тј. док је интервал $[l, d]$ непразан, односно док је $l \leq d$. У тренутку када је $l > d$, када се претрага заврши, важи да је $l = d + 1$. На основу инваријанте знамо да се први елемент већи или једнак x налази на позицији $l = d + 1$, јер су сви елементи у интервалу $(d, n) = [d+1, n)$ већи или једнаки од x. Изузетак је случај када је $l = n$, када су сви елементи низа a строго мањи од x.

Тражени елемент x постоји у низу ако и само ако је $l = d + 1 < n$ и ако је $a_l = x$.

Алгоритам се сигурно зауставља јер се у сваком кораку ширина интервала $[l, d]$ која је једнака $d-l+1$ строго смањује, док не достигне нулу.

Напомена. Приметимо да овим алгоритмом пролазимо позицију првог појављивања елемента x у низу, док алгоритмом који проверава једнакост током претраге проналазимо било коју позицију. Пошто је често потребно наћи баш прво појављивање елемента, овај алгоритам је јасно у предности.

Анализа сложености. Пошто се у сваком кораку интервал $[l, d]$ полови, пошто иницијално крећемо од интервала $[0, n - 1]$ и пошто се претрага завршава када се интервал испразни, број потребних корака да се то догоди је $O(\log n)$. Пошто се независно претражује m елемената, укупна сложеност је $O(m \log n)$. Приметимо да за разлику од варијанте у којој се унутар тела петље проверава и услов једнакости, за шта су потребна два поређења, у овом алгоритму се врши само једно поређење у телу петље, па је константни фактор мало мањи (што је често занемариво).

```
// funkcija proverava da li se u datom sortiranom nizu a duzine n
// nalazi element x
bool sadrzi(int a[], int n, int x) {
  // trazimo poziciju prvog elementa u nizu a koji je veci ili jednak x

  // [0, l) - elementi strogo manji od x
  // (d, n) - elementi veci ili jednaki x
  // [l, d] - nepoznati elementi
  int l = 0, d = n - 1;
  // dok god taj interval nije prazan
  while (l <= d) {
```

```
    // nalazimo sredinu intervala
    int s = l + (d - l) / 2;

    if (a[s] >= x)
      // posto je niz sortiran svi posle pozicije s-1 su veci ili jednaki x
      d = s - 1;
    else
      // posto je niz sortiran svi pre pozicije s+1 su strogo manji od x
      l = s + 1;
  }
  // prvi veci ili jednak nalazi se na poziciji d+1 = l

  // x je u nizu ako i samo ako u nizu postoji element koji je veci
  // ili jednak x i ako je prvi takav element upravo x
  return l < n && a[l] == x;
}
```

Задатак: Први који није дељив

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Линеарна претрага

Наивно решење може бити засновано на примени линеарне претраге (бројању елемената филтриране серије) да би се одредило колико у низу постоји елемената дељивих са учитаним делиоцем.

Анализа сложености. Ако низ има n елемената, а постоји m делилаца, сложеност овог решења је $O(mn)$. Приметимо да ово решење ни на који начин не користи особину низа да прво иду елементи који су дељиви, а онда елементи који нису дељиви датим делиоцем.

```
int prviKojiNijeDeljiv(const vector<long long>& a, long long d) {
    auto deljiv = [d](long long x) {return x % d == 0;};
    return count_if(begin(a), end(a), deljiv);
}
```

Библиотечке функције

Задатак можемо решити и помоћу библиотечких функција за бинарну претрагу.

Да бисмо нашли први елемент који задовољава неки услов, у језику C++ можемо употребити функцију `upper_bound`, на мало необичан начин. У случају претраге преломне тачке битни су нам само елементи низа, а не елемент који се тражи (јер заправо не тражимо никакав конкретан елемент унутар низа). Зато као елемент који тражимо можемо навести било шта (на пример, нулу). Кључно је дефинисати функцију поређења (која се прослеђује као последњи аргумент функцији `upper_bound`), тако да враћа информацију о томе да су елементи који не задовољавају услов мањи од траженог, док елементи који задовољавају услов нису мањи од траженог. Функција поређења, дакле, треба само да анализира свој други елемент и да врати информацију о томе да ли он задовољава услов (у нашем примеру, тражимо први елемент који није дељив бројем d и то је услов који се проверава у склопу функције поређења).

Рецимо да бисмо могли употребити и функцију `lower_bound`, али би тада у функцији поређења било потребно разменити редослед аргумената (њој је тражена вредност увек други аргумент) и негирати услов.

Анализа сложености. Пошто је сложеност библиотечке функције бинарне претраге $O(\log n)$, сложеност одговора на свих m упита је $O(m \log n)$.

```
int prviKojiNijeDeljiv(const vector<long long>& a, long long d) {
    auto it = upper_bound(begin(a), end(a), 0,
                          [d](long long _, long long x) {
                            return x % d != 0;
                          });
    return distance(begin(a), it);
```

```
}
```

Задатак: Минимум ротираног сортираног низа

Сортирани низ целих бројева у коме су сви елементи различити је ротиран за k места улево и тиме је добијен циклични низ који задовољава услов да је $x_k < x_{k+1} < \ldots < x_{n-1} < x_0 < \ldots < x_{k-1}$. Један такав низ је, на пример, `11 13 15 19 24 1 3 8 9`. Напиши програм који проналази најмањи елемент таквог низа. Потруди се да се након учитавања елемената минимум пронађе у временској сложености $O(\log n)$.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 50000$), а затим n елемената низа (n целих бројева развојених са по једним размаком).

Излаз: На стандардни излаз исписати најмањи елемент низа.

Пример

Улаз	*Излаз*
`9` `11 13 15 19 24 1 3 8 9`	`1`

Решење

Линеарна претрага

Задатак можемо решити класичним алгоритмом или библиотечком функцијом за проналажења минимума. У језику C++ можемо употребити функцију `min_element`.

Анализа сложености. Овај алгоритам захтева пролазак кроз све елементе низа, па је сложености $O(n)$.

```
int minRotiranogSortiranog(const vector<int>& a) {
  return *min_element(begin(a), end(a));
}
```

Бинарна претрага

Поређење са првим елементом низа

Боље решење се може добити бинарном претрагом. Након ротације сви елементи у почетном делу низа су већи од или једнаки почетном, а онда у завршном делу низа иду сви елементи који су строго мањи од почетног. Најмањи елемент који тражимо је први елемент у низу који је строго мањи од почетног и њега можемо наћи бинарном претрагом. Треба обратити пажњу на специјални случај у коме је низ ротиран за 0 места и тада не постоји елемент који је строго мањи од почетног. Бинарна претрага ће тада вратити позицију иза краја низа и у том случају најмањи елемент у низу је управо први елемент низа.

Дакле, поново тражимо први елемент који задовољава неки дати услов. Тај поступак је детаљно описан у задатку Први који није дељив. Одржавамо позиције l и d, и инваријанта петље је да су:

- сви елементи испред позиције l (елементи на позицијама из интервала $[0, l)$) већи или једнаки почетном и сортирани су,
- сви елементи иза позиције d (елементи на позицијама из интервала (d, n)) строго мањи од почетног и сортирани су.

Претрага се завршава у тренутку када је $l = d + 1$ и тада важи да су сви елементи иза позиције d тј. сви елементи од позиције $l = d + 1$ па до краја низа строго мањи од почетног елемента низа, док су елементи од почетка низа лево од позиције l већи или једнаки од почетног елемента низа. Ако постоје елементи од позиције l до краја низа, тј. ако је $l < n$, тада су они сигурно мањи од елемената испред позиције l, а пошто су сортирани, најмањи је први од њих, тј. елемент на позицији l. У супротном, ако је $l = n$, тада постоји само леви део низа, тј. сви елементи у низу су већи или једнаки почетном елементу, а пошто је тај део низа сортиран, најмањи елемент је почетни.

Анализа сложености. Сложеност бинарне претраге је $O(\log n)$, међутим, алгоритмом доминира учитавање елемената у низ, које је сложености $O(n)$, па се предности бинарне претраге не могу ефективно осетити.

```
int minRotiranogSortiranog(const vector<int>& a) {
  int l = 0, d = a.size()-1;
  while (l <= d) {
    int s = l + (d-l)/2;
```

```
    if (a[s] < a[0])
      d = s-1;
    else
      l = s+1;
  }

  int min = l < a.size() ? a[l] : a[0];
  return min;
}
```

Поређење са последњим елементом низа

Провера специјалног случаја након претраге се може избећи ако се уместо односа са првим, гледа однос са последњим елементом у низу. Тражимо први елемент који је мањи од или једнак последњем.

Инваријанта овог алгоритма је да су:

- сви елементи испред позиције l (елементи на позицијама из интервала $[0, l)$) строго већи од последњег елемента низа и сортирани су,
- сви елементи иза позиције d (елементи на позицијама из интервала (d, n)) мањи или једнаки од последњег елемента низа и сортирани су.

Када се петља заврши важи да је $l = d + 1$. Зато су сви елементи иза позиције l строго већи од елемената на позицији l. Пошто је део од позиције l до краја сортиран, минимум се налази на позицији l, јер је тај део увек непразан. Заиста, мора да важи да је $l < n$, јер би у супротном последњи елемент био лево од позиције l, што је немогуће, јер се лево од позиције l налазе елементи који су строго већи од последњег.

Анализа сложености. Сложеност бинарне претраге је $O(\log n)$, међутим, алгоритмом доминира учитавање елемената у низ, које је сложености $O(n)$, па се предности бинарне претраге не могу ефективно осетити.

```
int minRotiranogSortiranog(const vector<int>& a) {
  int n = a.size();
  int l = 0, d = n-1;
  while (l <= d) {
    int s = l + (d-l)/2;
    if (a[s] < a[n-1])
      d = s-1;
    else
      l = s+1;
  }
  return a[l];
}
```

Имплементацију можемо једноставно реализовати и инвентивним коришћењем библиотечке функције `upper_bound`. Та техника је описана у задатку Први који није дељив.

Анализа сложености. Сложеност бинарне претраге библиотечком функцијом је $O(\log n)$, међутим, алгоритмом доминира учитавање елемената у низ, које је сложености $O(n)$, па се предности бинарне претраге не могу ефективно осетити.

```
int minRotiranogSortiranog(const vector<int>& a) {
  int min = *upper_bound(begin(a), end(a), 0,
                         [&a](int _, int x) {
                           return x < a.back();
                         });
  return min;
}
```

2.15.3 Проналажење оптималне вредности решења бинарном претрагом

Бинарна претрага се може употребити и у процесу оптимизације, ако се проблем може формулисати као проблем проналажења преломне тачке. Овај облик претраге се понекад назива *Бинарна претрага по решењу*. Идеја је да се проблем оптимизације "наћи најмању вредност која задовољава одређени услов", сведе на проблем одлучивања "да ли дата вредност задовољава одређени услов". Бинарну претрагу је могуће применити

ако проблем задовољава својство монотоности, које захтева да ако нека вредност задовољава услов, онда услов задовољавају и све вредности веће од ње, а ако не задовољава, онда услов не задовољавају ни вредности мање од ње. Наравно, сасвим сличан је задатак проналажења највеће вредности која не задовољава услов. Карактеристично за ову употребу бинарне претраге је то што вредности о којима је реч обично нису индекси елемената низа, а често се врши оптимизација и над непрекидним скупом вредности (до на одређену тачност). Такође, провера испуњења услова за сваку конкретну вредност је обично спора и желимо да смањимо број провера испуњења услова колико је могуће. Стога се уместо коришћења библиотечких функција, бинарна претрага ручно имплементира.

Задатак: Дрва

Дрвосеча треба да насече одређену количину дрвета и има тестеру коју може да подешава да сече на било којој целобројној висини (у метрима). Пошто тестера сече само дрво изнад висине на коју је постављена, што је тестера више, насећи ће се мање дрвета. Пошто дрвосеча брине о околини, он не жели да насече више дрвета него што му је потребно. Напиши програм који одређује највишу могућу целобројну висину тестере, тако да дрвосеча добије довољно дрвета (претпостави да увек постоји довољно дрвета).

Улаз: Са стандардног улаза се учитава број дрвећа у шуми n ($1 \leq n \leq 10^5$), а затим низ висина сваког дрвета (низ природних бројева између 1 и 10000, раздвојених са по једним размаком). Након тога учитава се и количина насеченог дрвета (пошто су сва дебла исте дебљине, количина се мери у метрима висине исечених стабала).

Излаз: На стандардни излаз исписати тражену максималну целобројну висину тестере.

Пример

Улаз	*Излаз*
5 24 21 19 14 22 14	18

Решење

Оптимизација бинарном претрагом

Једно решење проблема се може засновати на бинарној претрази по решењу тј. по тражењу оптималне вредности коришћењем бинарне претраге. Постављањем тестере на висину h, код свих дрва која су виша од h биће одсечено $h_i - h$ метара, док од осталих дрва неће бити исечено ништа. На основу тога, за фиксирану висину тестере грубом силом (испитивањем сваког дрвета засебно) у времену $O(n)$ можемо израчунати укупну количину насеченог дрвета. Бинарна претрага је применљива јер знамо да је до одређених висина тестере дрвета довољно, а да је од одређене висине тестере дрвета премало, тако да заправо тражимо преломну тачку, тј. највећу висину тестере за коју је дрвета довољно тј. последњи елемент низа који задовољава услов. Техника како се то може урадити описана је у задатку Први који није дељив. Приметимо да у овом случају немамо вредности смештене у низ, већ их рачунамо по потреби. Зато бинарну претрагу имплементирамо ручно.

Анализа сложености. Ако је максимална висина дрвета M, тада је сложеност овог приступа $O(n \log M)$. Наиме, бинарном претрагом се претражује интервал $[0, M]$, па се провера да ли је насечено довољно дрвета позива $\log M$ пута. Израчунавање количине насеченог дрвета и провера да ли је она довољна врши се једним пролазак кроз низ дрвета и сложености је $O(n)$.

```
int testera(const vector<int>& visine, int potrebno) {
  int od_visina = 0;
  int do_visina = *max_element(begin(visine), end(visine));
  while (od_visina <= do_visina) {
    int visina = od_visina + (do_visina - od_visina) / 2;
    long long naseceno = 0;
    for (int v : visine)
      if (v >= visina)
        naseceno += v - visina;

    if (naseceno >= potrebno)
      od_visina = visina + 1;
    else
      do_visina = visina - 1;
```

```
  }
  return do_visina;
}
```

Задатак: Хиршов h-индекс

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Бинарна претрага

Приметимо да овај проблем има одређен облик монотоности. Смањивањем броја цитата, није могуће да се смањи и број књига које имају тај број цитата. Зато, ако за неки број h важи да је $B_h \geq h$, тада и за све вредности $h' \leq h$ важи да је $B_{h'} \geq h'$. Слично, ако је $B_h < h$, тада и за све вредности $h' \geq h$ важи да је $B_{h'} < h'$. Ово нам омогућава да h-индекс тј. највећи број h такав да је $B_h \geq h$ одредимо бинарном претрагом. У питању је дакле, оптимизација решења бинарном претрагом, или како се то некада каже бинарна претрага по решењу. Пошто вредности које се претражују нису смештене у низ, користимо ручну имплементацију бинарне претраге преломне тачке. Та техника је описана у задатку Први који није дељив.

Анализа сложености. Бинарном претрагом се претражује интервал $[0, n]$, тако да је број корака претраге $O(\log n)$. У сваком кораку се броје књиге са потребним бројем цитата, што се ради једним проласком кроз низ књига тј. позивом библиотечке функције, у сложености $O(n)$. Укупна сложеност је, дакле, $O(n \log n)$.

```
int hIndeks(vector<int>& broj_citata) {
  // binarnom pretragom pronalazimo najveci broj h takav da je B_h >= h
  int h_l = 0, h_d = broj_citata.size();
  while (h_l <= h_d) {
    int h = h_l + (h_d - h_l) / 2;
    // broj knjiga koje imaju bar h citata
    int Bh = count_if(begin(broj_citata), end(broj_citata),
                      [h](int c) {
                        return c >= h;
                      });
    if (Bh >= h)
      h_l = h+1;
    else
      h_d = h-1;
  }
  int h_indeks = h_l - 1;
  return h_indeks;
}
```

Задатак: Муцајући подниз

Ако је s ниска, онда нека s^n означава ниску која се добија ако се свако слово понови n пута (нпр. $(xyz)^3$ је $xxxyyyzzz$). Напиши програм који одређује највећи број n такав да је s^n подниз дате ниске t (то значи да се сва слова ниске s^n јављају у ниски t, у истом редоследу као у s^n, али не обавезно узастопно).

Улаз: У првом реду стандардног улаза налази се ниска s, а у другом ниска t.

Излаз: На стандардни излаз напиши тражени број n.

Пример

Улаз	*Излаз*
xyz	3
xaxxybyxyxzyzzb	

Решење

Дефинисаћемо функцију којом проверавамо да ли је s^n подниз ниске t. Један начин је да експлицитно формирамо ниску s^n и да применимо алгоритам провере подниске. Тај (суштински грамзиви) алгоритам је описан у задатку Реч у реч прецртавањем слова. Мало боље решење је да се алгоритам модификује тако да

се избегне ефективно креирање ниске s^n. Функција провере прима ниску s, број n и ниску t, а затим сваки од карактера из s тражи n пута унутар ниске t.

Линеарна претрага

Када на располагању имамо функцију за проверу, тада оптималну вредност n можемо наћи линеарном претрагом. Кључна опаска је да ако s^n није подниз t за неко n, онда s^k није подниз t ни за једно $k \geq n$. Зато ћемо степен увећавати кренувши од 1 све док не наиђемо на прву вредност n за коју s^n није подниз t и тада ћемо знати да је оптимална вредност $n - 1$.

Анализа сложености. Провера да ли је ниска s^n подниз ниске t врши се у линеарном времену у односу на збир дужина ниске. Ако је $|t|$ дужина ниске t, време за једну проверу је $O(|t|)$. Провере се врше све док се не нађе оптимално n. Оно највише може бити $\left\lfloor \frac{|t|}{|s|} \right\rfloor$, где је $|s|$ дужина ниске s па је сложеност $O(\frac{|t|^2}{|s|})$.

```
bool jeMucajuciPodniz(const string& podniz, const string& niz, int n) {
  int i = 0;
  for (char c : podniz) {
    for (int k = 0; k < n; k++) {
      while (i < niz.size() && niz[i] != c)
        i++;
      if (i == niz.size())
        return false;
      i++;
    }
  }
  return true;
}

int najduziMucajuciPodniz(const string& podniz, const string& niz) {
   int d = 1;
   while (jeMucajuciPodniz(podniz, niz, d))
      d++;
   return d - 1;
}
```

Бинарна претрага

Чињеница да постоји одређени облик монотоности у проблему, нам омогућава да тражену оптималну вредност нађемо бинарном претрагом. Важи да ако s^n није подниз t за неко n, онда s^k није подниз t ни за једно $k \geq n$, а да ако је s^n подниз t за неко n, онда је s^k подниз t за свако $k \leq n$. Зато се са порастом n јављају се прво оне вредности за које услов важи, након који иду вредности за које услов не важи.

Сигурни смо да се оптимална вредност налази у интервалу од 0 па до $\left\lfloor \frac{|t|}{|s|} \right\rfloor$. Бинарном претрагом проналазимо преломну тачку, тј. најмању вредност n такву да s^n није подниз t. Алгоритам је јако сличан оном описаном у задатку Први који није дељив. Приметимо да у овом случају не претражујемо вредности у неком низу, већ је низ потенцијалних вредности n који се претражује само имплицитан, па бинарну претрагу имплементирамо ручно.

Анализа сложености. Провера да ли је ниска s^n подниз ниске t врши се у линеарном времену у односу на збир дужина ниске. Ако је T дужина ниске t, време за једну проверу је $O(T)$. Интервал који се претражује је $[0, \frac{T}{S}]$, где је S дужина ниске s, па је сложеност $O(T \log \frac{T}{S})$.

```
bool jeMucajuciPodniz(const string& podniz, const string& niz, int n) {
  int i = 0;
  for (char c : podniz) {
    for (int k = 0; k < n; k++) {
      while (i < niz.size() && niz[i] != c)
        i++;
      if (i == niz.size())
        return false;
      i++;
    }
```

```
  }
  return true;
}

int najduziMucajuciPodniz(const string& podniz, const string& niz) {
  int l = 0, d = niz.size() / podniz.size();
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (jeMucajuciPodniz(podniz, niz, s))
      l = s + 1;
    else
      d = s - 1;
  }
  return d;
}
```

Задатак: Најкраћа подниска која садржи све дате карактере

Графички дизајнер је преуредио неколико слова у једном фонту и жели да своје промене прикаже клијенту. У дугачком тексту је потребно да одабере најкраћи део (сегмент узастопних слова) који садржи сва слова која је променио.

Улаз: У првој линији стандардног улаза налази се текст (једноставности ради претпоставимо да је састављен само од малих слова енглеског алфабета) чија је дужина највише 10^5 карактера. У другој линији се налази скуп слова (опет, претпоставимо малих слова енглеског алфабета) које је дизајнер променио (слова су написана једно до другог, без размака и без понављања).

Излаз: На стандардни излаз исписати један цео број који представља дужину најкраћег дела текста који садржи све карактере датог скупа. Ако такав део текста не постоји, исписати `nema`.

Пример 1

Улаз	*Излаз*
`dobardansvimakakoste` `arnk`	`10`

Пример 2

Улаз	*Излаз*
`abababababab` `abc`	`nema`

Решење

Бинарна претрага

Захваљујући инкременталности, проверу да ли за дату дужину подниске k постоји нека подниска која садржи све дате карактере можемо урадити у линеарном времену $O(n)$, у једном проласку кроз низ. Користићемо технику покретног прозора и приликом преласка са једне на наредну подниску, из скупа ћемо уклањати први карактер текуће подниске и додаваћемо последњи карактер нове подниске. Слична техника је употребљена и у задатку Сегмент дужине k највећег просека. Ово додатно можемо убрзати (додуше не асимптотски) ако је скуп карактера мали, тако што ћемо пролазити само кроз позиције оригиналне ниске на којима знамо да се јављају карактери из скупа.

Скуп карактера из S унутар текуће подниске ћемо представити помоћу пресликавања карактера у њихов број појављивања. Најједноставније је употребити библиотечку имплементацију мапе тј. речника, јер нам она пружа могућност ефикасног одређивања броја карактера из S унутар подниске (подниска је исправна тј. садржи све потребне карактере ако и само ако је број карактере из S које садржи једнак броју елемената скупа S).

Након тога, најмању дужину k можемо наћи техником бинарне претраге тако што нађемо најмању вредност k за коју важи неки услов. Тај поступак је описан у задатку Први који није дељив. Битно је нагласити да проблем задовољава својство монотоности тј. да ако за неко k не постоји подниска дужине k која садржи све карактере скупа, онда таква подниска не постоји ни за једно мање k тј. да ако за неко k постоји таква подниска, онда она постоји и за свако веће k.

```
// provera da li postoji podniska niske niska duzine k koja sadrzi sve
// karaktere iz S
bool postojiPodniska(const string& niska, const string& S, int k) {
```

```
  map<char, int> karakteri;
  for (int i = 0; i < niska.length(); i++) {
    if (i >= k && S.find(niska[i-k]) != string::npos)
      if (--karakteri[niska[i-k]] == 0)
        karakteri.erase(niska[i-k]);
    if (S.find(niska[i]) != string::npos) {
      karakteri[niska[i]]++;
      if (karakteri.size() == S.size())
        return true;
    }
  }
  return false;
}

// pronalazi duzinu najkrace podniske koja sadrzi sve date karaktere
int duzinaPodniske(const string& niska, const string& karakteri) {
  // binarnom pretragom odredjujemo najmanju duzinu k takvu da u T
  // postoji podniska duzine k koja sadrzi sve karaktere skupa S
  int l = 1, d = niska.length();
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (postojiPodniska(niska, karakteri, s))
      d = s - 1;
    else
      l = s + 1;
  }
  return l;
}
```

Анализа сложености. Бинарном претрагом се претражује интервал $[1, n]$, где је n дужина ниске, па се провера да ли постоји подниска неке фиксне која садржи сва потребна слова врши највише $\log n$ пута. Та провера се врши једним проласком кроз ниску. У сваком кораку врши се линеарна претрага ниске којом је одређен скуп карактера и врше се операције над мапом тј. речником у ком су кључеви карактери из скупа S. Ако претпоставимо да се операције над мапом извршавају у логаритамској сложености, сложеност једне претраге је зато $n \cdot m \cdot \log m$, где је m број карактера у S. Пошто је m веома мали број, фактор $m \log m$ можемо сматрати и константом и укупну сложеност грубо оценити са $O(n \log n)$. Да је m веће, сложеност би се могла поправљати тиме што би се провера припадности карактера скупу S вршила на ефикаснији начин (на пример, представљањем S помоћу бибилотечких колекција за представљање скупа или помоћу низа логичких вредности). Такође је могуће унапред одредити позиције карактера из S унутар ниске (што не утиче на асимптотску сложеност најгорег случаја, јер сви карактери у T могу припадати S, али може доста смањити фактор n у свакој појединачној провери).

Види другачија решења овог задатка.

2.16 Техника два показивача

Угнежђене петље обично подразумевају постојање две бројачке промењљиве од којих спољашња само увећава своју вредност током итерације, док се вредност бројачке петље у унутрашњој увећава до неке горње границе, затим се поново враћа на неку доњу границу и поново увећава и то се понавља више пута, све док спољашња бројачка промењљива не достигне своју максималну вредност. Ово по правилу доводи до квадратне сложености (тј. сложености вишег степена у случају угнежђавања већег броја петљи).

Техника два показивача обухвата широку класу ефикасних алгоритама које такође карактерише постојање две или више бројачких промењљивих, које се крећу кроз елементе неког низа (често сортираног). Међутим, оно што је карактеристично за њих је то што се, за разлику од унутрашњих промењљивих у угнежђеним петљама, ове промењљиве стално "крећу у истом смеру", тј. вредност им се или стално повећава или стално смањује (а честа је и комбинација где се "низ обилази са два краја", где се једна промењљива стално повећава, а друга стално смањује). Техничка реализација може бити било помоћу једне петље која контролише вредности обе промењљиве, било помоћу угнежђених петљи, али тако да се након завршетка тела унутрашње петље,

спољашња променљива увећава до места где се унутрашња петља завршила. Пошто се свака променљива може променити највише n пута (где је n неко горње ограничење њихове вредности, обично дужина низа), број промена (па самим тим и извршавања тела петље) је највише $2n$ и линеаран је по n тј. сложеност му је $O(n)$.

Алгоритми засновани на техници два показивача обично могу да се изведу коришћењем одсецања примењених на угнежђене петље, па је, као и код сваке примене одсецања, потребно пажљиво образложити њихову коректност.

Задатак: Обједињавање

У школи малих жутих мрава наставник је прегледао контролни задатак. Прво је прегледао ђаке који су радили групу А, а затим оне који су радили групу Б, средио је резултате за сваку групу и мраве поређао на основу броја поена који су освојили. Напиши програм који му помаже да од уређеног списка ученика који су радили задатке из групе А и од уређеног списка ученика који су радили задатке из групе Б добије јединствен уређен списак свих ученика.

Улаз: Са стандардног улаза се уноси број ђака m који су радили групу А ($5 \leq m \leq 25000$), а затим неопадајуће сортиран низ поена тих ђака (елементи су у једној линији, раздвојени са по једним размаком). Након тога се уноси број n ђака који су радили групу Б ($5 \leq n \leq 25000$), а затим неопадајуће сортиран низ поена тих ђака (елементи су у једној линији, раздвојени са по једним размаком).

Излаз: На стандардни излаз исписати неопадајуће сортирани низ поена свих ђака заједно, раздвојене са по једним размаком.

Пример

Улаз	*Излаз*

`4`
`1 3 5 7`
`3`
`2 4 5`
	`1 2 3 4 5 5 7`

Решење

Сортирање

Наиван начин да се задатак реши је да се елементи оба учитана низа прекопирају у трећи и да се онда сортирају.

Копирање се може извршити било помоћу петље, било библиотечком функцијом `copy`, док је сортирање најбоље вршити библиотечком функцијом `sort`.

Разни начини сортирања описани су у задатку Сортирање бројева.

Анализа сложености. Копирање низова дужине m и n захтева $m + n$ операција, а сортирање $O((m + n) \cdot \log(m + n))$. Технички, могли смо одмах елементе учитавати у резултујући низ и тако уштедети меморију и време потребно за копирање, али доминатни фактор, а то је време потребно за сортирање би остао. Приметимо да у овом решењу нисмо уопште употребили чињеницу да су полазни елементи већ сортирани.

Напомена. Иако ово решење по времену извршавања не заостаје пуно у односу на оптимално (његово време извршавања је квазилинеарно тј. $O((m+n) \cdot \log(m+n))$, а оптимално време је линеарно тј. $O(m+n)$), оно је доста компликованије него што је потребно. То се у овој имплементацији не види, јер је употребљена библиотечка функција сортирања, међутим, имплементација ефикасног алгоритма сортирања захтева напредне технике програмирања. Интересантно, један од популарних алгоритама сортирања је сортирање обједињавањем (енгл. merge sort) који као свој основни корак захтева извршавање обједињавања два сортирана низа у трећи. Самим тим, донекле је бесмислено проблем обједињавања решавати свођењем на компликованији проблем сортирања.

```
// objedinjavanje dva sortirana niza u treci
vector<int> objedini(const vector<int>& a, const vector<int>& b) {
  // kopiramo dva niza u treci, jedan iza drugog
  vector<int> c(a.size() + b.size());
  copy(a.begin(), a.end(), c.begin());
  copy(b.begin(), b.end(), next(c.begin(), a.size()));
  // sortiramo treci niz
```

```
  sort(c.begin(), c.end());
  return c;
}
```

Алгоритам обједињавања

Задатак можемо решити ефикасним алгоритмом, заснованом на техници два показивача. *Алгоритам обједињавања* (енгл. *merge*) подразумева да су низови који се обједињавају сортирани. Ако је један од низова празан, резултат обједињавања је други низ и његове елементе је потребно једноставно прекопирати у резултат. Ако низови нису празни, пошто су сортирани, први елемент низа је уједно најмањи у њему. Мањи од два почетна елемента је мањи (или једнак) од почетног елемента другог низа, па је мањи или једнак свим елементима у оба низа и самим тим је најмањи елемент од свих и треба да буде први у резултату. Када се тај елемент уклони из низа, добијамо проблем истог типа као и полазни, који се онда решава на исти начин. Имплементација може бити рекурзивна, међутим, рекурзија је репна и лако се елиминише.

Током итеративне имплементације одржавају се два показивача: i који указује на текући елемента првог и j који указује на текући елемент другог низа. Наравно, уместо показивача, i и j могу бити позиције тј. индекси текућих елемената. Док су обе ове промењиве мање од дужине низа по којем се крећу, поредимо елементе на тим позицијама. Ако је елемент на позицији i у првом низу мањи (или једнак) елементу на позицији j у другом низу, тада тај елемент преписујемо у трећи низ (на позицију k коју иницијализујемо на нулу и увећавамо приликом додавања сваког новог елемента) и увећавамо i за 1. У супротном у трећи низ преписујемо елемент из другог низа са позиције j и увећавамо j. Када бар једна од промењивих i или j достигне дужину одговарајућег низа, тада елементе преосталог низа преписујемо у трећи низ. Не морамо експлицитно проверавати да ли у неком од ових низова има преосталих елемената, већ можемо у једној петљи копирати преостале елементе првог, а у другој петљи копирати преостале елементе другог низа (једна од ових петљи ће бити празна).

Пример. Прикажимо рад овог алгоритма и на једном примеру.

- Претпоставимо да је потребно објединити наредна два низа.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   _ _ _ _ _ _ _ _ _
i           j         k
```

- У првом кораку је $i = 0$ и $j = 0$, па се пореде елементи на позицијама 0, тј. елементи 1 и 2. Пошто је 1 мањи, он се преписује у резултујући низ и увећава се леви показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 _ _ _ _ _ _ _ _
  i         j           k
```

- Сада је $i = 1$ и $j = 0$, па се пореде елементи на позицијама 1 и 0, тј. 3 и 2. Пошто је 2 мањи, он се преписује у резултујући низ и увећава се десни показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 _ _ _ _ _ _ _
  i           j           k
```

- Сада је $i = 1$ и $j = 1$, па се пореде елементи на позицијама 1 и 1, тј. 3 и 4. Пошто је 3 мањи, он се преписује у резултујући низ и увећава се леви показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 _ _ _ _ _ _
    i         j             k
```

- Сада је $i = 2$ и $j = 1$, па се пореде елементи на позицијама 2 и 1, тј. 6 и 4. Пошто је 4 мањи, он се преписује у резултујући низ и увећава се десни показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
```

```
1 3 6 8 9   2 4 5 8   1 2 3 4 _ _ _ _ _
    i           j           k
```

- Сада је $i = 2$ и $j = 2$, па се пореде елементи на позицијама 2 и 2, тј. 6 и 5. Пошто је 5 мањи, он се преписује у резултујући низ и увећава се поново десни показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 4 5 _ _ _ _
    i           j           k
```

- Сада је $i = 2$ и $j = 3$, па се пореде елементи на позицијама 2 и 3, тј. 6 и 8. Пошто је 6 мањи, он се преписује у резултујући низ и увећава се леви показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 4 5 6 _ _ _
      i         j           k
```

- Сада је $i = 3$ и $j = 3$, па се пореде елементи на позицијама 3 и 3, тј. 8 и 8. Пошто су једнаки, било који од њих (на пример леви) може бити преписан у резултујући низ и одговарајући показивач се увећава.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 4 5 6 8 _ _
        i         j             k
```

- Сада је $i = 4$ и $j = 3$, па се пореде елементи на позицијама 4 и 3, тј. 9 и 8. Пошто је 8 мањи, он се преписује у резултујући низ и одговарајући показивач се увећава.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 4 5 6 8 8 _
        i         j             k
```

- Пошто у другом низу више нема елемената, преостали елемент левог низа (9) се преписује на крај резултата.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 4 5 6 8 8 9
        i         j               k
```

Анализа сложености. Сваки показивач пролази кроз један од два низа и укупан број корака је $m + n$, па је сложеност овог алгоритма $O(m + n)$.

```
// objedinjava sortirani niz a od n elemenata i sortirani niz b od m
// elemenata smestajuci rezultat u sortirani niz c i vracajuci broj elemenata
int objedini(int a[], int n, int b[], int m, int c[]) {
  int i = 0, j = 0, k = 0;
  while (i < n && j < m)
    c[k++] = a[i] < b[j] ? a[i++] : b[j++];
  while (i < n)
    c[k++] = a[i++];
  while (j < m)
    c[k++] = b[j++];
  return m + n;
}
```

Библиотечка функција

Рецимо и да у стандардној библиотеци језика C++ постоји функција `merge` која врши обједињавање два сортирана низа. Функцији се прослеђују итератори који ограничавају први низ, итератори који ограничавају други низ и итератор на почетак трећег низа у који се смешта резултат. Додатно, могуће је проследити и

функцију поређења која одређује редослед сортирања (полазни низови морају бити сортирани у складу са поретком одређеним том функцијом).

```
// objedinjavanje dva sortirana niza u treci
vector<int> objedini(const vector<int>& a, const vector<int>& b) {
  vector<int> c(a.size() + b.size());
  merge(a.begin(), a.end(), b.begin(), b.end(), c.begin());
  return c;
}
```

Задатак: Близанци

Марија и Петар су близанци и желимо да свакоме од њих двоје купимо по једно одело као поклон за рођендан, али тако да се цене та два поклона што мање разликују (при томе није битно чији поклон ће бити скупљи).

Написати програм који учитава цене свих женских одела и свих мушких одела, а одређује и исписује најмању разлику између цена женског и мушког одела.

Улаз: Са стандардног улаза се учитава:

- у првом реду број мушких одела m ($1 \leq m \leq 50000$),
- у другом реду m целих бројева (цели бројеви између 1 и $2 \cdot 10^9$ раздвојени по једним размаком) – цене мушких одела
- у трећем реду број женских одела n ($1 \leq n \leq 50000$)
- и у четвртом реду n целих бројева (цели бројеви између 1 и $2 \cdot 10^9$ раздвојени по једним размаком) – цене женских одела.

Излаз: На стандардни излаз исписати најмању вредност разлике цена мушког и женског одела.

Пример

Улаз	*Излаз*
5 4680 2120 7940 11530 17820 4 850 13420 5770 6300	1090

Објашњење

Најмања разлика се постиже када се купе одела чије су цене 4680 и 5770 динара.

Решење

Груба сила

Један могући приступ је да одредимо разлику (прецизније, апсолутну вредност разлике) у цени између сваког мушког и сваког женског одела, па од тих разлика нађемо најмању.

Анализа сложености. Сложеност одговара броју парова и процењује се као $O(m \cdot n)$, где је m број мушких, а n број женских одела.

```
// najmanja razlika a1[i] - a2[j]
int blizanci(const vector<int>& a1, const vector<int>& a2) {
  int minRazlika = numeric_limits<int>::max();
  for (int x1 : a1)
    for (int x2: a2)
      minRazlika = min(minRazlika, abs(x1 - x2));
  return minRazlika;
}
```

Упоредни пролаз кроз сортиране низове

Ефикаснији приступ је да се низови цена најпре сортирају, а да се затим истовремено пролази кроз оба низа, рачунајући разлику текућих елемената и напредујући у оном низу у којем је цена тренутно мања, као код класичног алгоритма обједињавања два сортирана низа. Тај је алгоритам описан у задатку Обједињавање.

Успут се, наравно, по потреби ажурира најмања забележена разлика. Када се стигне до краја било којег низа, поступак је завршен и најмања забележена разлика је тада и укупно најмања.

Заиста, пошто су низови сортирани, када се упореде почетни елементи из оба низа, онај који је мањи од њих нема потребе упоређивати са осталим елементима низа коме он не припада, јер ће разлика моћи бити само већа (јер је тај низ сортиран). На тај начин вршимо одсецање, чиме добијамо на ефикасности. Тај елемент онда можемо избацити из даљег разматрања тако што ћемо у низу у ком се он налази прећи на следећи елемент. У специјалном случају када су почетни елементи оба низа једнаки, разлика је једнака нули, што је најмања могућа разлика, па нема потребе вршити даљу анализу.

Пример. На пример, нека су након сортирања вредности једнаке следећим.

```
1 14 28 33 45
8 21 22 41 56 68
```

- Прво поредимо елементе 1 и 8. Разлика је 7. Разлика између броја 1 и свих даљих бројева у доњем низу је већа од 7, па број 1 не морамо више анализирати.
- Након тога поредимо бројеве 14 и 8 и добијамо разлику 6. Разлика између броја 8 и свих бројева иза 14 је већа, па сада ни 8 не морамо више анализирати.
- Поредимо сада бројеве 14 и 21, разлика је 7, а 14 не морамо више да анализирамо.
- И разлика између 28 и 21 је 7, а број 21 не морамо више да анализирамо.
- Разлика између 28 и 22 је 6, а 22 не морамо да анализирамо даље.
- Разлика између 28 и 41 је 13, а 28 не морамо да анализирамо даље.
- Разлика између 33 и 41 је 8, а 33 не морамо да анализирамо даље.
- Разлика између 45 и 41 је 4, а 41 не морамо да анализирамо даље.
- Разлика између 45 и 56 је 11, а 45 не морамо да анализирамо даље. Пошто нема више елемената у горњем низу, поступак се завршава.

Можемо закључити да је најмања могућа разлика једнака 4 (за бројеве 41 и 45).

Анализа сложености. Сложеношћу доминира сортирање, које се извршава у времену $O(m \log m + n \log n)$. Након сортирања, пролазак са два показивача се извршава у времену $O(m + n)$.

```
// najmanja razlika a1[i] - a2[j]
int blizanci(const vector<int>& a1, const vector<int>& a2) {
  // pravimo sortirane kopije dva niza
  auto a1s = a1;
  sort(begin(a1s), end(a1s));
  auto a2s = a2;
  sort(begin(a2s), end(a2s));
  // algoritmom objedinjavanja odredjujemo najmanju razliku
  int i1 = 0, i2 = 0;
  int minRazlika = numeric_limits<int>::max();
  while (i1 < a1s.size() && i2 < a2s.size())
    if (a1s[i1] <= a2s[i2]) {
      minRazlika = min(minRazlika, a2s[i2] - a1s[i1]);
      i1++;
    } else {
      minRazlika = min(minRazlika, a1s[i1] - a2s[i2]);
      i2++;
    }
  return minRazlika;
}
```

Задатак: Број парова датог збира

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Итеративни обилазак са два краја низа помоћу два показивача

Задатак можемо решити тако што сортирамо низ неопадајуће (пошто су сви елементи различити, он ће заправо бити сортиран строго растуће) и применимо технику два показивача, имплементирану итеративно.

Чланове датог збира можемо тражити полазећи са оба краја низа. Обилазимо низ са оба краја: левог ($l = 0$) и десног ($d = n - 1$). Упоредимо $a_l + a_d$ са s.

- Ако је $a_l + a_d > s$ потребно је смањити збир пара елемената, што постижемо заменом елемента мањим, па пошто је низ сортиран у растућем поретку, прелазимо на следећи елемент у десном делу низа (d умањујемо за 1).
- Ако је $a_l + a_d < s$ потребно је повећати збир пара елемената, што постижемо заменом елемента већим, па пошто је низ сортиран у растућем поретку, прелазимо на следећи елемент у левом делу низа (l увећавамо за 1).
- Ако је $a_l + a_d = s$ увећамо број тражених парова, и прелазимо на следећи елемент у левом делу низа (l увећавамо за 1) и на следећи елемент у десном делу низа (d умањујемо за 1).

Процес настављамо док не обиђемо цео низ, то јест док је $l < d$.

Пример. Прикажимо ово на примеру проналажења елемената чији је збир 14 у сортираном низу $1, 2, 5, 7, 9, 11, 13, 14$. Крећемо од пара $(1, 14)$. Пошто је збир већи од траженог, померамо десни крај улево и анализирамо пар $(1, 13)$, који има тражени збир. Зато прелазимо на $(2, 11)$. Пошто је збир сада мањи, померамо леви крај удесно и анализирамо пар $(5, 11)$. Збир је превелики и померамо десни крај улево и анализирамо пар $(5, 9)$. Он има тражени збир, па прелазимо на $(7, 7)$, но тај пар не анализирамо, јер су се показивачи сусрели.

```
int brojParovaDatogZbira(const vector<int>& a, int s) {
  // pravimo sortiranu kopiju niza a
  auto as = a;
  sort(begin(as), end(as));

  // brojimo parove pomocu dva pokazivaca
  int brojParova = 0;
  int levo = 0, desno = as.size() - 1;
  while (levo < desno)
    if (as[levo] + as[desno] > s)
      desno--;
    else if (as[levo] + as[desno] < s)
      levo++;
    else {
      brojParova++;
      levo++;
      desno--;
    }

  return brojParova;
}
```

Доказ коректности. Докажимо коректност претходног поступка. Посматрајмо скуп $S_{l,d}$ који садржи све парове (a_i, a_j) такве да је $0 \leq i < l$ и да је $d < j < n$. Инваријанта претходне петље биће то да:

- променљива која чува текући број парова (означимо је са b) чува број парова скупа $S_{l,d}$ тако да је $a_i + a_j = s$,
- за свако $0 \leq i < l$ важи да је $a_i + a_d < s$ и за свако $d < j < n$ важи да је $a_l + a_j > s$.

Пре уласка у петљу важи да је $l = 0$ и $d = n - 1$. Тада нема индекса i таквог да важи $0 \leq i < l$ нити индекса j таквог да важи $d < j < n$, па је $S_{l,d}$ празан. Пошто је $b = 0$, оба дела инваријанте важе.

Претпоставимо да инваријанта важи при уласку у тело петље и докажимо да је извршавање тела петље одржава.

Претпоставимо прво да је $a_l + a_d > s$. Тада d умањујемо за 1 не мењајући при том број парова b, тј. након извршавања тела петље важи да је $l' = l$, $d' = d - 1$ и $b' = b$. Скуп $S_{l',d'} = S_{l,d-1}$ се може разложити на:

1. скуп $S_{l,d}$ и
2. скуп скуп свих парова (a_i, a_j) таквих да је $0 \leq i < l$ и $j = d$.

Број парова траженог збира у скупу $S_{l,d}$ једнак је b (на основу првог дела инваријанте), док се у другом скупу не налази ни један такав пар. Заиста, на основу другог дела инваријанте знамо да свако $0 \leq i < l$ важи да је $a_i + a_d < s$, па међу паровима другог скупа не може бити ни један који има збир једнак s. Пошто је $b' = b$, први део инваријанте остаје очуван. Потребно је да покажемо и да други део инваријанте остаје очуван. Прво, треба да докажемо да за свако $0 \leq i < l'$ важи да је $a_i + a_{d'} < s$. Пошто је $l' = l$, на основу другог дела инваријанте знамо да за све такве индексе i важи да је $a_i + a_d < s$, а пошто је низ сортиран и пошто су му елементи различити, важи да је $a_{d'} = a_{d-1} < a_d$, па је $a_i + a_{d'} = a_i + a_{d-1} < a_i + a_d < s$. Треба да докажемо и да за свако $d' < j < n$, важи да је $a_{l'} + a_j > s$. На основу другог дела инваријанте то важи за све $d < j < n$. Пошто је $l' = l$ и $d' = d - 1$, остаје само још да се докаже да тај услов важи за d, тј. само да се докаже да важи да је $a_l + a_d > s$, но то важи на основу претпоставке (тј. гране коју тренутно анализирамо).

Веома слично се показује да се у случају $a_l + a_d < s$ повећањем броја l и не мењањем броја d инваријанта одржава.

На крају, остаје случај када је $a_l + a_d = s$. У том случају се врши увећање броја l, умањење броја d и увећање броја b, тј. важи да је $l' = l + 1$, $d' = d - 1$ и $b' = b + 1$. Скуп $S_{l',d'} = S_{l+1,d-1}$ се може разложити на:

1. скуп $S_{l,d}$,
2. скуп свих парова (a_i, a_j) таквих да је $0 \leq i < l$, $j = d$,
3. скуп свих парова таквих да је $i = l$, $d < j < n$ и
4. скуп који садржи само пар (a_l, a_d).

На основу првог дела инваријанте важи да у скупу $S_{l,d}$ има b парова чији је збир s. Пошто је $a_l + a_d = s$, (a_l, a_d) је још један тражени пар. Остаје још да покажемо да у преостала два скупа не постоји ни један пар чији је збир s. Заиста, скуп свих парова таквих да је $0 \leq i < l$, $j = d$ и $a_i + a_j = s$, је празан, јер на основу другог дела инваријанате знамо да је у свим тим паровима $a_i + a_j < s$. Аналогно, на основу другог дела инваријанте доказујемо да је празан и скуп свих парова таквих да је $i = l$, да је $d < j < n$ и $a_i + a_j = s$, јер за све те парове важи да је $a_i + a_j > s$. Дакле, први део инваријанте остаје очуван. Потребно је још доказати да је очуван и други део инваријанте. Потребно је доказати да је за за свако $0 \leq i < l'$ важи да је $a_i + a_{d'} < s$. На основу другог дела инваријанте знамо да за свако $0 \leq i < l$ важи да је $a_i + a_d < s$. Пошто је низ сортиран и сви су му елементи различити и пошто је $d' = d - 1$, важи да је $a_{d'} < a_d$. Зато је $a_i + a_{d'} < a_i + a_d < s$. Пошто је $l' = l + 1$, остаје да се то докаже још да је $a_l + a_{d'} < s$. Но знамо да је $a_l + a_{d'} < a_l + a_d = s$. Аналогно се доказује још да за свако $d' < j < n$ важи $a_{l'} + a_j > s$.

Крај петље наступа када је $l \geq d$. Скуп свих парова (a_i, a_j) таквих да је $0 \leq i < j < n$ се може разложити на:

1. скуп свих парова (a_i, a_j) таквих да је $j \leq d$,
2. скуп $S_{l,d}$ тј. скуп свих парова (a_i, a_j) таквих да је $0 \leq i < l$ и $d < j < n$ и
3. скуп свих парова (a_i, a_j) таквих да је $l \leq i$.

У првом и трећем скупу нема ни један пар чији је збир једнак s. Заиста, ако је $j \leq d$, тада важи да је $0 \leq i < j \leq d \leq l$. На основу другог дела инваријанте знамо да тада важи да је $a_i + a_d < s$, а пошто је низ сортиран важи и да је $a_j \leq a_d$, па је $a_i + a_j \leq a_i + a_d < s$. Ако важи да је $l \leq i$, тада важи и да је $d \leq l \leq i < j < n$. На основу другог дела инваријанте знамо да је $a_l + a_j > s$, па пошто је низ сортиран важи да је $a_l \leq a_i$ и зато је $a_i + a_j \geq a_l + a_j > s$. На основу првог дела инваријанте знамо да је број b једнак броју парова из другог скупа, што је на основу претходног укупан број тражених парова.

Анализа сложености. Пошто се у сваком кораку разлика између d и l смањи бар за 1 (некада и за 2), укупан број корака не може бити већи од n, па је сложеност овог дела алгоритма $O(n)$. Наравно, сложеношћу доминира првобитно сортирање чија је сложеност $O(n \log n)$.

Задатак: Тројке датог збира (3sum)

Такмичари из програмирања имају свој рејтинг који је изражен као неки цео број. На државно екипно такмичење школе треба да пошаљу своје трочлане екипе, међутим, да би такмичење било што занимљивије, упутство организатора је да све екипе буду уједначене тј. да свака екипа на такмичењу има збирни рејтинг

нула. Ако су познати рејтинзи свих такмичара из неке школе, напиши програм који одређује на колико начина школа може да одабере своју екипу.

Улаз: Са стандардног улаза се уноси број такмичара n ($3 \leq n \leq 1000$), а затим и n различитих целих бројева из интервала $[-10^6, 10^6]$, раздвојених са по једним размаком (то су рејтинзи такмичара).

Излаз: На стандардни излаз исписати број могућих трочланих екипа таквих да је укупан збир рејтинга та три члана једнак нули.

Пример

Улаз	*Излаз*
9 -8 -5 7 4 1 -2 9 -3 2	4

Решење

Груба сила

Наивно решење је да се провере све тројке елемената (a_i, a_j, a_k) за $i < j < k$, тј. да се употребе три угнежђене петље у чијем се телу проверава да ли је $a_i + a_j + a_k = 0$ и ако јесте, да се увећа бројач. За то користимо алгоритам бројања елемената филтриране серије.

Анализа сложености. Сложеност овог алгоритма одговара броју тројки и једнака је $O(n^3)$, што је недопустиво велико.

```
// broj trojki ciji je zbir 0
int brojTrojkiZbira0(const vector<int>& a) {
  // broj elemenata niza
  int n = a.size();
  // ukupan broj trojki ciji je zbir 0
  int brojTrojki = 0;
  // prolazimo kroz sve trojke (ai, aj, ak) takve da je i < j < k
  for (int i = 0; i < n - 2; i++)
    for (int j = i + 1; j < n - 1; j++)
      for (int k = j + 1; k < n; k++)
        // ako je zbir trojke 0
        if (a[i] + a[j] + a[k] == 0)
          // uvecavamo broj trojki
          brojTrojki++;

  return brojTrojki;
}
```

Решење техником два показивача

Задатак може бити решен сортирањем и применом технике два показивача да би се иза сваког елемента a_i (почетног елемента тројке) израчунао број парова различитих елемената који имају збир $-a_i$. Претпоставимо да на почетку низ сортирамо растуће. Тада ће сваки суфикс иза позиције i бити сортиран, тако да на њега можемо применити алгоритам обиласка низа са два краја. Тај алгоритам је описан у задатку Број парова датог збира.

Анализа сложености. Пошто је бројање парова у делу низа дужине m сложености $O(m)$ и понавља се за све дужине од $n - 1$ до 2, а сложеност сортирања је $O(n \log n)$, укупна сложеност је $O(n^2)$.

```
// broj trojki ciji je zbir 0
int brojTrojkiZbira0(const vector<int>& a) {
  // pravimo sortiranu kopiju niza a
  auto as = a;
  sort(begin(as), end(as));

  // ukupan broj trojki ciji je zbir 0
  int brojTrojki = 0;
```

```
  // analiziramo svaki moguci pocetni element trojke
  for (int i = 0; i < as.size() - 2; i++) {
    // izracunavamo broj parova u delu niza [i+1, n) ciji je zbir -a[i]
    // posto je ceo niz sortiran, sortiran je i taj deo

    // koristimo tehniku dva pokazivaca
    int l = i + 1;
    int d = as.size() - 1;
    while (l < d) {
      if (as[i] + as[l] + as[d] > 0)
        d--;
      else if (as[i] + as[l] + as[d] < 0)
        l++;
      else {
        brojTrojki++;
        l++;
        d--;
      }
    }
  }

  return brojTrojki;
}
```

Задатак: Разлика висина

У једном одељењу бирају се глумци за школску представу “Станлио и Олио”. Ови глумци су познати по томе што им је била велика разлика у висини. Напиши програм који одређује на колико начина можемо да одаберемо два глумца из одељења тако да им је разлика једнака датом броју r.

Улаз: Са стандардног улаза се уноси прво позитиван природан број r, у наредном реду број ученика у одељењу n ($1 \leq n \leq 50000$), а након тога у наредних n редова висина сваког ученика у милиметрима.

Излаз: На стандардни излаз испиши број парова које је могуће формирати.

Пример

Улаз	*Излаз*
2350 5 15745 18095 15745 16234 13395	4

Решење

Груба сила

Наиван начин да се задатак реши је да се испитају сви уређени парови ученика и да се преброје они чија је разлика једнака траженој.

```
// broj parova elemenata niza a kojima je razlika jednaka datom broju
int brojParovaDateRazlike(const vector<int>& a, int razlika) {
  int broj = 0;
  for (int x : a)
    for (int y : a)
      if (y - x == razlika)
        broj++;
  return broj;
}
```

Анализа сложености. Пошто уређених парова ученика има n^2, сложеност оваквог алгоритма је $O(n^2)$.

Сортирање

Као и у многим проблемима претраге, сортирање низа може довести до ефикаснијих решења. За почетак, ако је низ сортиран, довољно је само да проверавамо парове такве да је други елемент пара иза првог. Дакле, за сваки елемент низа одређујемо број елемената иза њега који са њим дају тражену разлику (он је умањилац, а тражимо потенцијалне умањенике). Пошто је низ сортиран, чим наиђемо на први елемент иза њега који има већу разлику од тражене, такви ће бити и сви елементи у наставку низа, па можемо извршити одсецање и прећи на обраду следећег елемента (умањиоца).

```
// broj parova elemenata niza a kojima je razlika jednaka datom broju
int brojParovaDateRazlike(const vector<int>& a, int razlika) {
  // pravimo sortiranu kopiju niza
  auto as = a;
  sort(begin(as), end(as));

  int broj = 0;
  for (int i = 0; i < as.size(); i++)
    for (int j = i+1; j < as.size(); j++)
      if (as[j] - as[i] == razlika)
        broj++;
      else if (as[j] - as[i] > razlika)
        break;

  return broj;
}
```

Анализа сложености. У најгорем случају не долази до одсецања, а проверавају се сви парови којих има $\binom{n}{2} = \frac{n(n-1)}{2}$, па је сложеност овог приступа $O(n^2)$. Наравно, у сложеност је укључена и сложеност сортирања $O(n \log n)$, међутим, она је у овом случају занемарива у односу на сложеност испитивања свих парова.

Техника два показивача

Задатак можемо решити техником два показивача. Релативно слична техника је коришћена у решењу задатка Број парова датог збира. Низ мора бити сортиран, а оба показивача се крећу слева надесно. Једноставности ради, претпоставимо прво да у низу нема дупликата.

Пример. Прикажимо како би алгоритам радио на примеру одређивања броја елемената чија је разлика 8 у наредном низу.

`1 3 7 8 11 14 16 30 38`

- Крећемо од пара `1 3`. Пошто је разлика мања од тражене елемент `3` не може бити умањеник, па повећавамо умањеник на `7`.
- Анализирамо пар `1 7`. Ситуација је опет иста, па опет повећавамо умањеник на `8`. Наиме, померањем умањиоца са `1` на `3`, разлика би се само смањила, па `7` заиста не може бити умањеник.
- Анализирамо пар `1 8`. И ту је ситуација иста, па опет повећавамо умањеник на `11` (поново закључујемо да се померањем умањиоца са `1` надесно, на `3` или `7` разлика смањује, па `8` заиста не може бити умањеник).
- Анализирамо пар `1 11`. Овај пут је разлика већа од тражене. Стога можемо закључити да `1` не може бити умањилац (даљим померањем умањеника надесно, разлика би се само повећала). Стога прелазимо на наредни умањилац, а то је `3`. Кључна напомена је да су разлике свих умањеника испред `11` и броја `3` мање од тражене разлике `8` (јер су такве биле разике и када је умањилац био мањи)
- Анализирамо пар `3 11`. То је први пар који има дату разлику. Ако су елементи различити, тада се за све умањенике после `11` добија већа разлика у односу на умањилац `3`, па можемо да померимо умањилац на `7`. Слично, умањеник `11` не може да направи ни један даљи пар чија би разлика била једнака траженој, па можемо да померимо умањеник на `14`.
- Анализирамо пар `7 14`. Разлика је мања од тражене, па повећавамо умањеник на `16`.

- Анализирамо пар `7 16` разлика је већа од тражене, па померамо умањилац на `8`.
- Анализирамо пар `8 16` чија је разлика једнака траженој. Након тога можемо повећати и умањилац на `11` и умањеник на `17`.
- Анализирамо пар `11 30` и пошто му је разлика већа од тражене, померамо умањилац на `14`.
- Анализирамо пар `14 30` и пошто му је разлика већа од тражене, померамо умањилац на `16`.
- Анализирамо пар `16 30` и пошто му је разлика већа од тражене, померамо умањилац на `30`.
- Анализирамо пар `30 30` коме је разлика мања од тражене, па померамо умањеник на `38`.
- Анализирамо пар `38 30` коме је разлика једнака траженој, па померамо и умањилац и умањеник. Пошто не постоји већи умањеник, алгоритам се завршава.

```
// broj parova elemenata niza a kojima je razlika jednaka datom broju
int brojParovaDateRazlike(const vector<int>& a, int razlika) {
  // pravimo sortiranu kopiju niza a
  auto as = a;
  sort(begin(as), end(as));

  int broj = 0;
  int i = 0, j = 1;
  while (j < as.size()) {
    if (as[j] - as[i] < razlika)
      j++;
    else if (as[j] - as[i] > razlika)
      i++;
    else {
      // pronalazimo sve elemente jednake as[i]
      int ii;
      for (ii = i+1; ii < as.size() && as[ii] == as[i]; ii++)
        ;
      // odredjujemo koliko ih ima
      int broj_ai = ii - i;
      // preskacemo ih
      i = ii;

      // pronalazimo sve elemente jednake as[j]
      int jj;
      for (jj = j+1; jj < as.size() && as[jj] == as[j]; jj++)
        ;
      // odredjujemo koliko ih ima
      int broj_aj = jj - j;
      // preskacemo ih
      j = jj;

      // uvecavamo brojac za broj parova (as[i], as[j])
      broj += broj_ai * broj_aj;
    }
  }

  return broj;
}
```

Доказ коректности. Опишимо формално претходни поступак и докажимо његову коректност. Одредимо колико парова дате разлике r постоји у интервалу $[i, n)$, ако знамо да важи инваријанта да је $a_{j-1} - a_i < r$. Вршимо иницијализацију $i = 0$ и $j = 1$, тако да одређујемо број парова и интервалу $[0, n)$, а инваријанта је задовољена јер је $a_{j-1} - a_i = a_0 - a_0 = 0 < r$.

- Ако је $j = n$, тада у интервалу $[i, n)$ не постоји ни један пар дате разлике. Заиста, на основу инваријанте важи да је $a_{n-1} - a_i < r$. Пошто је низ сортиран, и повећањем i и смањивањем j разлика се смањује.

Зато парови бројева унутар интервала $[i, n)$ имају мању разлику од r.

- Ако је $a_j - a_i < r$, тада знамо да у интервалу $[i, j]$ не постоји ни један пар бројева чија је разлика једнака r (јер је разлика елемената унутар интервала увек мања неко разлика крајњих елемената). Инваријанта је задовољена за пар $(i, j + 1)$ па увећавамо j и настављамо поступак.
- Ако је $a_j - a_i > r$, тада ни један пар $a_{j'} - a_i$ за $i < j' < n$ нема разлику r. На основу инваријанте знамо да је $a_{j-1} - a_i$ мање од r, па пошто је низ сортиран то важи и за све елементе $i < j' < j$. Пошто је $a_j - a_i > r$ и пошто је низ сортиран повећањем j се повећава разлика, па су разлике за $j \leq j' < n$ веће од r. Зато се у интервалу $[i, n)$ сви евентуални парови чија је разлика r налазе у интервалу $[i + 1, n)$ и поступак настављамо тако што увећавамо i за 1. Још морамо доказати да тада инваријанта важи тј. да је $a_{j-1} - a_{i+1} < r$, међутим то важи јер је низ сортиран и важи $a_i < a_{i+1}$, а на основу инваријанте је важило да је $a_{j-1} - a_i < r$.
- На крају, ако је $a_j - a_i = r$, тада смо пронашли један пар. Пошто смо претпоставили да у низу нема дупликата и да је низ сортиран, a_i не може бити члан ни једног другог пара са разликом r у интервалу $[i, n)$ - пошто је низ сортиран померањем умањеника налево разлика се смањује, а померањем надесно, она се повећава. Дакле, сви евентуални парови чија је разлика r налазе се у интервалу $[i + 1, n)$. Поступак се може наставити увећавањем и индекса i и индекса j за 1. Заиста, инваријанта је задовољена јер је $a_{j+1-1} - a_{i+1} = a_j - a_{i+1} < a_j - a_i = r$.

Анализа сложености. Сложеност овог приступа је $O(n \log n)$ захваљујући почетном сортирању, док је сложеност друге фазе, након сортирања линеарна тј. $O(n)$. Заиста и умањеник и умањилац се крећу у истом смеру (вредност оба показивача се само увећава), па се може направити највише $2n$ корака.

Пређимо сада на случај у ком се елементи у низу могу понављати.

Обрада дупликата читањем серије истих елемената

Ако се елементи у низу понављају, онда у тренутку када нађемо први пар (i, j) такав да је $a_i - a_j = r$, одређујемо број појављивања n_i елемента a_i и број појављивања n_j елемента a_j, број парова увећавамо за $n_i \cdot n_j$ (јер свако појављивање вредности a_i можемо искомбиновати са сваким појављивањем вредности a_j) и након тога поступак настављамо од индекса $(i + n_i, j + n_j)$.

```
// broj parova elemenata niza a kojima je razlika jednaka datom broju
int brojParovaDateRazlike(const vector<int>& a, int razlika) {
  // pravimo sortiranu kopiju niza a
  auto as = a;
  sort(begin(as), end(as));

  int broj = 0;
  int i = 0, j = 1;
  while (j < as.size()) {
    if (as[j] - as[i] < razlika)
      j++;
    else if (as[j] - as[i] > razlika)
      i++;
    else {
      // pronalazimo sve elemente jednake as[i]
      int ii;
      for (ii = i+1; ii < as.size() && as[ii] == as[i]; ii++)
        ;
      // odredjujemo koliko ih ima
      int broj_ai = ii - i;
      // preskacemo ih
      i = ii;

      // pronalazimo sve elemente jednake as[j]
      int jj;
      for (jj = j+1; jj < as.size() && as[jj] == as[j]; jj++)
        ;
      // odredjujemo koliko ih ima
```

```
      int broj_aj = jj - j;
      // preskacemo ih
      j = jj;

      // uvecavamo brojac za broj parova (as[i], as[j])
      broj += broj_ai * broj_aj;
    }
  }

  return broj;
}
```

Анализа сложености. Сложеношћу и даље доминира сортирање чија је сложеност $O(n \log n)$, док је сложеност друге фазе и даље линеарна тј. $O(n)$.

Обрада дупликата пребројавањем појављивања

Још један начин да се реши проблем понављања елемената је да се у првој фази низ вредности трансформише у низ парова који садрже вредности и њихов број појављивања.

```
// broj parova elemenata niza a kojima je razlika jednaka datom broju
int brojParovaDateRazlike(const vector<int>& a, int razlika) {
  // pravimo sortiranu kopiju niza
  auto as = a;
  sort(begin(as), end(as));

  // odredjujemo broj pojavljivanja svakog elementa
  vector<pair<int, int>> b;
  b.reserve(as.size());
  b.emplace_back(as[0], 1);
  for (int i = 1; i < as.size(); i++) {
    if (as[i] == b.back().first)
      b.back().second++;
    else
      b.emplace_back(as[i], 1);
  }

  // trazimo elemente cija je razlika jednaka datoj
  int broj = 0;
  int i = 0, j = 0;
  while (j < b.size()) {
    if (b[j].first - b[i].first < razlika)
      j++;
    else if (b[j].first - b[i].first > razlika)
      i++;
    else {
      broj += b[j].second * b[i].second;
      i++; j++;
    }
  }

  return broj;
}
```

Анализа сложености. Сортирање је сложености $O(n \log n)$. Пребројавање елемената се након тога извршава у сложености $O(n)$, једним проласком кроз низ, након чега се са два показивача пролази кроз низ опет у сложености $O(n)$. Укупна сложеност је, дакле, $O(n \log n)$. Имплементација користи и помоћни низ, али меморијска сложеност остаје $O(n)$.

Задатак: Сегмент датог збира у низу природних бројева

У датом низу позитивних природних бројева наћи све сегменте (њихов почетак и крај) чији је збир једнак датом позитивном броју (бројање позиција почиње од нуле).

Улаз: У првој линији стандардног улаза налази се задати позитиван природни број z који представља дати збир $0 < z < 10^6$, у другој број елемената низа, N ($2 \leq N \leq 50000$), а затим, у свакој од наредних N линија стандардног улаза, по један елемент низа (позитиван природни број мањи од 200).

Излаз: У свакој линији стандардног излаза исписују се два броја (цели бројеви) одвојена празнином, који представљају индексе почетка и краја сегмента (бројано од нуле). Ако постоји више тражених сегмената њихове индексе исписати сортирано на основу левог краја.

Пример

Улаз	*Излаз*
125 10 60 40 25 50 50 100 25 35 30 35	0 2 2 4 5 6 6 9

Решење

Груба сила

Наивно решење грубом силом претпоставља да се израчунају збирових свих сегмената и да се провери да ли су једнаки датом броју. Чак и када збирове сегмената рачунамо инкрементално, добијамо неефикасно решење. Инкрементално израчунавање збирова сегмената приказано је, на пример, у задатку Највећи збир префикса.

```
void ispisiSegmenteDatogZbira(const vector<int>& a, int trazeniZbir) {
  // analiziramo sve intervale [i, j] za i <= j
  for (int i = 0; i < a.size(); i++) {
    // zbir intervala [i, j]
    int zbir = 0;
    for (int j = i; j < a.size(); j++) {
      // izracunavamo zbir intervala [i, j] na osnovu zbira intervala [i, j-1]
      zbir += a[j];
      // ako je zbir jednak trazenom, prijavljujemo interval
      if (zbir == trazeniZbir)
        cout << i << " " << j << endl;
    }
  }
}
```

Анализа сложености. Постоји $O(n^2)$ сегмената, а збир сваког сегмента на основу збира претходног сегмента добијамо у сложености $O(1)$, па је укупна сложеност $O(n^2)$.

Техника два показивача

Претрагу сегмената грубом силом можемо коришћењем одсецања начинити много ефикаснијом.

Пример. Посматрајмо пример проналажења првог сегмента у низу `1 2 3 5 15 1 2 5` који има збир 21.

Крећемо да испитујемо збирове сегмената који почињу на позицији 0. Све док је збир текућег сегмента строго мањи од 21, потребно је да проширујемо сегменте.

```
i   a[i]   zbir
0   1      1
1   2      3
2   3      6
3   5      11
4   15     26
```

У тренутку када је збир постао строго већи од тражене вредности 21, сигурни смо да ни један сегмент који почиње на позицији 0 не може имати збир 21. Наиме, пошто су сви даљи елементи стриктно позитивни, њиховим укључивањем би се добио само још већи збир. Због тога можемо да пређемо на сегменте који почињу на позицији 1. Важна (и не баш тривијална) опаска је то да сви сегменти који почињу на позицији 1 и завршавају се пре текуће позиције 4 имају збир строго мањи од 21 и стога их није потребно експлицитно испитивати. Наиме, сви сегменти који почињу на позицији 0, а завршавају се пре текуће позиције су имали збир мањи од 21, па се уклањањем елемента на позицији 0 добијају сегменти чији је збир још мањи. Дакле, први кандидат за збир 21, је сегмент који почиње на позицији 1 и завршава се на позицији 4. Његов збир лако добијамо одузимањем почетне вредности 1 са позиције 0 од збира текућег сегмента.

```
i   a[i]   zbir
1   2      2
2   3      5
3   5      10
4   15     25
```

Пошто и тај сегмент има збир већи од 21, такав збир ће имати и сви даљи сегменти који почињу на позицији 1, па можемо прећи на сегменте који почињу на позицији 2. Поново је први кандидат онај који се завршава на позицији 4 (јер сви који се раније завршавају сигурно имају збир мањи од 21).

```
i   a[i]   zbir
2   3      3
3   5      8
4   15     23
```

Поново је збир превелики, па прелазимо на сегменте који почињу на позицији 3. Први кандидат је сегмент који се завршава на позицији 4.

```
i   a[i]   zbir
3   5      5
4   15     20
```

Овај пут тај сегмент има збир мањи од траженог, па га је потребно проширити надесно. Додавањем наредног елемента добијамо сегмент чији је збир једнак траженом.

```
i   a[i]   zbir
3   5      5
4   15     20
5   1      21
```

Након овога смо сигурни да нема више сегмената траженог збира који почињу на позицији 3, па прелазимо на позицију 4 и поступак се по истом принципу наставља даље.

Дакле, одржавамо текући сегмент и његов збир. Док је тај збир мањи од траженог проширујемо сегмент надесно (док је то могуће), а када збир постане већи или једнак траженом скраћујемо сегмент са леве стране.

```
void ispisiSegmenteDatogZbira(const vector<int>& a, int trazeniZbir) {
  // granice segmenta
  int i = 0, j = 0;
  // zbir segmenta
  int zbir = a[0];
  while (true) {
    // na ovom mestu vazi da je zbir = sum(ai, ..., aj) i da
    // za svako i <= j' < j vazi da je sum(ai, ..., aj') < trazeniZbir

    if (zbir < trazeniZbir) {
      // prelazimo na interval [i, j+1]
```

```
      j++;
      // ako takav interval ne postoji, zavrsili smo pretragu
      if (j >= a.size())
        break;
      // izracunavamo zbir intervala [i, j+1] na osnovu zbira intervala [i, j]
      zbir += a[j];
    } else {
      // ako je zbir jednak trazenom, vazi da je sum(ai, ..., aj) = trazeniZbir
      // pa prijavljujemo interval
      if (zbir == trazeniZbir)
        cout << i << " " << j << endl;
      // prelazimo na interval [i+1, j]
      // izracunavamo zbir intervala [i+1, j] na osnovu zbira intervala [i, j]
      zbir -= a[i];
      i++;
    }
  }
}
```

Доказ коректности. Докажимо и формално да је претходни поступак коректан. Обележимо са $z_{ij} = \sum_{k=i}^{j} a_k$ збир елемената низа a чији индекси припадају сегменту $[i, j]$, а са z тражени збир елемената. Пошто су сви елементи низа a позитивни, збирови елемената сегмента задовољавају својство монотоности тј. важи да из $i < i' \leq j$ следи $z_{ij} > z_{i'j}$ и да из $i \leq j < j' < n$ следи $z_{ij} < z_{ij'}$.

Претпоставимо да за неки интервал $[i, j]$ знамо да за свако j' такво да је $i \leq j' < j$ важи да је $z_{ij'} < z$. Постоје следећи случајеви за однос z_{ij} и z.

- Прво, ако је $z_{ij} < z$, тада ни за један интервал који почиње на позицији i, а завршава се најкасније на позицији j не може важити да му је збир елемената z, и проверу је потребно наставити од интервала $[i, j + 1]$, увећавајући j за 1. Ако такав интервал не постоји (ако је $j + 1 = n$), онда се претрага може завршити (јер је и за свако i' такво да је $i < i' \leq j = n - 1$ важи $z_{i'j} < z_{ij} < z$, а зато и за свако j' такво да је $i' \leq j' < j = n - 1$ важи да је $z_{i'j'} < z_{i'j} < z$, тако да за сваки интервал $[i', j']$ такав да је $i \leq i' \leq j' < n$ важи да је $z_{i'j'} < z$).

- Друго, претпоставимо да је $z_{ij} \geq z$. Ако је $z_{ij} = z$, тада је пронађен један задовољавајући интервал и потребно је обрадити његове границе i и j. То је једини сегмент који почиње на позицији i са збиром z. Ако је $z_{ij} > z$, онда таквих сегмената нема. Наиме, пошто су сви елементи низа a позитивни, за свако j'' такво да је $j < j'' < n$ важи да је $z \leq z_{ij} < z_{ij''}$. Дакле, претрагу можемо наставити увећавајући вредност i. За све вредности j' такве да је $i + 1 \leq j' < j$ важи да је $z_{(i+1)j'} < z$. Наиме, пошто је $a_i > 0$ важи да је $z_{(i+1)j'} < z_{ij'} < z$. Дакле, на сегмент $[i + 1, j]$ може се применити анализа случајева истог облика као на интервал $[i, j]$.

Приликом имплементације одржаваћемо интервал $[i, j]$ и његов збир ћемо израчунавати инкрементално - приликом повећања броја j збир ћемо увећавати за a_j, а приликом повећања броја i збир ћемо умањивати за a_i. сличну технику смо користили, на пример, у задатку Највећи збир префикса.

Имплементација са једном петљом

Један начин да се на основу претходне анализе направи имплементација је да се у сваком кораку петље одржавају две променљиве i и j и променљива $zbir$. Обезбедићемо да при сваком уласку у тело петље важи да променљива $zbir$ чува текућу вредност z_{ij} и да је за свако $j' < j$ испуњено да је $z_{ij'} < z$. Ако се i и j иницијализују на нулу, тада се $zbir$ треба иницијализовати на a_0, чиме се задовољава претходни услов.

У телу петље проверавамо да ли је $zbir$ мањи од траженог и ако јесте, увећавамо вредност j. Ако j достигне вредност n, тада можемо прекинути петљу и завршити претрагу. Ако је увећано j мање од n, онда збир увећавамо за a_j и прелазимо на наредни корак петље (услов који смо наметнули да важи при уласку у петљу ће бити овим бити задовољен).

Ако вредност променљиве $zbir$ није мања од тражене вредности z проверавамо да ли јој је једнака. Ако јесте, пријављујемо пронађени интервал $[i, j]$. Затим прелазимо на обраду наредног интервала тако што збир умањујемо за вредност a_i и i увећавамо за 1 (услов који смо наметнули да важи при уласку у петљу ће овим

бити опет задовољен).

Имплементација са угнежђеним петљама

Још једна могућа имплементација садржи две угнежђене петље. У првој десни крај сегмента померамо надесно све док не стигнемо до краја или док је збир текућег сегмента мањи од траженог. У другој леви крај сегмента померамо надесно све док је збир већи или једнак од траженог (при том проверавајући да ли је збир једнак траженом и ако јесте, пријављујући пронађени интервал).

```
void ispisiSegmenteDatogZbira(const vector<int>& a, int trazeniZbir) {
  int i = 0, j = 0; // granice segmenta
  int zbir = 0;     // zbir segmenta [i, j-1]
  while (j < n) {
    // prosirujemo segment nadesno dok god je zbir manji od trazenog
    while (j < n && zbir < trazeniZbir) {
      // dodajemo novi element
      zbir += a[j];
      j++;
    }

    // skracujemo interval sve dok je zbir veci od trazenog
    while (zbir >= trazeniZbir) {
      // ako je zbir intervala jednak trazenom ispisujemo
      // pronadjeni interval
      if (zbir == trazeniZbir)
        cout << i <<  " " <<  j - 1 << endl;
      // uklanjamo pocetni element
      zbir -= a[i];
      i++;
    }
  }
}
```

Анализа сложености. Иако садржи угнежђене петље, ово решење је линеарне сложености у односу на дужину низа тј. сложености је $O(n)$. Наиме, промењиве i и j се у сваком кораку увећавају за један и никада се не умањују, тако да је укупан број корака ограничен вредношћу $2n$.

Имплементација у којој се обрађује сваки десни крај сегмента

Још један начин имплементације је да у петљи обрађујемо све вредности десног краја j од 0 до $n-1$. Обезбедићемо да при сваком уласку у тело петље променљива $zbir$ садржи вредност $z_{i(j-1)}$ и да је та вредност строго мања од тражене вредности збира z. Пошто i иницијализујемо на 0, збир је такође потребно иницијализовати на нулу, чиме се тражени услов обезбеђује (јер је тражена вредност строго позитивна).

На почетку петље увећавамо $zbir$ за вредност a_j и тако је постављамо на вредност z_{ij}.

Док је збир већи од траженог увећавамо леви крај интервала i и збир умањујемо за a_i све док збир не постане мањи или једнак траженом (ако је збир у почетку једнак траженом, ово померање левог краја се неће ни једном извршити). Таква вредност збира ће се сигурно достићи у неком тренутку (када i постане веће од j тада ће се збир спустити до нуле, што је сигурно мање од тражене вредности). Након тога проверавамо да ли је збир једнак траженом и ако јесте, пријављујемо да смо пронашли одговарајући интервал $[i, j]$, увећавамо i за један и умањујемо збир за вредност a_i. Након овога ће сигурно важити да је збир мањи од траженог и да ће при евентуалном наредном уласку у тело петље бити једнак вредности $z_{i(j-1)}$ (за увећану вредност j).

Рецимо и да, ако је $zbir$ мањи од траженог, није потребно ништа додатно урадити, јер ће у евентуалном наредном уласку у петљу тражени услов бити испуњен ($zbir$ садржати вредност $z_{i(j-1)}$, за увећану вредност j, која је мања тражене вредности).

```
void ispisiSegmenteDatogZbira(const vector<int>& a, int trazeniZbir) {
  int i = 0;      // pocetak intervala
  int zbir = 0;   // zbir segmenta
  for (int j = 0; j < a.size(); j++) {
    // na ovom mestu vazi da je  zbir = sum(ai, ..., aj-1) < trazeniZbir
```

```
    // izracunavamo zbir intervala [i, j] na osnovu zbira intervala [i, j-1]
    zbir += a[j]; // sada vazi da je zbir = sum(ai, ..., aj)

    // dok je zbir intervala [i, j] veci od trazenog, umanjujemo ga
    // suzavanjem intervala s leve strane
    while (zbir > trazeniZbir) {
      zbir -= a[i];
      i++;
      // ostaje da vazi da je zbir = sum(ai, ..., aj)
    }
    // sada je zbir = sum(ai, ..., aj) <= trazeniZbir

    // ako je zbir intervala [i, j] jednak trazenom
    if (zbir == trazeniZbir) {
        // znamo da je zbir = sum(ai, ..., aj) = trazeniZbir, pa
        // prijavljujemo interval [i, j]
        cout << i << " " << j << endl;
        // prelazimo na interval [i+1, j], azurirajuci zbir
        zbir -= a[i];
        i++;
    }
    // na ovom mestu znamo da je zbir = sum(ai, ..., aj) < trazeniZbir
  }
}
```

Анализа сложености. Из истих разлога као и претходно и ово решење је линеарне сложености у односу на дужину низа тј. сложености $O(n)$.

Задатак: Најкраћа подниска која садржи све дате карактере

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Груба сила - провера свих подниски

Најдиректнији алгоритам био би да се разматрају све подниске (оне су одређене индексима $0 \leq i \leq j < n$), да се за сваку од њих провери да ли је исправна тј. да ли садржи све карактере из скупа S и да се међу свим исправним поднискама пронађе најкраћа. Овакву претрагу грубом силом је веома једноставно имплементирати.

```
// proverava da li niska na pozicijama [i, j] sadrzi karakter c
bool podniskaSadrziKarakter(const string& niska, int i, int j, char c) {
  for (int k = i; k <= j; k++)
    if (niska[k] == c)
      return true;
  return false;
}

// provera da li niska na pozicijama [i, j] sadrzi sve karaktere iz niske S
bool podniskaSadrziSveKaraktere(const string& niska, int i, int j,
                                const string& S) {
  for (char c : S)
    if (!podniskaSadrziKarakter(niska, i, j, c))
      return false;
  return true;
}
```

```
// "beskonacno"
const int INF = numeric_limits<int>::max();

// pronalazi duzinu najkrace podniske koja sadrzi sve karaktere iz skupa S
int duzinaPodniske(const string& niska, const string& S) {
  // duzina najkrace do sada pronadjene niske koja sadrzi sve
  // karaktere iz S
  int min_duzina = INF;

  // obradjujemo sve pozicije pocetka podniske
  for (int i = 0; i < niska.size(); i++) {
    // preskacemo karaktere koji nisu u skupu S
    // jer najkraca podniska mora da pocne sa karakterom iz S
    if (S.find(niska[i]) == string::npos) continue;

    // obradjujemo sve pozicije kraja podniske koja pocinje na poziciji i
    for (int j = i; j < niska.size(); j++)
      if (podniskaSadrziSveKaraktere(niska, i, j, S)) {
        int duzina = j - i + 1;
        if (duzina < min_duzina)
          min_duzina = duzina;
        break;
      }
  }
  return min_duzina;
}
```

Анализа сложености. Проверава се $O(n^2)$ подниски, и за сваку се врши m линеарних провера, па се може показати да је сложеност је $O(n^3 \cdot m)$ где је n дужина текста, а m број карактера у скупу S. Параметар m има малу вредност, али димензија n може бити велика, па је овај алгоритам веома неефикасан.

Груба сила - оптимизације

Када подниска одређена позицијама i и j садржи све карактере из скупа S, онда и све подниске одређене већим вредностима j такође садрже све карактере из тог скупа, а за њих знамо да су дуже и нема потребе разматрати их (вршимо одсецање у претрази).

Пример. На пример, ако се унутар ниске `xCxxBxxBxxAxxBxxCxxxxBxAxAxCxxxBx` тражи подниска у коме се јављају слова из скупа `ABC`. Након проналаска ниске `CxxBxxBxxA`, која садржи све потребне карактере, нема потребе разматрати њена проширења надесно (на пример, подниску `CxxBxxBxxAxxB`).

Дакле прва пронађена исправна подниска која почиње на позицији i уједно је и најкраћа исправна подниска пронађена на тој позицији. Зато је одмах након проналаска прве исправне пондиске и ажурирања вредности најмање дужине могуће прекинути унутрашњу петљу (наредбом `break`).

Анализа сложености. Ова оптимизација не поправља асимптотску сложеност најгорег случаја (на пример, у случајевима када не постоји тражена подниска), али у многим случајевима може довести до убрзања.

Разматрање само позиција унутар ниске на којима су карактери из скупа

Приметимо да тражена најкраћа подниска мора да почиње и да се завршава карактером из скупа S (рећи ћемо да су само ти карактери релевантни). У супротном би карактери на почетку и на крају подниске који нису у скупу S могли да буду уклоњени чиме би се добила краћа подниска која би и даље садржала све карактере из скупа S. Такође, приликом провере да ли одређена подниска садржи све карактере из S, потребно је анализирати само оне њене позиције на којима су карактери из S. Зато ћемо у фази претпроцесирања само изградити низ позиција релевантних карактера у ниски (рећи ћемо да су то релевантне позиције) и затим ћемо разматрати само подниске који почињу и завршавају се на позицијама унутар тог низа (на тај начин вршимо одсецање у претрази).

Анализа сложености. Ако се провера да ли је позиција релевантна врши линеарном претрагом ниске `S`, Време потребно за изградњу низа релевантних позиција је $O(n \cdot m)$ (пошто је m веома мали број, ово је

практично линеарно тј. $O(n)$, а може се снизити и на $O(m+n)$ ако би се скуп S представио својом карактеристичном функцијом – асоцијативним низом 26 логичких вредности).

Иако се овим не снижава сложеност најгорег случаја (на пример, када су сви карактери ниске у скупу S), у случају да постоји значајан број карактера у тексту који нису у скупу S, претрага се може убрзати.

Скуп релевантних карактера текуће подниске

Проверу да ли се сви карактери из скупа S јављају унутар подниске између две релевантне позиције i и j можемо извршити тако што одредимо скуп свих релевантних карактера на свим релевантним позицијама ниске између i и j (укључујући и њих). Сви карактери из тако направљеног скупа ће бити у скупу S (јер разматрамо само релевантне позиције), па је уместо испитивања да ли је тај скуп једнак скупу S, довољно испитати само да ли има исти број елемената као S (а пошто ниска S по условима задатка нема поновљених карактера, број елемената скупа S једнак је дужини ниске S). Изградњу скупа релевантних карактера који се појављују унутар текуће ниске S можемо вршити инкрементално, тако што приликом сваког повећања j, тј. приликом преласка на нову релевантну позицију j додамо карактер на тој позицији у тај скуп, што захтева само константно време (ако скуп карактера представимо низом или библиотечким скупом, јер је укупан број могућих карактера само 26).

Скуп карактера у језику C++ можемо представити преко низа 26 логичких вредности и додатног бројача који представља број карактера у скупу. Ипак, елегантнија имплементација се добија ако употреби библиотечка имплементација скупа (`set` или `unordered_set`). Пример употребе скупа може се видети у задатку Дупликати.

```
// "beskonacno"
const int INF = numeric_limits<int>::max();

// pronalazi duzinu najkrace podniske koja sadrzi sve karaktere iz skupa S
int duzinaPodniske(const string& niska, const string& S) {
  // pozicije unutar niske na kojima su karakteri iz skupa karakteri
  vector<int> relevantne_pozicije;
  for (int i = 0; i < niska.size(); i++)
    // ako se niska[i] nalazi u skupu karakteri
    if (S.find(niska[i]) != string::npos)
      // zapamti njegovu poziciju i
      relevantne_pozicije.push_back(i);

  int min_duzina = INF;
  for (auto i = relevantne_pozicije.begin(); i != relevantne_pozicije.end(); i++) {
    // relevantni karakteri koje sadrzi trenutna podniska
    set<char> karakteri_podniske;
    for (auto j = i; j != relevantne_pozicije.end(); j++) {
      karakteri_podniske.insert(niska[*j]);
      if (karakteri_podniske.size() == S.size()) {
        int duzina = *j - *i + 1;
        if (duzina < min_duzina)
          min_duzina = duzina;
        break;
      }
    }
  }
  return min_duzina;
}
```

Анализа сложености. Сложеност алгоритма који за сваку позицију i одређује најкраћу исправну подниску која почиње на позицији i уз све наведене оптимизације је $O(n^2)$.

Техника два показивача

Ниске и низови се по правилу обрађују с лева на десно. Често се у алгоритмима који обрађују подниске (тј. сегменте низа) за сваку фиксирану позицију разматрају све подниске које на њој *почињу*. Ипак, ефикаснији алгоритми се често могу конструисати тако што се за сваку позицију одреде све подниске који се на њој *завршавају*. У питању је индуктивна конструкција која конструкцију најбоље подниске која се завршава

на текућој позицији одређује инкрементално, на основу познавања најбоље подниске која се завршава на претходној позицији. Дуално, могу се разматрати подниске који почињу на свакој позицији, али, да би се добио ефикаснији алгоритам заснован на инкременталности, позиције би требало обрађивати здесна на лево.

Обрађиваћемо све релевантне позиције (позиције на којима се налазе карактери које подниска мора да садржи) унутар ниске, редом, са лева на десно, и за сваку од њих ћемо проналазити најкраћу подниску која се на њој завршава. Ако за сваку позицију знамо најкраћу подниску која се на њој завршава, од свих таквих можемо наћи најкраћу. Опет имамо једноставан аргумент да ће најкраћа подниска бити пронађена, јер се он сигурно завршава на некој релевантној позицији и уједно је најкраћа од свих подниски које се на тој позицији завршавају, тако да ће сигурно бити узета у обзир приликом одређивања најмање дужине. Кључне опаске за ефикасно одређивање најкраће исправне подниске која се завршава на некој релевантној позицији су то да ћемо њено одређивање увек започињати проширивањем најкраће исправне подниске који се завршава на претходној релеватној позицији, као и то да ако се први релевантан карактер у подниски јавља бар још једном касније у њој, онда та подниска не може бити најкраћа подниска који садржи све карактере скупа S (заиста, почетно парче те подниске све до следећег карактера из скупа S се може уклонити и опет ће се добити исправна подниска тј. подниска која садржи све потребне карактере).

Алгоритам ће прво пронаћи прву исправну подниску (ако таква постоји). Затим ће се обрађивати једна по једна релевантна позиција надесно, и за њу ће се одређивати најкраћа исправна подниска која се на њој завршава, тако што ће се кретати од најкраће исправне подниске која се завршава на претходној позицији, затим ће се она проширивати надесно да укључи карактере до тренутне позиције и затим ће се из те подниске избацивати карактери са почетка, све док се не наиђе на релевантан карактер који се не појављујује касније. Та подниска мора бити најкраћа од свих исправних подниски које се завршавају на тренутној позицији, јер би се избацивањем тог карактера изгубило својство да подниска садржи све карактере из скупа S тј. подниска не би више била исправна.

Пример. У примеру `xCxxBxxBxxAxxBxxCxxxxBxAxAxCxxxBx` разматрали бисмо подниску `C`, затим `CxxB`, затим `CxxBxxB` и затим `CxxBxxBxxA` који садржи све потребне карактере (она је најкраћа од свих подниски који се завршавају на тој позицији).

Преласком на следећу релевантну позицију добија се `CxxBxxBxxAxxB` и то је најкраћа од свих подниски који се на тој позицији завршавају (зато што се почетно `C` јавља само једном и не сме се уклонити).

Следећа позиција је позиција наредног слова `C`. Конструкцију најкраће подниске која се завршава на тој позицији започињемо тако што проширимо претходну подниску надесно и добијамо `CxxBxxBxxAxxBxxC`. Затим уклањамо почетно `C` и карактере `x` иза њега (јер се `C` јавља у подниски и касније), чиме добијамо `BxxBxxAxxBxxC`, затим уклањамо и `Bxx` (јер се `B` још два пута јавља) и поново уклањамо `Bxx` (јер се `B` још једном јавља). Када се дође до `AxxBxxC` она је најкраћа подниска који се завршава на позицији тог слова `C` јер се `A` јавља само на њеном почетку и не сме се уклонити.

Истим поступком добија се да је најкраћа подниска која се завршава на наредној позицији `AxxBxxCxxxxB`, на наредној позицији је то `CxxxxBxA`, на наредној позицији је то `CxxxxBxAxA`, затим `BxAxAxC`, и на крају `AxCxxxB`. Пошто смо за сваку позицију пронашли најкраћу подниску, можемо одредити и дужину глобално најкраће и то је 7 (ту дужину имају подниске `AxxBxxC`, `BxAxAxC` и `AxCxxxB`).

Тренутна подниска биће одређена помоћу два итератора i и j низа релевантних позиција, а пошто за сваки карактер треба да знамо колико пута се јавља унутар подниске уместо скупа карактера тренутне подниске користићемо мапу која сваки карактер пресликава у његов број појављивања. У језику C++ пресликавање можемо представити низом од 26 целих бројева и додатним бројачем који показује колико елемената тог низа је различито од нуле. Ипак, елегантније решење је ако се употреби библиотечка имплементација мапе (`map` или `unordered_map`). Употреба мапе описана је, на пример, у задатку Фреквенције речи.

Анализа сложености. Оценимо сада и сложеност овог алгоритма. Спољна петља по j пролази кроз све релевантне позиције којих има највише $O(n)$. Кључна опаска за оцену сложености је да се унутрашња петља (у којој се врши скраћивање сегмента који се завршава на позицији j) може извршити укупно $O(n)$ пута (i је све време мање или једнако j и стално се увећава, а никада не умањује). У оквиру петљи врши се ажурирање вредности у мапи, али с обзиром на то да та операција има константну сложеност (пошто је распон типа `char` ограничен и веома мали), а број карактера у скупу S је мали, можемо рећи да је сложеност алгоритма $O(n)$, тј. да је алгоритам линеаран у односу на дужину текста.

```
// "beskonacno"
const int INF = numeric_limits<int>::max();
```

```
// pronalazi duzinu najkrace podniske koja sadrzi sve date karaktere iz skupa S
int duzinaPodniske(const string& niska, const string& S) {
  // pozicije unutar niske na kojima su karakteri iz skupa S
  vector<int> relevantne_pozicije;
  for (int i = 0; i < niska.size(); i++)
    // ako se niska[i] nalazi u skupu karakteri
    if (S.find(niska[i]) != string::npos)
      // zapamti njegovu poziciju i
      relevantne_pozicije.push_back(i);

  // najmanja duzina podniske
  int min_duzina = INF;
  // broj pojavljivanja svakog relevantnog karaktera u trenutnoj podniski
  map<char, int> broj_pojavljivanja_u_podniski;
  // Trenutna podniska je odredjena pozicijama [i, j]
  vector<int>::const_iterator i, j;
  for (i = j = relevantne_pozicije.begin(); j != relevantne_pozicije.end(); j++){
    // trenutnu podnisku prosirujemo do sledece pozicije j i
    // uvecavamo broj pojavljivanja karaktera koji se nalazi na
    // poziciji odredjenoj sa j (jer se i on sada javlja u podniski)
    broj_pojavljivanja_u_podniski[niska[*j]]++;
    // ako mapa ima isto elemenata kao i skup S, onda su svi elementi
    // skupa S prisutni u podniski
    if (broj_pojavljivanja_u_podniski.size() == S.size()) {
      // trazimo najkracu podnisku koji se zavrsava na poziciji
      // odredjenoj sa j tako sto podnisku skracujemo sa leve strane
      // dok god se prvi karakter podniske javlja vise puta u njemu
      while (broj_pojavljivanja_u_podniski[niska[*i]] > 1) {
        // podnisku skracujemo i uklanjamo sve karaktere sa pocetka sve do
        // sledeceg karaktera iz skupa S
        broj_pojavljivanja_u_podniski[niska[*i]]--;
        i++;
      }
      // izracunavamo duzinu trenutnog podteksta
      int duzina = *j - *i + 1;
      // azuriramo minimum ako je to potrebno
      if (duzina < min_duzina)
        min_duzina = duzina;
    }
  }

  return min_duzina;
}
```

Задатак: Двоструко сортирана претрага

Дата је матрица у којој су све врсте и све колоне сортиране растући. Напиши програм који ефикасно проналази елементе у таквој матрици.

Улаз: Са стандардног улаза уносе се димензије матрице m и n ($1 \leq m, n \leq 1000$), а затим и елементи матрице (елементи сваке врсте у посебном реду, раздвојени размацима). Елементи су цели бројеви између -10^5 и 10^5. Након тога учитава се у сваком реду до краја улаза по један број који се тражи у матрици.

Излаз: За сваки број који се тражи у матрици на стандардни излаз исписати колико пута се појављује у матрици.

Пример

Улаз	*Излаз*
4 5 1 3 5 8 10 4 7 9 11 15 5 9 13 14 20 8 11 14 16 22 11 12 13	2 0 1

Решење

Линеарна претрага

Наиван начин је да сваки елемент тражимо линеарном претрагом, тако што у угнежђеним петљама пролазимо кроз сваки елемент матрице.

Анализа сложености. Сложеност овог приступа била би $O(m \cdot n)$.

```
int brojPojavljivanja(const vector<vector<int>>& A, int x) {
  int broj = 0;
  int m = A.size(), n = A[0].size();
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++)
      if (A[i][j] == x)
        broj++;
  }
  return broj;
}
```

Бинарна претрага

Пошто су елементи сваке врсте сортирани, на претрагу сваке врсте могуће је применити поступак бинарне претраге. Тај поступак је детаљно описан, на пример, у задатку Провера бар-кодова.

Ако претрагу вршимо по врстама, онда можемо употребити библиотечку функцију. Ако бисмо претрагу вршили по колонама, тада не бисмо могли да применимо библиотечку функцију.

Анализа сложености. Сложеност тог приступа је $O(m \cdot \log n)$. Ако има много више врста него колона, ефикасније би било претрагу вршити по колонама (сложеност би тада била $O(n \cdot \log m)$).

```
int brojPojavljivanja(const vector<vector<int>>& A, int x) {
  int broj = 0;
  int m = A.size();
  for (int i = 0; i < m; i++)
    if (binary_search(A[i].begin(), A[i].end(), x))
      broj++;
  return broj;
}
```

Претрага из доњег левог угла

Задатак веома елегантно и ефикасно у линеарној сложености можемо решити техником два показивача (где суштински одсецамо велике делове претраге).

Посматрајмо део полазне матрице (подматрицу) у чијем се доњем левом углу налази елемент a_{vk} и који се простире до горњег десног угла полазне матрице. Пошто су елементи у врстама и колонама строго растући, сви елементи у колони k изнад елемента a_{vk} су мањи од њега, док су сви елементи у врсти v десно од елемента a_{vk} већи од њега.

На почетку посматрамо целу матрицу (тј. почињемо од елемента a_{vk} за $v = m - 1$ и $n = 0$).

- Ако је елемент a_{vk} већи од траженог, тада се тражени елемент не може налазити у последњој врсти посматране подматрице и претрагу можемо наставити у подматрици којој је доњи леви угао $a_{(v-1)k}$ (одсецамо целу последњу врсту). Ако је нова подматрица празна (што се дешава када је $v = 0$), претрагу није потребно даље настављати. Претрагу нове подматрице настављамо на потпуно исти начин као и полазне (тако што умањимо индекс v за 1).
- Ако је елемент a_{vk} мањи од траженог, тада се тражени елемент не може налазити у првој колони посматране подматрице и претрагу можемо наставити у подматрици којој је доњи леви угао $a_{v(k+1)}$ (одсецамо целу прву колону). Ако је нова подматрица празна (што се дешава када је $k = n - 1$), претрагу није потребно даље настављати. Претрагу нове подматрице настављамо на потпуно исти начин као и полазне (тако што увећамо индекс k за 1).
- Ако је елемент a_{vk} једнак траженом, тада можемо увећати бројач појављивања траженог елемента. Пошто су елементи у врстама и колонама строго растући, знамо да се елемент не може налазити ни изнад ни десно од елемента a_{vk}, тако да наставак претраге можемо наставити у подматрици којој је доњи десни угао $a_{(v-1)(k+1)}$ (одсецамо целу прву врсту и последњу колону). Ако је нова подматрица празна (што се дешава када је $k = n - 1$), претрагу није потребно даље настављати. Претрагу нове подматрице настављамо на потпуно исти начин као и полазне (тако што увећамо индекс k за 1).

Пример. Прикажимо како да пребројимо појављивања елемента 11 у следећом матрици.

```
1 3 5 8 10
4 7 9 11 15
5 9 13 14 20
8 11 14 16 22
```

Претрага тече на следећи начин.

- Крећемо из доњег левог угла и поредимо 8 и 11. Пошто је 8 мање од 11, мањи од 11 су и сви елементи прве врсте (јер су врсте и колоне сортиране) и целу прву колону можемо надаље занемарити. Зато се померамо удесно.
- Наишли смо на елемент 11, па увећавамо његов број појављивања. Пошто су врсте и колоне сортиране строго растуће, знамо да се ни изнад тог елемента 11, ни десно од њега не могу налазити нови елементи 11, па другу колону и последњу, четврту врсту можемо надаље занемарити. Зато се померамо горе-десно.
- Поредимо елемент 13 и 11. Пошто је 13 веће од 11, знамо да се десно од тог елемента 13 не могу налазити елементи 11, па целу претпоследњу, трећу врсту можемо надаље занемарити. Померамо се навише.
- Поредимо елемент 9 и 11. Пошто је 9 мање од 11, знамо да се изнад елемента 9 не могу налазити елементи 11, па целу трећу колону можемо надаље занемарити. Померамо се надесно.
- Наишли смо још један елемент 11. Знамо да се ни изнад ни десно од њега не могу налазити нови елементи 11. Зато елиминишемо целу четврту колону и другу врсту.
- Поредимо једини преостали елемент 10. Он је мањи од 11, па се померамо надесно. Пошто тиме излазимо из опсега матрице, претрага се завршава.

Рецимо и да је могуће дуално решење у ком би се уместо елемента у доњем левом, посматрао елемент у горњем десном углу матрице.

Анализа сложености. Сложеност овог приступа у најгорем случају је $O(m + n)$. Напоменимо да је ово решење лошије од бинарне претраге када имамо мали број прилично дугачких врста.

```
int brojPojavljivanja(const vector<vector<int>>& A, int x) {
  int broj = 0;
  int m = A.size(), n = A[0].size();
  int v = m - 1, k = 0;
  while (v >= 0 && k < n)
    if (A[v][k] < x)
      k++;
    else if (A[v][k] > x)
      v--;
```

```
    else {
      broj++;
      v--; k++;
    }
  return broj;
}
```

Претходни поступак сасвим природно може бити описан и рекурзивном функцијом.

```
int brojPojavljivanja(const vector<vector<int>>& A, int x, int v, int k) {
  if (v < 0 || k >= A[0].size())
    return 0;
  if (A[v][k] < x)
    return brojPojavljivanja(A, x, v, k+1);
  else if (A[v][k] > x)
    return brojPojavljivanja(A, x, v-1, k);
  else
    return 1 + brojPojavljivanja(A, x, v-1, k+1);
}
```

Глава 3

Конструкција алгоритама рекурзијом тј. индукцијом

Један од основних механизама конструкције алгоритама подразумева да се до решења проблема долази тако што се проблем сведе на решавање једног или више потпроблема истог облика, али мање димензије. Свођење, наравно, не може да тече у недоглед, већ је потребно да проблеме мале димензије умемо да решимо директно, без даљег свођења. На пример, проблем димензије 0 се решава директно, док се за свако $n > 0$, проблем димензије n своди на проблем димензије $n - 1$.

Имплементација овог поступка може бити реализована на два начина.

- Могуће је дефинисати рекурзивну функцију (функције која позива саму себе), којој се преко улазних параметара (уз евентуално коришћење додатних, глобалних променљивих) прослеђује опис проблема који се тренутно решава. Унутар функције се врши анализа да ли је прослеђени проблем довољно мале димензије да би се могао директно решити или се његово решење добија тако што функција позове сама себе да реши један или више мањих потпроблема.
- Могуће је дефинисати итеративни поступак, који подразумева да се у петљи променљиве ажурирају, кренувши од решења проблема мале димензије па проширујући решење мало по мало, све док се не дође до решења проблема тражене димензије.

Без обзира на то да ли се користи рекурзивна или итеративна имплементација, у основни оваквих алгоритама лежи исти поступак и њихова коректност следи на основу приниципа математичке индукције. Случај који се директно решава представља базу индукције. Индуктивна је претпоставка да су потпроблеми коректно решени и на основу те индуктивне претпоставке се доказује да се полазни проблем коректно решава. Стога ћемо често говорити о **индуктивно-рекурзивној** конструкцији и често ћемо приказивати и рекурзивну и нерекурзивну имплементацију (до које ћемо долазити било директно, било ослобађањем рекурзије).

3.1 Извођење итеративних алгоритама из рекурзивних

Задатак: Грејов код

Грејов кôд реда n подразумева ређање свих n-тоцифрених бинарних записа тако да се свака два суседна записа разликују тачно у једном биту (при чему ово важи и за први и последњи запис, тако да се може сматрати да су сви записи поређани у круг).

Грејов кôд дужине 0 садржи само један елемент и то празну ниску. Грејов кôд дужине $n+1$ се може добити од кода дужине n тако што се испред сваког броја у коду дужине n допише цифра 0, затим се редослед елемената у коду дужине n обрне и на сваком броју се на почетак допише цифра 1 и два тако добијена низа бројева се споје. Нпр. Грејов кôд реда 2 је

```
00
01
11
10
```

На основу претходног поступка добијамо Грејов кôд реда 3.

```
0 00  |                k
0 01  |                0:  000
0 11  |                1:  001
0 10  v                2:  011
             tj.       3:  010
1 10  ^                4:  110
1 11  |                5:  111
1 01  |                6:  101
1 00  |                7:  100
```

Заиста, у коду дужине $n+1$ бројеви у првој половини сви почињу нулом, па се разликују тачно у биту у ком се разликују одговарајући кодови у Грејовом коду дужине n, бројеви у другој половини сви почињу јединицом, па и за њих важи исто, док се последњи број прве и први број друге половине разликују само на почетном биту, а исто важи и за први број прве и последњи број друге половине.

Напиши програм који за дату дужину кода n и дату позицију k ($0 \leq k < 2^n$) одређује бинарни број који се налази на позицији k у коду дужине n.

Улаз: Са стандардног улаза се учитава дужина кода n ($1 \leq n \leq 32$) и позиција k ($0 \leq k < 2^n$).

Излаз: На стандардни излаз исписати тражени бинарни број.

Пример 1

Улаз	*Излаз*
3	011
2	

Пример 2

Улаз	*Излаз*
30	10011001010111101011010000000
999999999	

Решење

Дефинишимо функцију која одређује k-ти по реду запис Грејовог кода дужине n, који има 2^n бројева (подразумеваћемо да је $0 \leq k < 2^n$). Њу је једноставно дефинисати рекурзивно.

Ако је n нула, резултат је празна ниска. У супротном треба израчунати неки елемент Грејовог кода дужине $n-1$ и затим га допунити слева нулом или јединицом. Треба разликовати случај елемената у првој и у другој половини листе кодова. Пошто укупно има 2^n кодова, елементи у првој половини су на позицијама $0 \leq k < 2^{n-1}$, док су елементи у другој половини на позицијама $2^{n-1} \leq k < 2^n$.

- Када је $0 \leq k < 2^{n-1}$, тада се враћа k-ти елемент Грејовог кода дужине $n-1$ допуњен почетном нулом.
- Када је $2^{n-1} \leq k < 2^n$ тада се враћа $(2^n - 1 - k)$-ти елемент Грејовог кода дужине $n-1$ допуњен почетном јединицом. Изразом $2^n - 1 - k$ се позиција k своди у распон $[0, 2^{n-1})$ и уједно се обрће редослед бројева. Наиме, операцијом $k - 2^{n-1}$ вршимо редукцију интервала $[2^{n-1}, 2^n)$ на интервал $[0, 2^{n-1})$. Генерално, приликом обртања редоследа елемената, свака позиција p у интервалу $[0, m)$ се пресликава у позицију $m-p-1$ (позиција 0 се слика у $m-1$, док се $m-1$ слика у 0). Стога се приликом обртања интервала $[0, 2^{n-1})$ позиција $k-2^{n-1}$ слика у $2^{n-1}-(k-2^{n-1})-1$, но то је једнако 2^n-1-k.

Израчунавање степена двојке најједноставније се врши битовским операцијама (при чему треба обратити пажњу на потенцијално прекорачење).

Резултат можемо представити у облику ниске карактера. Иако дописивање карактера на почетак ниске може бити неефикасна операција, с обзиром на то да су ниске са којима радимо прилично кратке (најдужа има 32 карактера), о том не морамо да бринемо.

```
string grej(unsigned n, unsigned k) {
   if (n == 0)
      return "";
   if (k < (1u << (n - 1)))
      return "0" + grej(n - 1, k);
   else
      return "1" + grej(n - 1, (1ul << n) - 1 - k);
}
```

Функцију можемо реализовати и итеративно. Током итерације у промењивој `rez` налазиће се префикс траженог броја дужине $n_0 - n$, где је n_0 почетна, а n тренутна дужина кода. У зависности од тога да ли је текућа

позиција k испод или изнад средине, префикс ћемо проширивати (здесна) нулом или јединицом, и уместо рекурзивног позива вредности k и n ћемо ажурирати вредностима које би се наводиле у рекурзивном позиву.

```
string grej(unsigned n, unsigned k) {
   string rez = "";
   while (n > 0) {
     if (k < 1u << (n-1))
        rez = rez + "0";
     else {
        rez = rez + "1";
        k = (1ul << n) - 1 - k;
     }
     n--;
   }
   return rez;
}
```

Напоменимо и да се тражени број у Грејовом коду може израчунати и директно, ако се представи у облику неозначеног броја изразом `k ^ (k >> 1)` (због водећих нула дужина n тада није битна). Овим се врши ексклузивна дисјункција позиције k и броја добијеног шифтовањем битова те позиције удесно. Неозначени број можемо затим претворити у ниску карактера и издвојити њен суфикс дужине n.

```
string grej(unsigned n, unsigned k) {
  return bitset<32>(k ^ (k >> 1)).to_string().substr(32 - n, n);
}
```

Задатак: Звезда

Особа је *звезда* (енгл. *superstar*) у некој групи људи ако њу сви остали познају, а она не познаје никога. Написати програм који одређује да ли у датој групи људи постоји звезда и која је то особа. Иако је за учитавање података потребно време које квадратно зависи од броја присутних особа, алгоритам треба да ради у времену које линеарно зависи од броја особа.

Улаз: Са стандардног улаза се учитава број особа присутних на забави n ($1 \leq n \leq 100$), а затим матрица димензије $n \times n$ која на позицији (i, j) тј. у врсти i и колони j садржи 1 ако особа i познаје особу j тј. 0 ако је не познаје. На дијагонали матрице се налазе јединице (свака особа познаје саму себе).

Излаз: На стандардни излаз исписати редни број особе која је звезда, ако звезда постоји (особе се броје од 0 до $n - 1$) или -1 ако звезда не постоји.

Пример

Улаз	*Излаз*
3 1 1 0 0 1 0 1 1 1	1

Објашњење

Особа на позицији 1 је звезда, јер она не познаје ни особу на позицији 0, ни особу на позицији 2 (у врсти 1 су све нуле, осим на позицији 1), а њу познају и особа на позицији 0 и особа на позицији 2 (у колони 1 су све јединице).

Решење

Директан начин је да за сваку особу проверимо да ли задовољава услов звезде.

Анализа сложености. Сложеност најгорег случаја овог алгоритма је $O(n^2)$, мада се тај најгори случај ретко јавља, јер је за очекивати да ће се обе провере у случају када кандидат није звезда прекинути много пре него што се стигне до краја низа.

```
bool poznajeNekog(const vector<vector<bool>>& poznaje, int i) {
  for (int j = 0; j < poznaje[i].size(); j++) {
    if (i != j && poznaje[i][j])
```

```
      return true;
  }
  return false;
}

bool sviJePoznaju(const vector<vector<bool>>& poznaje, int i) {
  for (int j = 0; j < poznaje.size(); j++) {
    if (i != j && !poznaje[j][i])
      return false;
  }
  return true;
}

int zvezda(const vector<vector<bool>>& poznaje) {
  for (int i = 0; i < poznaje.size(); i++)
    if (!poznajeNekog(poznaje, i) && sviJePoznaju(poznaje, i))
      return i;
  return -1;
}
```

Побољшање можемо покушати индуктивно-рекурзивним приступом. Прва идеја је да проблем проналажења звезде у скупу од n особа сводимо на проблем проналажења звезде у скупу од $n-1$ особа.

- База индукције је празан скуп особа у коме не постоји звезда (приметимо и да једночлан скуп увек садржи звезду, но тај случај нема потребе засебно разматрати).
- Претпоставимо као индуктивну хипотезу да умемо да израчунамо звезду у скупу од $n-1$ особа. Раздвојмо скуп од n особа на подскуп од почетних $n-1$ особа и последњу особу o_{n-1}. Звезда целог скупа може бити или звезда тог подскупа или особа o_{n-1} (јер ако је особа звезда неког скупа онда је она уједно и звезда сваког подскупа којем припада).

  1. Ако у подскупу од $n-1$ особа постоји звезда, да би она била звезда целог скупа потребно је да је познаје особа o_{n-1} и да она не познаје особу o_{n-1}, то се може проверити веома брзо и једноставно.
  2. У супротном (ако у подскупу не постоји звезда или ако звезда постоји, али она познаје особу o_{n-1} или особа o_{n-1} не познаје њу) морамо још испитати да ли је особа o_{n-1} звезда. То питање никако не зависи од тога да ли у подскупу постоји звезда и да би се то испитало потребно је посебно проверити да ли свих претходних $n-1$ особа познаје особу o_{n-1} и да ли она не познаје никога од њих. Нажалост, овај услов не можемо ефикасно проверити.

Анализа сложености. Ако скуп од првих $n-1$ особа садржи звезду, у времену $O(1)$ можемо проверити и да ли скуп од n особа садржи звезду. Међутим, ако скуп од $n-1$ особа не садржи звезду, тада је потребно време $O(n)$ да би се испитало да ли је особа o_{n-1} звезда. Тај се случај може догађати већ од тренутка када се појаве две особе, па је једначина која описује време извршавања $T(n) = T(n-1) + O(n)$, $T(1) = O(1)$, па је сложеност алгоритма $O(n^2)$.

```
int zvezda(const vector<vector<bool>>& poznaje, int n) {
  if (n == 0)
    return -1;
  int z = zvezda(poznaje, n-1);
  // nismo imali zvezdu u manjem skupu ili
  // imali smo zvezdu koju poslednja osoba ne poznaje ili ona poznaje poslednju osobu
  if (z != -1 && poznaje[n-1][z] && !poznaje[z][n-1])
    return z;
  // proveravamo da li je poslednja osoba zvezda
  if (!poznajeNekog(poznaje, n-1) && sviJePoznaju(poznaje, n-1))
    return n-1;
  return -1;
}

int zvezda(const vector<vector<bool>>& poznaje) {
```

```
  return zvezda(poznaje, poznaje.size());
}
```

Из претходне рекурзивне конструкције лако је елиминисати рекурзију, али алгоритам остаје неефикасан.

```
int zvezda(const vector<vector<bool>>& poznaje) {
  int z = -1;
  for (int i = 0; i < poznaje.size(); i++) {
    // nismo imali zvezdu medju do sada obradjenim osobama ili
    // imali smo zvezdu koju tekuca osoba ne poznaje ili ona poznaje tekucu osobu
    if (z == -1 || !poznaje[i][z] || poznaje[z][i]) {
      // proveravamo da li je tekuca osoba zvezda
      if (!poznajeNekog(poznaje, i) && sviJePoznaju(poznaje, i))
        z = i;
      else
        z = -1;
    }
  }
  return z;
}
```

Основна идеја ефикасног решења је да се проблем посматра "уназад". Број особа које нису звезде је сигурно много већи од броја особа које јесу звезде, па је идентификовање не-звезде много једноставније од идентификовања звезде. Запитајмо се да ли две особе могу бити звезде. Претпоставимо да у скупу постоје две звезде. Ако прва од њих познаје другу, онда прва није звезда, а ако друга познаје прву онда друга није звезда. Дакле, у скупу може постојати највише једна звезда и кључна идеја ове ефикасне индуктивне конструкције је да веома брзо и једноставно (само једним питањем) из сваког скупа можемо уклонити особу за коју знамо да није звезда и на тај начин смањити димензију проблема. Када у скупу остане само једна особа, она је једини кандидат да буде звезда полазног скупа (јер су све остале особе елиминисане на основу тога што смо утврдили да не могу бити звезде). За ту једину преосталу особу онда можемо директно испитати да ли је звезда или није. Елиминација не-звезде из скупа се може извршити крајње једноставно. Одаберемо произвољне две особе у скупу и питамо се да ли особа A зна особу B. Ако је одговор потврдан, онда особа A не може бити звезда (јер звезда никога не познаје). Ако је одговор одричан, онда особа B није звезда (јер звезду сви познају).

Анализа сложености. Алгоритам заснован на овом поступку задовољава једначину $T(n) = O(1)+T(n-1)$, чије је решење $O(n)$. После овога, преостаје још да се провери да ли је преостали једини кандидат стварно звезда. То је могуће урадити грубом силом за шта нам је довољно $2n-2$ питања ($n-1$ питање којим се проверава да ли остале особе познају тог кандидата и $n-1$ питање којим се проверава да ли кандидат познаје остале особе), тако да је сложеност и ове фазе $O(n)$, па је укупна сложеност обе фазе $O(n)$. Ипак, ако се у обзир узме и фаза учитавања матрице, за шта је потребно $O(n^2)$ операција, види се да је укупна сложеност програма $O(n^2)$, тако да се, нажалост, ова дивна оптимизација неће осетити у укупном времену извршавања.

Остаје питање техничке реализације, односно питање како да чувамо скуп кандидата који нису још елиминисани и како да из њега бирамо две особе које ћемо поредити. Једна могућност би била да су све особе сложене на један стек. Поредимо две особе са врха стека и на стек враћамо само ону која није елиминисана њиховим поређењем. Поступак настављамо док стек не постане једночлан.

Ипак, биће приказано још једноставније решење које користи два показивача. Први показивач, i, показује на особу која је тренутни кандидат за звезду, тј. на прву особу у низу за коју још није установљено да није звезда. Други показивач, j, показује на особу за коју се проверава да ли је текући кандидат за звезду познаје, тј. на прву особу након позиције i за коју још није утврђено да није звезда. На почетку су показивачи `i` и `j` на суседним позицијама ($i = 0, j = 1$). Особе се обрађују слева на десно секвенцијално.

- Уколико особа o_i не познаје особу o_j, тада је o_i и даље кандидат, а o_j сигурно није звезда (јер сви морају да познају звезду), па се показивач i помера на следећу особу.
- Уколико особа i познаје особу j, онда i није звезда (јер звезда не познаје никога), али j можда јесте, па, пошто између i и j нико није звезда, први следећи кандидат за звезду је текућа особа j и показивач i се помера на j, а j на прву следећу особу (јер особе иза j још нису анализиране, па се за њих не зна да ли су звезде).

```
int zvezda(const vector<vector<bool>>& poznaje) {
  int i = 0, j = 1;
  while (j < poznaje.size()) {
    if (poznaje[i][j])
      i = j;
    j++;
  }
  if (!poznajeNekog(poznaje, i) && sviJePoznaju(poznaje, i))
    return i;
  return -1;
}
```

Доказ коректности. Докажимо формално коректност претходног поступка.

Лема: Инваријанта претходне петље да ниједан елемент у интервалу $[0, i)$ и ниједан елемент у интервалу (i, j) не може бити звезда (при чему је $0 \leq i < j \leq n$).

- На почетку је $i = 0$ и $j = 1$, па су оба интервала празна, а услов важи (под претпоставком да је $n > 0$).
- Претпоставимо да услов важи при уласку у петљу.
  - Ако особа i познаје особу j тада она не може бити звезда. Пошто на основу претпоставке знамо да звезде не могу бити ни особе $[0, i)$ као ни особе (i, j) и како је $i' = j$, знамо да након тела петље звезде сигурно нису ни особе у интервалу $[0, i')$. Пошто је $j' = j + 1$, у том случају је интервал (i', j') празан, па тривијално задовољава услов.
  - Ако особа i не познаје особу j тада знамо да j не може да буде звезда. Тада је $i' = i$, а $j' = j+1$, па на основу претпоставке знамо да звезде не могу бити особе из интервала $[0, i')$, знамо и да не могу бити особе из интервала (i', j), а пошто ни j не може бити звезда, знамо да звезде не могу бити особе из интервала (i', j'). Однос између променљивих тривијално остаје очуван у оба случаја.

Теорема: Функција или исправно проналази звезду или враћа -1 ако звезда не постоји.

Када се петља заврши није $j < n$, па на основу инваријанте важи да је $j = n$. Тада знамо да звезде не могу бити особе из интервала $[0, i)$ као ни особе из интервала (i, n). Дакле, једини кандидат за звезду је особа i. За њу се експлицитно проверава тражени услов, тако да функција враћа коректну вредност.

Задатак: Апсолутни победник на гласању

Апсолутни победник избора је онај ко освоји више од половине гласова изашлих бирача. Ако су познати сви гласачки листићи, одреди да ли постоји апсолутни победник избора и који је то кандидат (нагласимо да је апсолутни победник, ако постоји, јединствен тј. да није могуће да постоје два различита апсолутна победника).

Улаз: Са стандардног улаза се уноси број гласача n, а затим и гласови (сваки глас представља шифру неког кандидата - цео број из интервала $[0, 10^9]$).

Излаз: На стандардни излаз исписати број победника ако постоји апсолутни победник, тј. `nema` у супротном.

Пример 1

Улаз	*Излаз*
`10` `342 123 342 756 123 756 123 756 756 756`	`nema`

Пример 2

Улаз	*Излаз*
`13` `342 123 342 756 123 756 123 756 756 756 342 756 756`	`756`

Решење

Овај проблем се у литератури назива и *majority voting*.

Пребројавање гласова за сваког кандидата

Очигледан начин да се реши задатак је да се преброје сви гласови за сваког кандидата и да се пронађе да ли је неки кандидат освојио више од пола гласова. Бројање можемо извршити помоћу асоцијативног низа. Слична

техника је коришћена у задацима Фреквенција знака и Фреквенције речи. У језику C++ можемо употребити колекцију `unordered_map` засновану на хеширању или `map` засновану на балансираним бинарним стаблима.

Анализа сложености. Сложеност приступа заснованог на бројању гласова зависи од сложености уметања у структуру података која пресликава кандидате у њихове освојене бројеве гласова. Ако је та структура података заснована на хеширању, та сложеност може бити константна у просеку (иако је у најгорем случају линеарна), а ако је заснована на балансираним стаблима, она је $\log n$. Зато укупна сложеност варира од $O(n)$ до $O(n \log n)$, уз коришћење $O(m)$ меморије потребне за чување асоцијативног низа, где је m број различитих кандидата (обратимо пажњу на то да оригинални низ гласова није потребно чувати).

```
int apsolutniPobednik(const vector<int>& glasovi) {
  // brojimo glasove za svakog kandidata
  unordered_map<int, int> broj_glasova;
  for (int glas : glasovi)
    broj_glasova[glas]++;

  // proveravao da li neki kandidat ima vise od n/2 glasova
  int pobednik = -1;
  for (auto it : broj_glasova)
    if (it.second > glasovi.size() / 2) {
      pobednik = it.first;
      break;
    }

  return pobednik;
}
```

Сортирање

Још један начин може бити заснован на учитавању свих гласова, његовом сортирању, и затим провери броја узастопних једнаких елемената (анализирањем серије узастопних елемената). Разни начини сортирања описани су у задатку Сортирање бројева, а одређивање најдуже серије узастопних елемената описано је у задатку Најдужа серија победа.

Анализа сложености. Сложеност овог приступа зависи од сложености сортирања и једнака је $O(n \log(n))$ ако се користи библиотечко сортирање. Одређивање најдуже серије једнаких елемената врши се у времену $O(n)$.

```
int apsolutniPobednik(const vector<int>& glasovi) {
  // ukupan broj glasova
  int n = glasovi.size();
  // pravimo sortiranu kopiju niza glasova
  auto glasovi_s = glasovi;
  // sortiramo glasove za sve kandidate
  sort(begin(glasovi_s), end(glasovi_s));

  // duzina tekuce serije jednakih elemenata
  // element glasovi[0] zapocinje prvu seriju
  int broj_glasova = 1;
  for (int i = 1; i <= n; i++)
    if (i == n || glasovi_s[i] != glasovi_s[i-1]) {
      // serija jednakih elemenata je upravo zavrsena
      if (broj_glasova > n/2) {
        // ako u upravo zavrsenoj seriji ima vise od n/2 jednakih,
        // onda je u njoj apsolutni pobednik
        return glasovi_s[i-1];
      }
      // element glasovi[i] je zapoceo novu seriju
      broj_glasova = 1;
    } else
      // element glasovi[i] je nastavio tekucu seriju
```

```
      broj_glasova++;

  // ne postoji pobednik
  return -1;
}
```

Средишњи елемент

Неколико алгоритама подразумевају да сачувамо низ гласова, да у првом пролазу на неки начин одредимо потенцијалног кандидата за апсолутног победника, а да у другом пролазу проверимо да ли је тај кандидат заиста освојио потребан број гласова, тако што избројимо његов број гласова и проверимо да ли је већи од $\frac{n}{2}$.

Једна могућност да нађемо кандидата за апсолутног победника је да одредимо средишњи елемент у низу (када се низ сортира, то је онај елемент који се налази на позицији $\lfloor \frac{n}{2} \rfloor$ или на позицији $\lceil \frac{n}{2} \rceil$.

На пример, ако има 7 гласова, занима нас елемент на позицији 3.

```
0 1 2 3 4 5 6
. . . x . . .
```

Ако има 8 гласова, занима нас елемент на позицији 4 (а могли бисмо посматрати и елемент на позицији 3).

```
0 1 2 3 4 5 6 7
. . . x x . . .
```

Наиме, ако постоји апсолутни победник није могуће да он не заузме то средишње место (у случају парне димензије низа, заузимаће оба средишња места). На пример, ако има 7 гласова, апсолутни победник мора да има бар 4 гласова и они ће у сортираном редоследу бити узастопни. Ако би почињали на позицији 0, сигурно би захватили и позицију 3, ако би се завршавали на позицији 6, опет би сигурно захватили позицију 3, а слично важи и за било коју могућност између те две граничне ситуације. Слично, ако на пример има 8 елемената, апсолутни победник има бар 5 гласова, што значи да обухвата и позицију 6 и позицију 7 (било да почиње на позицији 0, да се завршава на позицији 7 или у било ком случају између та два гранична).

Средишњи елемент можемо у језику C++ можемо одредити библиотечком функцијом `nth_element` (чија је сложеност линеарна). Она врши парцијално сортирање и прима три итератора: први и трећи итератор ограничавају распон који се сортира, док је други итератор који указује на неку позицију између њих. Након примене ове функције гарантовано је да се на позицији у низу на коју указује други итератор налази вредност која би се на тој позицији нашла и након сортирања целог распона, као и да сви елементи лево од те позиције имају мању или једнаку вредност од те позиције (не обавезно у сортираном редоследу).

Ручну имплементацију је најбоље реализовати алгоритмом QuickSelect.

Тај алгоритам је описан у задатку Збир k најбољих.

```
int apsolutniPobednik(const vector<int>& glasovi) {
  // posto se redosled elemenata niza menja, moramo da napravimo kopiju
  auto Glasovi = glasovi;
  int n = Glasovi.size();
  // odredjujemo sredisnji element u sortiranom nizu - ako postoji
  // apsolutni pobednik on sigurno zauzima i sredisnju poziciju
  nth_element(begin(Glasovi), begin(Glasovi) + n / 2, end(Glasovi));
  int kandidat = Glasovi[n / 2];

  // proveravamo da li je kandidat ostvari vise od pola glasova
  if (count(begin(Glasovi), end(Glasovi), kandidat) > n / 2)
    return kandidat;
  else
    return -1;
}
```

Бојер-Муров алгоритам

У наставку ћемо прказати сасвим елементаран и изразито елегантан алгоритам за решење овог проблема који су увели Бојер и Мур у раду *"A fast majority vote algorithm"* (интересантно, у оригиналном раду алгоритам је уведен у циљу приказа могућности аутоматске формалне верификације софтвера).

Покушајмо да решимо проблем индуктивно-рекурзивном конструкцијом. Претпоставимо да желимо да испитамо постојање апсолутног победника у скупу од n гласова. Формално, наравно, ради се о мултискупу (јер може да садржи поновљене елементе), али ћемо једноставности ради у наредном опису користити термин скуп. Питање је како проблем свести на проблем мање димензије. Класично свођење проблема са димензије n на димензију $n-1$ не доводи до решења, јер избацивање било ког појединачног гласа може да промени решење (јер ако апсолутни победник има за један глас више од свих осталих, након избацивања тог гласа он више неће бити апсолутни победник). Самим тим, ако као резултат рекурзивног позива добијемо информацију да мањи скуп нема апсолутног победника, то нам никако не помаже да сазнамо да ли полазни, шири скуп има апсолутног победника.

Бојер-Муров алгоритам се заснива на следећој идеји. Замислимо поступак у којем би се гласови за различите кандидате међусобно потирали и нестали. Уколико постоји апсолутни победник тада ће након потирања остати бар један (а можда и више) гласова за тог водећег кандидата. Покажимо ово мало прецизније.

Лема: Ако из неког скупа који има апсолутног победника избацимо било која два различита гласа, онда ће и мањи скуп имати истог апсолутног победника.

Доказ коректности. Докажимо наведену лему. Ако постоји апсолутни победник p у ширем скупу од n гласова и он има m гласова, онда је $m > n/2$.

- Ако су из скупа избачена два гласа која нису за апсолутног победника онда n има и даље m гласова, а редуковани скуп има $n-2$ елемента, па важи $m > n/2 > (n-2)/2$ па апсолутни победник није промењен.
- Ако је избачен један глас за особу p и један који није за њу, тада p има $m-1$ глас, а у скупу има $n-2$ елемента, из $m > n/2$ следи $m-1 > n/2-1 = (n-2)/2$, па апсолутни победник није промењен.

Обратимо пажњу на то да постојање апсолутног победника у мањем скупу не имплицира то да у већем скупу постоји апсолутни победник. На пример, ако су гласови $[1,1,1,2,2,3,3]$ апсолутног победника нема, међутим, избацивањем гласова $2,3$ добијамо скуп $[1,1,1,2,3]$, у коме је кандидат 1 апсолутни победник.

Дакле, избацивањем два различита гласа, редукујемо димензију проблема. Ако у тако добијеном мањем скупу нема апсолутног победника, нема га ни у већем, а ако мањи скуп има апсолутног победника, он је једини кандидат за апсолутног победника у већем скупу, међутим, потребно је накнадно експлицитно проверити да ли је тај кандидат заиста апсолутни победник у већем скупу.

Излаз из овог суштински рекурзивног поступка настаје када не можемо више избацивати парове различитих елемената. Празан скуп нема апсолутног победника, а непразан скуп у коме не постоје два различита елемента садржи гласове само за једног кандидата који је очигледно апсолутни победник.

Описали смо, дакле, индуктивно-рекурзивну конструкцију која нам може омогућити да добијемо кандидата за апсолутног победника (што ће се десити ако скуп гласова избацивањем парова различитих гласова сведемо на непразан скуп у ком су сви гласови за истог кандидата) или да утврдимо да апсолутни победник не постоји (што ће се десити ако скуп гласова избацивањем парова различитих гласова сведемо на празан скуп). Ова рекурзивна конструкција нам даје могућност имплементације решења, међутим, питање проналажења две различите особе у скупу и њиховог избацивања није рачунски тривијално, па ћемо имплементацију направити на други начин.

На основу ове рекурзивне конструкције, следи коректност следећег тврђења. Претпоставимо да је неки скуп гласова подељен на два дисјунктна подскупа, таква да се у једном од њих налазе парови гласова таквих да се у сваком пару налазе гласови за различите особе и да су сви гласови у другом скупу за једну исту особу. Избацивањем једног по једног пара гласова из првог скупа, проблем се рекурзивно своди на све мању и мању димензију, све док не остану само елементи другог скупа. На основу описане рекурзивне конструкције следи да ако је други скуп празан, онда не постоји апсолутни победник у почетном скупу свих гласова, а ако није, онда је особа за коју су сви гласови у другом скупу једини кандидат за апсолутног победника.

Алгоритам ћемо добити индуктивном конструкцијом, тако што ћемо кренути од празног скупа особа подељеног на два празна подскупа и у скуп особа који тренутно обрађујемо додавати једну по једну особу полазног скупа, док их све не обрадимо. У сваком кораку ћемо претпоставити да знамо поделу текућег скупа на скуп

парова различитих гласова и скуп гласова за неку (произвољну) особу (тај услов ће бити централна инваријанта петље). Кључни задатак сада је да на основу такве поделе текућег скупа направимо такву поделу скупа који се добија када се текућем скупу дода нека особа.

- Ако је други скуп празан или ако је нова особа једнака његовим елементима, други скуп ћемо проширити новом особом (у другом скупу ће се и даље налазити парови различитих гласова, а сви гласови у првом скупу ће и даље бити за исту особу).
- Ако је други скуп непразан и ако је нова особа различита од његових елемената (који су сви исти), тада један елемент тог другог скупа заједно са новом особом можемо пребацити у први скуп (у првом скупу ће и даље сви гласови бити за исту особу, а у другом ће бити сви ранији парови различитих гласова и овај новододати пар за који смо такође сигурни да је пар гласова за различите особе).

Приметимо да је уместо два скупа довољно само да памтимо особу o за коју су сви гласови из првог скупа, као и број тих гласова b.

- Ако је $b = 0$ или ако је глас за нову особу једнак o, увећаћемо b за 1. Ако је $b = 0$, ажурирамо вредност o.
- У супротном смањујемо b за 1.

Ако је $b = 0$ након обраде свих гласова апсолутни победник не постоји. У супротном је o једини кандидат за апсолутног победника. Пошто немамо гаранције да ће последњи кандидат за апсолутног победника бити стварно апсолутни победник, та се провера мора експлицитно извршити на крају програма.

Пример. Размотримо гласове $1, 2, 1, 3, 2, 2, 3, 2, 2$ и прикажимо како се два скупа (тј. њихова репрезентација помоћу променљивих o и b) мењају додавањем једног по једног елемента.

glas	prvi skup	drugi skup	o	b
	[]	[]	-	0
1	[]	[1]	1	1
2	[(1, 2)]	[]	1	0
1	[(1, 2)]	[1]	1	1
3	[(1, 2), (1, 3)]	[]	1	0
2	[(1, 2), (1, 3)]	[2]	2	1
2	[(1, 2), (1, 3)]	[2, 2]	2	2
3	[(1, 2), (1, 3), (2, 3)]	[2]	2	1
2	[(1, 2), (1, 3), (2, 3)]	[2, 2]	2	2
2	[(1, 2), (1, 3), (2, 3)]	[2, 2, 2]	2	3

Кандидат је, дакле, 2, а накнадна провера пребројавањем његових гласова показује да је он апсолутни победник (јер има 5 гласова, од укупних 9).

Анализа сложености. Обе фазе (и одређивање кандидата за апсолутног победника и провера да ли је он апсолутни победник) су сложености $O(n)$, док су меморијски захтеви такође $O(n)$ (јер морамо запамтити све гласове, да бисмо на крају проверили да ли је кандидат заиста апсолутни победник).

```
int apsolutniPobednik(const vector<int>& glasovi) {
  // odredjujemo kandidata za pobednika glasove delimo u dve grupe: u
  // prvoj grupi se svi glasovi mogu ponistiti tako sto se spajaju dva
  // po dva razlicita glasa, a u drugoj grupi su svi glasovi za
  // kandidata za pobednika

  // broj glasova u drugoj grupi
  int broj = 0;
  // kandidat za pobednika
  int kandidat;
  for (int glas : glasovi) {
    // druga grupa je prazna
    if (broj == 0) {
      // trenutni glas je kandidat za pobednika i ubacujemo ga u drugu
      // grupu
      kandidat = glas;
      broj = 1;
```

```
    } else if (glas == kandidat)
      // trenutni glas je za kandidata i ubacujemo ga u drugu grupu
      broj++;
    else
      // trenutni glas moze da se ponisti sa jednim glasom za
      // kandidata i oba ih prebacujemo u prvu grupu
      broj--;
  }

  // proveravamo da li postoji kandidat za pobednika i da li je
  // ostvario vise od n/2 glasova
  if (broj > 0 &&
      count(begin(glasovi), end(glasovi), kandidat) > glasovi.size() / 2)
    return kandidat;
  else
    return -1; // nema pobednika
}
```

Задатак: Цикличко померање за k места улево

Дати низ од n целих бројева циклички померити (ротирати) за k места улево.

Улаз: У првој линији стандардног улаза налази се природан број k ($1 \leq k \leq 10^5$) који представља број места за која се низ помера улево, у другој природан број n ($1 \leq n \leq 10^5$), који представља број елемената низа, а затим се у следећих n линија налазе цели бројеви у границама од -1000 до 1000 који представљају елементе низа.

Излаз: У n линија стандардног излаза исписати елементе низа који се добија цикличким померањем учитаног низа за k места улево.

Пример

Улаз	*Излаз*

`3`
`10`
`1 2 3 4 5 6 7 8 9 10`
	`4 5 6 7 8 9 10 1 2 3`

Решење

Пошто се цикличним померањем за n места низ враћа у своју оригиналну позицију, није тешко уочити да је цикличко померање за k места улево исто што и цикличко померање за $k \bmod n$ места улево. Зато ћемо у свим наредним решењима претпоставити да се након учитавања броја `k` он мења вредношћу `k % n` и да важи да је $0 \leq k < n$.

Задатак се може решити на заиста много различитих начина, који се разликују по количини додатне меморије коју користе и по броју операција које је потребно применити. С обзиром на потенцијално велике димензије улазног низа, пожељно је користити она решења која низ трансформишу у месту тј. не користе помоћне низове, и код који је број операција линеаран (тј. пропорционалан броју n).

Наивна решења

k цикличких померања за једно место улево

Решење је засновано на k цикличких померања за једно место улево. Потребно је само цикличко померање за једно место улево поновити k пута.

Анализа сложености. Ово решење не користи додатни низ (користи само једну помоћну целобројну променљиву), али мана му је велики број операција које се извршавају. Број додела које се извршавају отприлике је једнак nk, што може бити превише за велике вредности n и k (пошто број k може имати исти ред величине као и n, овај алгоритам у најгорем случају има квадратну сложеност).

```
void rotirajUlevoZaJednoMesto(vector<int>& a, int n) {
  // rotiranje za niza za jedno mesto ulevo
  int pom = a[0];
  for (int j = 0; j < n - 1; j++)
```

```
    a[j] = a[j + 1];
  a[n - 1] = pom;
}

void rotirajUlevoZaKMesta(vector<int>& a, int n, int k) {
  // k puta ponavljamo rotiranje za jedno mesto
  for (int i = 0; i < k; i++)
    rotirajUlevoZaJednoMesto(a, n);
}
```

Оптимална решења

Коришћење библиотечких функција

Језик C++ нуди функцију `rotate` која ротира елементе низа тако да дати елемент постане први. Функцији се прослеђују итератор који указује на почетак низа (или вектора), итератор који указује на елемент који треба да постане први и итератор који указује непосредно иза краја низа.

```
// rotiramo niz primenom bibliotecke funkcije
rotate(begin(a), next(begin(a), k), end(a));

// da je koriscen niz, a ne vektor, poziv bi bio
// rotate(a, next(a, k), next(a, n));
```

Размена блокова без коришења помоћне меморије

Ако би низ имао $2k$ елемената, ротација за k места би се вршила тако што би се разменили блокови елемената који чине прву и другу половину низа. У општем случају ситуација је мало компликованија, али се поново може свести на размену блокова елемената. Нека низ има n елемената и нека га ротирамо за k места улево. Нека првих k елемената чине блок који ћемо означити са L, а преосталих $n-k$ елемената чине блок који ћемо означити са D. Размотримо следеће случајеве, у зависности од односа дужина та два блока.

- Ако су блокови L и D једнаке дужине, њиховом разменом се добија тражено решење.
- Ако је блок L краћи, означимо са D_1 почетни део блока D дужине k, а са D_2, преостали део блока D. Елементи блока D_1 треба да буду почетни елементи у траженом решењу и да бисмо то постигли, можемо их разменити са елементима блока L. Тиме из ситуације LD_1D_2 долазимо у ситуацију D_1LD_2, док је тражено решење облика DL, тј. D_1D_2L. Да бисмо то постигли, потребно је да низ LD_2 заротирамо за дужину блока L улево (а то је опет k), тј. да разменимо блокове L и D_2, што је проблем истог облика, али мање димензије од полазног.
- Ако је блок L дужи, означимо са L_1 почетни део блока L дужине $n-k$, а са L_2, преостали део блока L. Елементи блока D треба да буду почетни елементи у траженом решењу и да бисмо то постигли, можемо их разменити са елементима блока L_1. Тиме из ситуације L_1L_2D долазимо у ситуацију DL_2L_1, док је тражено решење облика DL, тј. DL_1L_2. Да бисмо то постигли, потребно је да низ L_2L_1 заротирамо за дужину блока L_2 улево (а то је $2k-n$), тј. да разменимо блокове L_2 и L_1, што је проблем истог облика, али мање димензије од полазног.

Дакле, задатак се може решити индуктивно-рекурзивном конструкцијом. Први случај можемо подвести под други (или трећи) јер се комплетан леви блок замењује са десним, након чега остаје да се размене празни блокови, што не производи никакав ефекат, па излаз из рекурзије може бити случај када је било који од блокова празан.

Претпоставићемо да на располагању имамо функцију `razmeni` која у датом низу или вектору размењује блокове исте дужине који се не преклапају и који почињу на две дате позиције. Ту функцију је веома једноставно имплементирати. У језику C++ може се искористити библиотечка функција `swap_ranges` која као аргументе прима два итератора која ограничавају први блок и итератор који указује на почетак другог блока.

Решење се може испрограмирати рекурзивно.

```
// razmenjujemo u vektoru v blok duzine d koji pocinje na
// poziciji p1 sa blokom duzine duzina koji pocinje na poziciji p2
void razmeni(vector<int>& v, int p1, int p2, int d) {
  swap_ranges(next(v.begin(), p1), next(v.begin(), p1 + d),
```

```
            next(v.begin(), p2));
}


// razmenjujemo u vektoru v blok L koji pocinje na poziciji pl i
// duzine je dl i blok D koji pocinje na poziciji pd i duzine je dd
void razmeniBlokove(vector<int>& v, int pl, int dl, int pd, int dd) {
  // ako je neki blok prazan nema potrebe za razmenom
  if (dl == 0 || dd == 0)
     return;
  if (dl <= dd) {
     // razmenjujemo kompletan levi blok sa pocetkom desnog
     razmeni(v, pl, pd, dl);
     // iz situacije L.D1.D2 dosli smo u situaciju D1.L.D2 i
     // da bismo dosli u zeljenu situaciju D1.D2.L moramo
     // da razmenimo L i D2
     razmeniBlokove(v, pl + dl, dl, pd + dl, dd - dl);
  } else {
     // razmenjujemo kompletan desni blok sa pocetkom levog
     razmeni(v, pl, pd, dd);
     // iz situacije L1.L2.D dosli smo u situaciju D.L2.L1 i
     // da bismo dosli u zeljenu situaciju D.L1.L2 moramo
     // da razmenimo L1 i L2
     razmeniBlokove(v, pl + dd, dl - dd, pd, dd);
  }
}

void rotiraj(vector<int>& v, int k) {
  // razmenjujemo pocetni blok duzine k i ostatak niza
  razmeniBlokove(v, 0, k, k, v.size() - k);
}
```

Пример. Размотримо следећи пример. Желимо да ротирамо следећи низ $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ за 4 места улево.

Након ротације требало би да добијемо $[5, 6, 7, 8, 9, 10, 1, 2, 3, 4]$.

- Ротација се врши тако што размењујемо леви блок $L = [1, 2, 3, 4]$ са десним блоком $D = [5, 6, 7, 8, 9, 10]$. Важи да је $p_l = 0$, $d_l = k = 4$, $p_d = k = 4$ и $d_d = n - k = 6$. Пошто је леви блок краћи од десног блока, можемо заменити цео блок L са блоком $D_1 = [5, 6, 7, 8]$. Тако добијамо $[5, 6, 7, 8, 1, 2, 3, 4, 9, 10]$.

  На тај начини бројеви $[5, 6, 7, 8]$ су дошли на своје место, а да би се од овог низа добио крајњи резултат, потребно је у делу низа иза њих разменити блок $L = [1, 2, 3, 4]$ и блок $D_2 = [9, 10]$.

- Сада је $L = [1, 2, 3, 4]$, а $D = [9, 10]$. Важи да је $p_l = 4$, $d_l = 4$, $p_d = 8$ и $d_d = 2$. Пошто је леви блок дужи од десног блока, размењујемо цео блок D са блоком $L_1 = [1, 2]$ и тако добијамо $[5, 6, 7, 8, 9, 10, 3, 4, 1, 2]$.

  На тај начин су бројеви $[9, 10]$ дошли на своје место, а да би се од овог низа добио крајњи резултат, потребно је у делу низа иза њих разменити блокове $L_2 = [3, 4]$ и $L_1 = [1, 2]$.

- Сада је $L = [3, 4]$ и $D = [1, 2]$. Важи да је $p_l = 6$, $d_l = 2$, $p_d = 8$ и $d_d = 2$. У овом случају је дужина оба блока једнака, па се након њихове размене добијамо $[5, 6, 7, 8, 9, 10, 1, 2, 3, 4]$ и процедура се зауставља.

Анализа сложености. Докажимо да се алгоритам зауставља и проценимо му сложеност. Централна мера прогреса у алгоритму је укупна дужина блокова који се размењују. Укупна дужина креће од n и смањује се све док не дође до нуле. У првом случају се врши размена блокова дужине d_l за шта је потребно $O(d_l)$ корака и након тога се врши рекурзивни позив такав да је збир дужина блокова управо за d_l мањи од полазног збира дужина (нови збир је $d_l + (d_d - d_l) = d_d$, а полазни $d_l + d_d$). Слично, у другом случају се врши замена блокова дужине d_d за шта је потребно $O(d_d)$ корака и након тога се врши рекурзивни позив такав да је збир дужина блокова управо за d_d мањи од полазног збира дужина (нови збир је $(d_l - d_d) + d_d = d_l$, а полазни $d_l + d_d$). Дакле, рекурентна једначина је у оба случаја једнака $T(n) = T(n - d) + O(d)$, где је $d = \min(d_l, d_d)$ и њено

решење је $T(n) = O(n)$.

Још једноставније, у сваком функције `razmeni` се помоћу d размена тачно d елемената доводи на своје финално место, одакле се више не мрда, а у рекурзивној функцији се осим тога (и рекурзивних позива) не врши никакав други посао. Пошто се по завршетку рекурзије сваки елемент налази на свом месту извшено је тачно n размена.

Рекурзија је репна и лако је се можемо ослободити. Могућа је и додатна оптимизација, јер почетне позиције блокова који се размењују не морамо посебно памтити, већ их можемо увек израчунати ако знамо дужине блокова који се размењују и знамо да се они налазе у завршном делу низа. Илустрације ради, приказаћемо мало другачије итеративно решење у ком ћемо претпоставити да у сваком кораку блок који треба да се нађе на крају низа доводимо на његово место, а да након тога размене преосталих блокова вршимо на почетку низа.

Претпоставимо да је у сваком кораку петље потребно разменити блокове L и D који се састоје од почетних d_l, па затим d_d елемената низа. Вредности d_l и d_d се иницијализују на k и $n - k$, а затим се у петљи понавља следеће.

- Ако су d_l и d_d једнаки, тада се размењују блокови који почињу на позицијама 0 и d_l и који су дужине $d_l = d_d$ и петља се прекида.
- Ако је $d_l < d_d$, низ је облика LD_1D_2, где је $|D_2| = |L| = d_l$. Разменићемо блокове L и D_2. Први почиње на позицији 0, а други на позицији $|LD_1| = d_l + (d_d - d_l) = d_d$. Након тога ћемо добити низ D_2D_1L и преостаће нам да разменимо блокове D_2 и D_1, који су на почетку низа. Дужина блока D_2 је d_l, а блока D_1 је $d_d - d_l$. Зато се вредност d_d умањује за d_l, док се d_d не мења.
- Ако је $d_l > d_d$, низ је облика L_1L_2D, где је $|L_2| = |D| = d_d$. Разменићемо блокове L_2 и D. Први почиње на позицији $|L_1| = d_l - d_d$, а други на позицији $|L_1L_2| = d_l$. Након тога ћемо добити низ L_1DL_2 и преостаће нам да разменимо блокове L_1 и D. Дужина блока L_2 једнака је дужини блока D и износи d_d, па је дужина блока L_1 једнака $d_l - d_d$. Зато се d_l умањује за d_d док се d_d не мења.

Пример. Применимо описани алгоритам на још једном примеру. Нека је дат низ $[1, 2, 3, 4, 5]$ и $k = 3$.

- У почетку је $d_l = k = 3$ и $d_d = n - k = 2$, па треба разменити блокове $L = [1, 2, 3]$ и $D = [4, 5]$. Пошто је $d_d < d_l$, размењује се се блок $L_2 = [2, 3]$, који почиње на позицији $d_l - d_d = 1$ и блок $D = [4, 5]$, који почиње на позицији $d_l = 3$. Оба су дужине $d_d = 2$. Тако се добија низ $[1, 4, 5, 2, 3]$. Након тога се d_l умањује за d_d и постаје 1.
- Сада је $d_l = 1$ и $d_d = 2$, па треба разменити блокове $L = [1]$ и $D = [4, 5]$. Пошто је $d_l < d_d$, размењују се леви блок $L = [1]$, који почиње на позицији 0 и десни блок $D_2 = [5]$, који почиње на позицији $d_d = 2$ (оба су дужине $d_l = 1$). Тако се добија низ $[5, 4, 1, 2, 3]$. Након тога се d_d умањује за d_l и постаје 1.
- Сада је $d_l = d_d = 1$. Зато се размењују блокови $[5]$ и $[4]$, који почињу на позицији 0 тј. $d_l = 1$, чиме се низ доводи у жељену конфигурацију $[4, 5, 1, 2, 3]$.

Анализа сложености. Приметимо да је сложеност овог поступка линеарна, јер се у сваком кораку помоћу d размена бар d елемената доводи на своје коначно место и елиминише из даље обраде.

```
// razmenjujemo u vektoru "v" blok duzine "duzina" koji pocinje na
// poziciji "p1" sa blokom duzine d koji pocinje na poiciji "p2"
void razmeni(vector<int>& v, int p1, int p2, int d) {
  for (int i = 0; i < d; i++)
      swap(v[p1+i], v[p2+i]);
}

// funkcija rotira elemente niza a, duzine n za k mesta ulevo
void RotirajUlevo(vector<int>& a, int k) {
  // broj elemenata niza
  int n = a.size();

  // ciklicko pomeranje za k mesta je isto sto i ciklicko pomeranje za
  // k % n mesta
  k %= n;
```

```
  // vrsimo rotiranje niza
  int dl = k;
  int dd = n-k;

  while (true) {
    if (dl < dd) {
      // niz je oblika LD1D2, |L|=|D2| i razmenjujemo L i D2 tako da
      // dobijamo D2D1L
      razmeni(a, 0, dd, dl);
      dd -= dl;
    } else if (dl > dd) {
      // niz je oblika L1L2D, |L1|=|D| i razmenjujemo L2 i D tako da
      // dobijamo L1DL2
      razmeni(a, dl-dd, dl, dd);
      dl -= dd;
    } else {
      // niz je oblika LD, |L|=|D| i razmenjujemo L i D cime
      // zavrsavamo postupak
      razmeni(a, 0, dl, dl);
      break;
    }
  }
}
```

Базирано на идеји размене блокова, тј. на претходним индуктивно-рекурзивним конструкцијама можемо формулисати још један алгоритам (са доста елегантном имплементацијом). Претпоставимо да имамо на располагању промењиве l и d које указују на почетак блокова L и D и промењиве k и n које указују на њихове крајеве (прецизније, позиције иза њихових крајева).

На почетку је $l = 0$ и $d = k$.

У петљи размењујемо елементе на позицијама l и d, увећавајући, при том, оба индекса. Током поступка ће се догодити или да индекс l достигне k или да индекс d достигне n.

- Ако се обе ствари догоде истовремено, блокови су били исте дужине и успешно смо их разменили, чиме је посао завршен.
- Ако l достигне k пре него што d достигне n, тада је леви блок био краћи и у овој ситуацији је заправо размењен са почетним делом десног блока, чиме се дошло у конфигурацију D_1LD_2. Преостало је још разменити блокове L и D_2. Почетак блока L је на месту тренутног индекса l, а крај му је тик испред текућег индекса d и зато ажурирамо вредност k на вредност d. Почетак и крај дела D_2 су на својим исправним вредностима (d и n).
- Ако d достигне n пре него што l достигне k, тада је десни блок био краћи и у овој ситуацији је заправо размењен са почетним делом левог блока, чиме се дошло у конфигурацију DL_2L_1. Преостало је још разменити блокове L_2 и L_1. Почетак блока L_2 је на месту тренутног индекса l, а крај му је тик испред текућег индекса k. Међутим, почетак блока L_1 је на месту текућег индекса k и зато је потребно вредност d поставити на k, док је крај блока L_1 тик испред индекса n.

Дакле, алгоритам може имати наредну, веома елегантну имплементацију.

```
void RotirajUlevo(vector<int>& a, int k) {
  // broj clanova niza
  int n = a.size();

  // ciklicko pomeranje za k mesta je isto sto i ciklicko pomeranje za
  // k % n mesta
  k %= n;

  // vrsimo rotiranje niza razmenom blokova
  int l = 0; // pocetak levog bloka
  int d = k; // pocetak desnog bloka
```

```
  // razmenjujemo blokove L i D tj. blokove [l, k) u [d, n)
  while (true) {
    // menjamo tekuce elemente u blokovima
    swap(a[l++], a[d++]);

    // stigli smo do kraja oba bloka, pa je posao zavrsen
    if (l == k && d == n)
      break;

    if (d == n) // stigli smo do kraja desnog bloka
      // razmenili smo pocetak levog bloka sa desnim i dosli u
      // situaciju DL2L1 i jos treba da razmenimo blokove L2 i L1, a
      // to su blokovi [l, k) u [k, n), tako da d treba postaviti na k
      d = k;

    if (l == k) // stigli smo do kraja levog bloka
      // razmenili smo levi blok sa pocetkom desnog i dosli u situaciju
      // D1LD2 i jos treba da razmenimo blokove L i D2, a to su blokovi
      // [l, d) u [d, n), tako da k treba postaviti na d
      k = d;
  }
}
```

Анализа сложености. Сложеност је линеарна у односу на дужину низа и не користи се додатни меморијски простор осим помоћне промењиве у склопу размене два елемента низа.

Пример. Применимо описани алгоритам на примеру. Нека је дат низ $[1, 2, 3, 4, 5]$ и $k = 3$.

- Бројач l креће од вредности 0, бројач d од вредности $k = 3$, тј. креће се са обрадом блокова $[1, 2, 3]$ и $[4, 5]$. У тренутку када бројач d дође до краја (до вредности n), бројач l има вредност 2. Тада је конфигурација низа $[4, 5, 3, 1, 2]$.
- Тада се вредност d поставља на текућу вредност k а то је 3 и прелази се на обраду блокова $[3]$ и $[1, 2]$. Овај пут l стиже до вредности k када d има вредност 4 и тада је конфигурација $[4, 5, 1, 3, 2]$.
- Тада се k помера на 4 и разматрају се блокови $[3]$ и $[2]$. Након њихове размене долази се до краја и тражене конфигурације $[4, 5, 1, 2, 3]$.

Доказ коректности. Веома интересантно би било доказати директно да је овај алгоритам коректан. У наставку ћемо једино доказати да се претходни алгоритам зауставља (што није само по себи тривијално). Једна од инваријанти претходне петље је да је $0 \leq l \leq k \leq d \leq n$. Докажимо то.

- Пошто је $0 < n$, $0 < k$ и $k < n$, на почетку важи $0 = l < k = d < n$, па је инваријанта испуњена.
- Пошто је $0 < k$, на основу услова прекида петље, када се уђе у тело петље знамо да је $0 \leq l < k \leq d < n$. Након тога се l и d увећавају за 1, па након тога важи $0 \leq l \leq k < d \leq n$. Ако је $l = k$ и $d = n$, петља се завршава, а инваријанта важи и након прекида петље. У супротном, ако је $l = k$, тада је $d < n$, док се k поставља на d, па однос $0 \leq l < k \leq d < n$ важи и пре наредне итерације. Слично, ако је $d = n$, тада је $l < k$, док се d поставља на k, па однос $0 \leq l < k \leq d < n$ важи и пре наредне итерације. На основу овога знамо да ова инваријанта остаје испуњена и након извршавања тела петље.

Потребно је још доказати да се у неком тренутку мора догодити да је $l = k$ или да је $d = n$. Међутим, у сваком кораку петље промењива l се увећава за 1, док се горња граница n не мења. Стога знамо да ће се након највише n корака десити да је $l = n$. На основу инваријанте тада ће морати да важи да је $l = k = d = n$, па ће се петља зауставити. Уједно видимо и да се у најгорем случају петља извршава n пута, па је алгоритам сложености $O(n)$.

Задатак: Абацаба

Низ слова $ABACABADABACABAEABACABADABACABAFABACA...$ се може формирати на следећи начин:

1. Низ је на почетку празан.

2. На низ се допише прво велико слово енглеског алфабета које се не појављује у формираном делу низа, а иза тог слова се понове сва слова која су се појавила пре њега.
3. Корак 2 се понови потребан број пута

Тако после прве примене корака 2 добијамо низ A, после друге низ ABA, после треће низ $ABACABA$ итд.

Одредити слово које се појављује на n-том месту у низу, бројећи места од 1. Редослед слова у енглеском алфабету је ABCDEFGHIJKLMNOPQRSTUVWXYZ.

Улаз: Један природан број мањи од 67 108 864.

Излаз: Једно велико слово енглеског алфабета.

Пример 1		**Пример 2**		**Пример 3**	
Улаз	*Излаз*	*Улаз*	*Излаз*	*Улаз*	*Излаз*
`8`	`D`	`65`	`A`	`100`	`C`

Решење

Решење грубом силом је да се у меморији креира цео низ карактера и да се затим прочита карактер са одговарајуће позиције. Ово решење троши превише меморије, а и времена док се дугачак низ карактера не изгради.

```
string s = "";
char slovo = 'A';
while (s.size() <= n - 1) {
  s = s + slovo + s;
  slovo++;
}
cout << s[n-1] << endl;
```

Задатак се може решити без креирања дугачке ниске карактера, ако пажљиво проучимо правилност по којој се слова појављују. Прво слово `A` се налази на месту 1, прво слово `B` на месту `2`, прво слово `C` на месту `4`, прво слово `D` на месту `8` итд., па се може наслутити да се прва појављивања слова налазе на местима која су степени броја 2.

Заиста, ако дужину низа карактера пре уметања новог слова абецеде обележимо са d_k, важи да је $d_0 = 0$ (пре уметања слова `A` не налази се ниједан карактер) и да је $d_{k+1} = 2d_k + 1$ (пре уметања новог слова налази се ниска која је добијена тако што је претходно слово спојено са два појављивања ниске која се појављивала пре тог претходног слова). Зато је $d_k = 2^k - 1$ (важи да је $2^0 - 1 = 0$ и да је $2^{k+1} - 1 = 2 \cdot (2^k - 1) + 1$), док је прва позиција слова k (ако се слова броје од један) једнака 2^k.

Дакле, ако је унети број n неки степен двојке $n = 2^k$, онда је у питању место на ком се слово први пут појављује и важи да је $k = \log_2 n$ ($\log_2 n$ означава број k такав да је $2^k = n$, нпр. $\log_2 32 = 5$, јер је $2^5 = 32$). Знајући k слово лако можемо одредити сабирајући k са ASCII/UNICODE кодом слова `A`.

У супротном проблем можемо свести на проблем мање димензије. Нека је $2^k < n < 2^{k+1}$, тј. нека је k највећи степен двојке мањи од n. Карактер који тражимо налази се, дакле, иза позиције 2^k у делу низа који је добијен копирањем дела низа испред позиције 2^k. Зато је n-ти карактер у целом низу једнак $(n - 2^k)$-том карактеру у копираном делу, а пошто део који се копира почиње на почетку низа, једнак је $(n - 2^k)$-том карактеру у целом низу.

Овим смо описали рекурзивни поступак којим ефикасно можемо доћи до решења.

$$f(n) = \begin{cases} \log_2 n, \text{за } n = 2^k \\ f(n - 2^k), \text{за } 2^k < n < 2^{k+1} \end{cases} .$$

На пример, $f(23) = f(23 - 16) = f(7) = f(7 - 4) = f(3) = f(3 - 2) = f(1) = 0$, па се на месту 23 налази слово `A`. Слично, на пример, важи да је $f(40) = f(40 - 32) = f(8) = 3$, па се на месту 40 налази слово `D`.

На основу претходне дефиниције би се могла имплементирати рекурзивна функција (за то би било потребно испитати да ли је дати број степен броја 2, наћи логаритам таквог броја, и наћи највећи степен броја 2 од ког је дати број већи или једнак). Ипак, за тим нема потребе. Анализирајући рад такве рекурзивне функције можемо унапред закључити шта ће њен резултат бити, без потребе за њеним извршавањем. Претпоставимо да знамо бинарни запис броја n тј. да знамо да се број n представља као збир неких степенова двојке $2^{s_m} +$

$2^{s_{m-1}}+\ldots+2^{s_0}$. Током рекурзије од броја ће се одузимати један по један степен двојке, све док не остане само 2^{s_0} и тада ћемо знати да је $k = s_0$. Дакле важи да је резултат једнак најмањем степену двојке који учествује разлагању броја на збир степенова двојке тј. да је тражени број k једнак позицији најдешње јединице у бинарном запису броја (претпостављамо да се позиције броје од 0, здесна). До траженог резултата је могуће доћи дељењем броја са 2_0^s, тј. узастопним дељењем броја са 2 све док последњи сабирак не постане 1, тј. све док је број паран (то ће се догодити тачно s_0 пута).

```
int k = 0;
while (n % 2 == 0) {
  n /= 2;
  k++;
}
cout << (char)('A' + k) << endl;
```

Сличан резултат можемо добити и баратајући директно са интерним записом броја у рачунару, коришћењем битовских оператора. Позицију крајње десне јединице једноставно можемо израчунати шифтовањем (померањем) бинарног записа броја удесно за једно место све док последња цифра не постане једнака 1 (последњу цифру можемо испитати битовском конјункцијом са 1).

```
int k = 0;
while ((n & 1) == 0) {
  n >>= 1;
  k++;
}
cout << (char)('A' + k) << endl;
```

Савремени хардвер често поседује и инструкцију *count trailing zeroes* којом се одређује број нула иза последње јединице у бинарном запису (што је баш позиција последње јединице). Иако програмски језици обично не стандардизују приступ овој операцији, неки компилатори нуде подршку за то (на пример, ако се користи GCC, може се употребити функција `__builtin_ctz`, а ако се користи Microsoft Visual C++, може се употребити функција `_BitScanReverse`).

```
cout << (char)('A' + __builtin_ctz(n)) << endl;
```

Задатак: Морзеов низ

Низ $1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, \ldots$, који се састоји од нула и јединица, гради се на следећи начин: први елемент је 1; други се добија логичком негацијом првог $NOT(1) = 0$, трећи и четврти логичком негацијом претходна два $NOT(1) = 0$, $NOT(0) = 1$, пети, шести, седми и осми логичком негацијом прва четири – добија се $0, 1, 1, 0$ итд. Дакле, кренувши од једночланог сегмента 1, сваком почетном сегменту који је дужине 2^k (k узима вредности $0, 1, 2, \ldots$) дописује се сегмент исте дужине добијен логичком негацијом свих елемената почетног сегмента. За задато n одредити n-ти члан низа (бројање креће од 1).

Улаз: У првој линији стандардног улаза налази се природан број n ($1 \leq n \leq 10^9$).

Излаз: На стандарном излазу приказати цифру (0 или 1) на позицији n

Пример 1

Улаз	*Излаз*
15	0

Пример 2

Улаз	*Излаз*
1234	0

Пример 3

Улаз	*Излаз*
12345678	1

Решење

Формирање целог низа

Формулација задатка сугерише решење у коме би се почев од задатог првог елемента $x_1 = 1$ редом формирали сегменти по задатим правилима (x_2), (x_3, x_4), $(x_5 \ldots x_8)$, $(x_9 \ldots x_{16})$, $(x_{17} \ldots x_{32})$, ... Дакле, елементи текућег сегмента се формирају негацијом претходно формираних елемената $x_1 \ldots x_k$, који су на растојању k, где је k степен броја 2 (увећава се дупло након сваког преписаног сегмента).

Сложеност овог приступа је $O(n)$.

Ово уписивање се може реализовати помоћу угнежђених петљи где унутрашња петља дуплира садржај низа, а спољашња контролише колико пута садржај треба дуплирати (додатно осигуравајући да се уписивање прекине када се попуни потребних n елемената низа).

```
bool Morzeov(int n) {
  // Morzeov niz
  vector<bool> a(n+1);
  a[1] = true;
  int upisano = 1; // broj upisanih elemenata
  int duzina = 1; // duzina segmenta koji se trenutno negira
  while (upisano < n) {
    // negiramo segment trenutne duzine
    // prekidamo petlju ranije ako tokom toga dostignemo n upisanih elemenata
    for (int i = 1; i <= duzina && upisano < n; i++) {
      a[duzina + i] = !a[i];
      upisano++;
    }
    // naredni segment koji treba negirati je duplo duzi
    duzina *= 2;
  }

  // upisano je n elemenata niza, pa ocitavamo rezultat
  return a[n];
}
```

Низ се може попунити и помоћу само једне петље у којој се врши преписивање елемената док се не упише њих n тако што се на место i преписује елемент са позиције $i - k$, увећавајући степен двојке k када се цео претходни подсегмент препише.

```
bool Morzeov(int n) {
  // Morzeov niz
  vector<bool> a(n+1);
  a[1] = true;
  // rastojanje elemenata koji se negiraju
  int k = 1;
  // popunjavamo niz zakljucno sa pozicijom n
  for (int i = 2; i <= n; i++) {
    // negiramo odgovarajuci element
    a[i] = !a[i - k];
    // negirali smo ceo segment, pa prelazimo na sledeci
    if (i == 2 * k)
      k *= 2;
  }
  // upisano je n elemenata niza, pa ocitavamo rezultat
  return a[n];
}
```

Веза између одговарајућих чланова

Директна решења задатка подразумевају формирање свих елемената низа који претходе n-том члану. Да би се израчунао 2001. члан мора се израчунати свих 2000 претходних чланова.

Уочимо следеће, како бисмо проблем решили без коришћења низа и без израчунавања свих чланова низа који претходе n-том: формирање елемената сегмента на основу претходног поступка, значи да се сваки пут дужина низа удваја, тј. представља степен броја 2.

- Приликом негирања почетног сегмента добијамо да је $x_2 = NOT(x_1)$. У овом случају важи да је $x_n = NOT(x_{n-1})$.
- Негирањем наредног сегмента добијамо да је $x_3 = NOT(x_1)$ и $x_4 = NOT(x_2)$. У овом случају важи да је $x_n = NOT(x_{n-2})$.
- Након тога добијамо да је $x_5 = NOT(x_1)$, $x_6 = NOT(x_2)$, $x_7 = NOT(x_3)$ и $x_8 = NOT(x_4)$. У овом случају важи да је $x_n = NOT(x_{n-4})$.

Дакле, за $n > 1$ важи рекурентна формула $x_n = NOT(x_{n-m})$, где је m - максимални степен броја 2, који је

строго мањи од n. Ова рекурентна формула омогућава веома ефикасно израчунавање траженог члана низа. На пример,

$x_{15} = NOT(x_{15-8}) = NOT(x_7) = NOT(NOT(x_{7-4})) = x_{7-4} = x_3 = NOT(x_{3-2}) = NOT(x_1) = NOT(1) = 0$.

Тражени број се добија у неколико корака и за веће вредности броја n. На пример, за $n = 2001$:

$x_{2001} = NOT(x_{2001-1024}) = NOT(x_{977}) = x_{977-512} = x_{465} = NOT(x_{465-256}) = NOT(x_{209}) = x_{209-128} = x_{81} = NOT(x_{81-64}) = NOT(x_{17}) = x_{17-16} = x_1 = 1$.

Рекурентна формула нам указује да решење можемо веома једноставно реализовати уз помоћу рекурзивне функције. У имплементацији се користи помоћна функција за одређивање траженог степена двојке (ову функцију је могуће имплементирати и ефикасније, коришћењем операција над битовима, међутим и ова једноставна имплементација је сасвим довољна).

Пошто се сваки елемент низа израчунава на основу неког из претходне половине низа, у сваком кораку се n бар полови, тако да је сложеност овог приступа $O(\log n)$.

```
// vraca najveci stepen od 2, koji je manji od n
int MaxStepen2(int n) {
  int max = 1;
  while (max * 2 < n)
    max *= 2;
  return max;
}

// vraca n-ti element Morzeovog niza ako se pozicije broje od 1
bool Morzeov(int n) {
  if (n == 1)
    return true;
  return !Morzeov(n - MaxStepen2(n));
}
```

Рекурзију је могуће једноставно елиминисати и до решења можемо доћи и итеративно тако што ћемо полазећи од $x = 1$, при сваком преласку на индекс настао умањењем за највећи степен броја негирати текућу вредност x.

```
// vraca n-ti element Morzeovog niza ako se pozicije broje od 1
bool Morzeov(int n) {
  bool x = true; // prvi clan niza
  while (n > 1) {
    n -= MaxStep2(n);
    x = !x;
  }
  return x;
}
```

Приметимо да се током претходно описаног поступка уклања један по један бит броја, све док не остане само један бит (тј. док број не постане степен двојке). Након тога се тај бит помера за једно место надесно, све док број не постане 1.

На пример, низ

$$x_{44} = NOT(x_{12}) = x_4 = NOT(x_2) = x_1 = 1$$

се бинарно може представити помоћу

$$x_{101100} = NOT(x_{001100}) = x_{000100} = NOT(x_{000010}) = x_{000001} = 1.$$

Ова друга фаза би се могла избећи, а имплементација поједноставити ако би бројање позиција било од 0, а не од 1 (што лако можемо постићи ако одмах након учитавања умањимо вредност n за 1). Тада би важило

да је $x_1 = NOT(x_0)$, затим $x_2 = NOT(x_0)$, $x_3 = NOT(x_1)$, затим $x_4 = NOT(x_0)$, $x_5 = NOT(x_1)$, $x_6 = NOT(x_2)$ и $x_7 = NOT(x_3)$. Разлика је, дакле, у томе што се од сваког индекса одузима највећи степен двојке који је већи или једнак од датог броја (разлика је значајна баш када је индекс који тренутно разматрамо степен двојке). У тој варијанти бисмо уместо x_{44} рачунали x_{43} и важило би

$$x_{43} = NOT(x_{11}) = x_3 = NOT(x_1) = x_0 = 1$$

тј. бинарно

$$x_{101011} = NOT(x_{001011}) = x_{000011} = NOT(x_{000001}) = x_{000000} = 1.$$

Дакле, на овај начин се у сваком кораку уклања један бит броја, све док број не постане 0, негирајући сваки пут резултујући бит. Увид који нас доводи до лепше имплементације је то да ће се исти резултат добити и када се битови уклањају један по један, кренувши од битова најмање, уместо од битова највеће тежине. Уклањање последњег бита 1 у бинарном запису броја `n` се може урадити веома ефикасно коришћењем израза `n & (n - 1)`.

Можемо приметити да се у овом решењу потврђује да је бројање од 0 уместо од 1 природније и да има лепша математичка својства.

```
// vraca n-ti element Morzeovog niza ako se pozicije broje od 1
bool Morzeov(int n) {
  // prilagodjavamo brojanje tako da krene od 0 umesto od 1
  n--;

  int x = 1;
  while (n) {
    // uklanjamo poslednji bit iz binarnog zapisa broja
    n = n & (n-1);
    x = !x;
  }
  return x;
}
```

На крају, можемо увидети да је битан само број јединица у бинарном запису броја $n - 1$. Ако је тај број паран, резултат је 1, а ако је непаран, резултат је 0. Постоје ефикасни алгоритми да се тај број израчуна, а неки рачунари имају и уграђену хардверску инструкцију за то. Иако програмски језик C++ не стандардизује приступ тој инструкцији, ако се користи компилатор GCC, могуће је употребити функцију `__builtin_popcount`, а ако се користи Microsoft Visual C++, могуће је употребити функцију `__popcnt`.

```
// vraca n-ti element Morzeovog niza ako se pozicije broje od 1
bool Morzeov(int n) {
   // prilagodjavamo brojanje tako da krene od 0 umesto od 1
   n--;
   // ispitujemo parnost ukupnog broja bitova jednakih 1 u binarnom
   // zapisu broja n
   return !(__builtin_popcount(n) & 1);
}
```

Задатак: Избацивање цифара на све начине

Напиши програм који за дати природан број n одређује збир свих бројева који се могу добити избацивањем неких цифара броја n.

Улаз: Са стандардног улаза се учитава број који може да има највише 1000 цифара.

Излаз: На стандардни излаз исписати тражени збир.

Пример

Улаз	*Излаз*
123	177

Објашњење

$123 + 12 + 13 + 23 + 1 + 2 + 3 + 0 = 177$.

Решење

До решења можемо доћи индуктивно-рекурзивном конструкцијом. Ако је број једнак нули, тада је тражени збир једнак нули. У супротном се тај број може разложити на своју последњу цифру и цифре које јој претходе (нпр. број 1234 се може разложити на број 123 и цифру 4). Претпоставимо да умемо да одредимо тражени збир за број коме је уклоњена последња цифра и размотримо како се комбинацијом тог броја и последње цифре може добити тражени збир. У текућем примеру, претпостављамо, дакле, да умемо да одредимо тражени збир за број 123. На сваки од сабирака који учествује у том збиру можемо додати четворку здесна. Сваки од тих сабирака се добија тако што се оригинални број помножи са 10 и дода се 4. Њихов се збир зато може добити тако што се полазни збир (177) помножи са 10 и затим увећа за онолико четворки колико има сабирака (у текућем примеру их има 8). Пошто су овим покривене све могућности, укупан збир се добија сабирањем полазног и овако трансформисаног збира.

```
177     177*10 + 8*4 = 1802          177 + 1802 = 1979

123     1234
12      124
13      134
23      234
1       14
2       24
3       34
0       4
```

У општем случају, дакле, добијени збир префикса множимо са 10 и увећавамо производом последње цифре и броја бројева који се добијају прецртавањем цифара тог префикса. Тај број није тешко израчунати јер је прилично очигледно да k-тоцифрен број може да генерише 2^k бројева (свака од k цифара може бити или прецртана или непрецртана). Зато можемо ојачати индуктивну хипотезу и направити функцију која уз тражени збир враћа још додатно и број цифара броја (или још боље, одговарајући степен двојке).

Дакле, проблем једноставно можемо решити тако што дефинишемо рекурзивну функцију која прима број n а враћа тражени збир за тај број n и одговарајући степен двојке. Више вредности функција може да врати помоћу својих излазних параметара.

Додатни проблем представља величина бројева који су допуштени на улазу, тако да је јасно да имплементација алгоритма са библиотечким типовима података није исправна. Стога имплементација која користи само основне бројевне типове података не ради коректно за све могуће улазе (то се може решити коришћењем великих бројева).

```
void Zbir(int n, int& zbir, int& b) {
  if (n == 0) {
    zbir = 0;
    b = 1;
  } else {
    int zbirR, brojR;
    Zbir(n / 10, zbirR, brojR);
    zbir = zbirR +  zbirR*10 + brojR * (n % 10);
    b = brojR * 2;
  }
}

int Zbir(int n) {
  int zbir, b;
  Zbir(n, zbir, b);
  return zbir;
}
```

Овакве рекурзије је веома једноставно ослободити се и алгоритам се лако може имплементирати и итеративно. Размотримо како се извршава рекурзивни позив за аргумент $n = 123$.

```
Zbir(123)             177, 8
    Zbir(12)          15, 4
       Zbir(1)        1, 2
          Zbir(0)     0, 1
```

Примећујемо, дакле, да се израчунавања врше у повратку кроз рекурзију и то тако што се вредност претходног збира и претходног степена двојке ажурирају на основу описаног поступка. Исто израчунавање се може описати и итеративним поступком у ком се промењиве иницијализују на 0 и 1, а затим током итерације ажурирају коришћењем једне по једне цифре полазног броја, с лева надесно. Пролазак кроз цифре броја у том редоследу једноставнији је ако се број представи као ниска карактера.

```
int Zbir(const string& broj) {
  int zbir = 0, b = 1;
  for (char c : broj) {
    zbir += 10*zbir + b * (c - '0');
    b *= 2;
  }
  return zbir;
}
```

Задатак: Не садрже цифру 3

Напиши програм који одређује колико природних бројева из интервала $[0, n]$ не садрже цифру 3 у свом декадном запису.

Улаз: Прва линија стандардног улаза садржи природан број n ($n \leq 2 \cdot 10^9$).

Излаз: У првој линији стандардног излаза приказати тражени резултат.

Пример 1

Улаз *Излаз*
15 14

Објашњење

У интервалу $[0, 15]$ постоји 16 бројева, а бројеви 3 и 13 једини садрже цифру 3.

Пример 2

Улаз

999

Излаз

729

Пример 3

Улаз

12345

Излаз

8262

Решење

Бројање бројева који не садрже цифру 3

Задатак можемо решити тако што за сваки број од 0 до n проверимо да ли садржи цифру 3, и ако не садржи увећамо бројач бројева који не садрже цифру 3 (тај бројач у почетку иницијализујемо на нулу). У том решењу примењује се алгоритам одређивања броја елемената серије који задовољавају дати услов, тј. алгоритам бројања филтриране серије. Проверу да ли број садржи цифру 3 можемо реализовати у посебној функцији.

```
bool sadrziCifru3(int n) {
  do {
    if (n % 10 == 3)
```

```
      return true;
    n /= 10;
  } while (n > 0);
  return false;
}

int brojeviBezCifre3(int n) {
  int br = 0;
  for (int i = 0; i <= n; i++)
    if (!sadrziCifru3(i))
      br++;
  return br;
}
```

Ефикасније израчунавање броја бројева

Задатак можемо решити и на много ефикаснији начин (али је решење у том случају доста комплексније). Нека $f(a, b)$ означава број бројева из интервала $[a, b]$ који у свом декадном запису не садрже цифру 3, а $f_0(n)$ број таквих бројева из интервала $[0, n]$. Размотримо пример у коме желимо да израчунамо вредност $f_0(4251)$. Све бројеве из интервала $[0, 4251]$ можемо поделити у неколико група тј. подинтервала. Важи да је

$$[0, 4251] = [0, 999] \cup [1000, 1999] \cup [2000, 2999] \cup [3000, 3999] \cup [4000, 4251].$$

Зато је

$$f_0(4251) = f(0, 999) + f(1000, 1999) + f(2000, 2999) + f(3000, 3999) + f(4000, 4251).$$

Важи да је $f(0, 999) = f_0(999)$. Такође, важи и да је $f(1000, 1999)$ једнак броју $f(0, 999)$ тј. $f_0(999)$. Заиста, између интервала $[0, 999]$ и $[1000, 1999]$ може се успоставити бијекција таква да слика у свом декадном запису садржи цифру 3 ако и само ако њен оригинал у свом декадном запису садржи цифру 3. Слично, важи и да је $f(2000, 2999) = f_0(999)$, док је $f(3000, 3999) = 0$ јер сви бројеви у интервалу $[3000, 3999]$ садрже цифру 3. На крају, важи и да је $f(4000, 4251)$ једнако $f(0, 251)$ тј. $f_0(251)$. Зато је

$$f_0(4251) = 3 \cdot f_0(999) + f_0(251).$$

Дакле, ако знамо бројеве $f_0(999)$ и $f_0(251)$ тада можемо израчунати и број $f_0(4251)$.

Иста техника се може применити и на израчунавање бројева $f_0(999)$ и $f_0(251)$.

Важи да је

$$[0, 999] = [0, 99] \cup [100, 199] \cup \ldots \cup [900, 999],$$

па је

$$f_0(999) = f(0, 99) + f(100, 199) + \ldots + f(900, 999).$$

Важи да је $f(0, 99) = f(100, 199) = f(200, 299) = f(400, 499) = \ldots = f(900, 999) = f_0(99)$, а да је $f(300, 399) = 0$. Стога је

$$f_0(999) = 8 \cdot f_0(99) + f_0(99) = 9 \cdot f_0(99).$$

Слично, важи да је

$$f_0(99) = 9 \cdot f_0(9),$$

док је $f_0(9) = 9$ (јер у интервалу $[0, 9]$ који има 10 елемената, једино елемент 3 садржи цифру 3).

Важи и да је

$$[0, 251] = [0, 99] \cup [100, 199] \cup [200, 251],$$

па је

$$f_0(251) = 2 \cdot f_0(99) + f_0(51).$$

Пошто је

$$[0, 51] = [0, 9] \cup [10, 19] \cup [20, 29] \cup [30, 39] \cup [40, 49] \cup [50, 51],$$

важи да је

$$f_0(51) = 4 \cdot f_0(9) + f_0(1).$$

Важи и да је $f_0(1) = 2$ јер су оба елемента интервала $[0, 1]$ такви да не садрже цифру 3.

Дакле, $f_0(4251) = 3 \cdot f_0(999) + f_0(251) = 3 \cdot (9 \cdot f_0(99)) + 2 \cdot f_0(99) + f_0(51) = 3 \cdot (9 \cdot (9 \cdot f_0(9))) + 2 \cdot (9 \cdot f_0(9)) + 4 \cdot f_0(9) + f_0(1) = 3 \cdot 9 \cdot 9 \cdot 9 + 2 \cdot 9 \cdot 9 + 4 \cdot 9 + 2 = 2387$.

Функцију f_0 је могуће рекурзивно дефинисати. Ако се број n разлаже на почетну цифру c и суфикс n' тада се $f_0(n)$ може израчунати на следећи начин, у зависности од цифре c (претпостављамо да број $9 \ldots 9$ има онолико деветки колико цифара има број n').

- Ако је $c < 3$ тада је $f_0(n) = c \cdot f_0(9 \ldots 9) + f_0(n')$,
- Ако је $c = 3$ тада је $f_0(n) = c \cdot f_0(9 \ldots 9)$,
- Ако је $c > 3$ тада је $f_0(n) = (c - 1) \cdot f_0(9 \ldots 9) + f_0(n')$.

Излаз из рекурзије може бити и само случај $f_0(0) = 1$ (0 је једини број у интервалу $[0, 0]$ и он не садржи цифру 3).

Проблем са имплементацијом овакве рекурзивне функције је то што се цифре одвајају слева надесно, што је компликованије него здесна налево, када број n лако разлажемо на $n \operatorname{div} 10$ и $n \bmod 10$. Стога пре уласка у рекурзију можемо обрнути цифре броја. Такође, за дати број n потребно је одредити одговарајући број који се састоји само од деветки. То једноставно можемо решити тако што конструишемо број који се од броја n добија заменом свих цифара цифром 9 и ако тај број прослеђујемо као други параметар рекурзије (тај број у сваком позиву садржи само деветке и то онолико деветки колико цифара има број n).

Приметимо да се од једног рекурзивног позива за број са k цифара најчешће добијају два рекурзивна позива за бројеве са $k-1$ цифара, што указује на то да је сложеност експоненцијална (за основу 2), што је допустиво, јер је број цифара мали. Ипак, с обзиром на то да се исти рекурзивни позиви преклапају (пре свега они облика $f_0(9 \ldots 9)$), имплементација се може убрзати динамичким програмирањем или тако што се примети да је да је $f_0(\underbrace{9 \ldots 9}_{k}) = 9^k$, па би се ови рекурзивни позиви могли специјализовати.

```
int brojeviBezCifre3(int n, int d) {
  if (n == 0)
    return 1;
  int c = n % 10;
  if (c < 3)
    return c * brojeviBezCifre3(d / 10, d / 10) + brojeviBezCifre3(n / 10, d / 10);
  else if (c == 3)
    return c * brojeviBezCifre3(d / 10, d / 10);
  else
    return (c - 1) * brojeviBezCifre3(d / 10, d / 10) + brojeviBezCifre3(n / 10, d / 10);
}

int brojeviBezCifre3(int n) {
```

```
  int nObratno = 0;
  int devetke = 0;
  do {
    nObratno = nObratno * 10 + n % 10;
    devetke = devetke * 10 + 9;
    n /= 10;
  } while (n > 0);

  return brojeviBezCifre3(nObratno, devetke);
}
```

Уместо рекурзије која ради одозго наниже, решење можемо синтетизовати одоздо навише.

Обрађиваћемо једну по једну цифру броја n, здесна налево и мало по мало ћемо проширивати обрађени суфикс n' броја n. Уведимо променљиву, нпр. br, која током итерације чува вредности $f_0(n')$ за суфиксе n' броја n које током итерације обрађујемо, и променљиву, нпр. t, која чува вредности бројева $f_0(9\ldots9) = 9^k$, за бројеве $9\ldots9$ који имају k цифара 9, колико укупно цифара има у тренуном суфиксу n'.

Променљиве t и br иницијализујемо на 1 (претпостављамо да је пре петље обрађен празан суфикс који одговара броју 0 и да је $9^0 = 1$). Затим у петљи обрађујемо цифру по цифру броја n, кренувши од цифре јединица. Променљиву br ажурирамо на следећи начин, у зависности од текуће цифре c.

- Ако је $c < 3$ тада је $br = c \cdot t + br$,
- Ако је $c = 3$ тада је $br = c \cdot t$,
- Ако је $c > 3$ тада је $br = (c - 1) \cdot t + br$.

Променљиву t ажурирамо на вредност $9 \cdot t$.

Размотримо поново пример израчунавања $f_0(4251)$ и применимо претходно описани алгоритам на број 4251.

Претпоставимо да на почетку променљиве t и br имају вредност 1.

Прво обрађујемо последњу (прву с десна) цифру броја n, а то је цифра 1. Пошто је она мања од 3, ажурирамо br на вредност $c \cdot t + br = 1 \cdot 1 + 1 = 2$, а вредност t на вредност $9 \cdot t = 9$. Након овога, променљива br има вредност $f_0(1)$, а променљива t има вредност $f_0(9)$.

Наредна цифра је 5 и пошто је она већа од 3, ажурирамо вредност br на $(c - 1) \cdot t + br = 4 \cdot 9 + 2 = 38$, а вредност t на вредност $9 \cdot t = 81$. Након овога, променљива br има вредност $f_0(51)$, а променљива t има вредност $f_0(99)$.

Наредна цифра је 2 и пошто је она мања од 3, ажурирамо вредност br на $c \cdot t + br = 2 \cdot 81 + 38 = 200$, а вредност t на вредност $9 \cdot t = 9 \cdot 81 = 729$. Након овога, променљива br има вредност $f_0(251)$, а променљива t има вредност $f_0(999)$.

На крају, почетна цифра је 4 и пошто је она већа од 4, ажурирамо вредност br на $(c-1)\cdot t+br = 3\cdot729+200 = 2387$. Вредност t се ажурира на $9 \cdot 729 = 6561$, али се та вредност даље не користи. Након овога, променљива br има вредност $f_0(4251)$, а променљива t има вредност $f_0(9999)$.

```
int brojeviBezCifre3(int n) {
  int t = 1, br = 1;
  while (n > 0) {
    int c = n % 10;
    if (c < 3)
      br = c*t + br;
    else if (c == 3)
      br = c*t;
    else
      br = (c-1)*t + br;
    t = 9*t;
    n /= 10;
  }
  return br;
}
```

Задатак: Максимални збир несуседних елемената низа

Напиши програм који одређује највећи збир подниза датог низа целих бројева који не садржи два узастопна члана низа.

Улаз: Са стандардног улаза се уноси број чланова низа n ($1 \leq n \leq 10^5$), а затим из наредног реда чланови низа раздвојени размацима.

Излаз: На стандардни излаз исписати тражени максимални збир.

Пример 1

Улаз	*Излаз*
`6` `7 3 2 4 1 5`	`16`

Објашњење

Максимални збир је збир сегмента $7 + 4 + 5 = 16$.

Пример 2

Улаз

```
12
3 -2 4 5 -1 3 -4 -5 4 5 2 -1
```

Излаз

```
17
```

Решење

Задатак можемо једноставно решити индуктивно-рекурзивном конструкцијом. Низ можемо разложити на последњи елемент и префикс пре њега. Да бисмо одредили максимални збир несуседних елемената низа, анализираћемо случај када је последњи елемент низа део тог максималног збира и када није (то су једине две могућности и једна од њих је сигурно тачна). Зато је максимални збир несуседних елемената низа максимум следећа два збира:

- првог, који се добија тако што се последњи елемент дода на максимални збир несуседних елемената префикса, при чему у том збиру не укључује претпоследњи елемент низа (тј. последњи елемент префикса) и
- другог, који се добија као максимални збир несуседних елемената префикса који може да укључи и претпоследњи елемент низа (тј. последњи елемент префикса).

Дакле, осим што претпостављамо да умемо да решимо проблем мање димензије тј. да за префикс умемо да одредимо максимални збир несуседних елемената, потребно је да ојачамо индуктивну хипотезу и да за префикс умемо да одредимо и максимални збир несуседних елемената тог префикса ако последњи елемент префикса није укључен у тај збир. Наравно, као и увек када се ојачава индуктивна хипотеза, "дуг" се мора вратити па за префикс проширен додатним елементом поред максималног збира несуседних елемената, морамо да умемо да одредимо и максимални збир несуседних елемената када последњи елемент није укључен. Међутим, то није тешко, јер је то управо максимални збир несуседних елемената префикса. Максимални збир проширеног низа (без обзира на то да ли укључује или не укључује последњи елемент) одређујемо као максимум два описана збира (која на основу индуктивне хипотезе лако израчунавамо).

Излаз из рекурзије је случај празног низа. Максимални збир његових несуседних елемената у свакој варијанти једнак је нули.

Дакле, обележимо са m_k максимални збир несуседних елемената првих k елемената датог низа. Циљ је да одредимо m_n. Нека је m'_k максимали збир несуседних елемената првих k елемената низа у који није укључен елемент a_{k-1}. Важе следеће рекурентне релације: $m_0 = m'_0 = 0$, док за $k > 0$, важи $m'_k = m_{k-1}$ и $m_k = \max(a_{k-1} + m'_{k-1}, m_{k-1})$.

Анализа сложености. Сложеност овог алгоритма је $O(n)$.

```
int maks_zbir_bez = 0;
int maks_zbir = 0;
for (int i = 0; i < n; i++) {
```

```
  int x;
  cin >> x;
  int maks_zbir_sa = maks_zbir_bez + x;
  maks_zbir_bez = maks_zbir;
  maks_zbir = max(maks_zbir_bez, maks_zbir_sa);
}
cout << maks_zbir << endl;
```

Још једна индуктивно-рекурзивна конструкција којом се проблем може решити подразумева да се за сваки префикс низа одреди максимални збир несуседних елемената када је последњи елемент префикса укључен и максимални збир несуседних елемената када последњи елемент префикса није укључен.

Обележимо са m_k^{sa} максимални збир несуседних елемената првих k елемената низа, који садржи и елемент a_{k-1} и са m_k^{bez} максимални збир несуседних елемената првих k елемената низа, који не садржи елемент a_{k-1}. База индукције може бити једночлан низ и важи $m_1^{sa} = a_0$, $m_1^{bez} = 0$. За сваку вредност $k > 1$, важи $m_k^{sa} = a_{k-1} + m_{k-1}^{bez}$ и $m_k^{bez} = \max(m_{k-1}^{sa}, m_{k-1}^{bez})$. Тражена вредност једнака је $\max(m_n^{sa}, m_n^{bez})$.

Анализа сложености. Сложеност овог алгоритма је $O(n)$.

```
// prvi element se ucitava van petlje zbog inicijalizacije
// to je ujedno i baza indukcije, jednoclani prefiks niza
int x;
cin >> x;
int maks_zbir_bez = 0;
int maks_zbir_sa = x;

// u petlji obradjujemo tekuci element
for (int i = 1; i < n; i++) {
  int x;
  cin >> x;
  int maks_zbir = max(maks_zbir_sa, maks_zbir_bez);
  maks_zbir_sa = maks_zbir_bez + x;
  maks_zbir_bez = maks_zbir;
}

cout << max(maks_zbir_bez, maks_zbir_sa) << endl;
```

Индуктивно-рекурзивна конструкција може тећи и на следећи начин. Претпостављамо да умемо да израчунамо максимални збир сваког префикса низа. Ако је префикс празан, максимални збир је нула, а ако је једночлан, тада је једнак већем од броја нула и првог елемента низа. Ако је префикс бар двочлан онда је максимални збир несуседних елемената тог префикса максимум максималног збира префикса без последњег елемента и збира последњег елемента и максималног збира без последња два елемента.

Дакле, добијамо да је $m_0 = 0$, $m_1 = \max(a_0, 0)$ и $m_i = \max(m_{i-1}, a_{i-1} + m_{i-2})$.

Ова рекурзивна конструкција се може испрограмирати рекурзивном функцијом. Нажалост, то решење је прилично неефикасно.

Анализа сложености. Сложеност овог решења задовољава једначину $T(n) = T(n-1) + T(n-2) + O(1)$, $T(1) = T(0) = 1$, чије је решење експоненцијална функција (иста једначина се јавља и приликом директне рекурзивне имплементације израчунавања елемената Фибоначијевог низа).

```
int maksZbir(const vector<int>& a, int n) {
  if (n == 0)
    return 0;
  if (n == 1)
    return max(a[0], 0);
  return max(maksZbir(a, n-1), a[n-1] + maksZbir(a, n-2));
}
```

Рекурзију можемо уклонити и добити наредну итеративну имплементацију.

Анализа сложености. Сложеност овог алгоритма је $O(n)$.

```
int maks_zbir_p = 0;
int x;
cin >> x;
int maks_zbir = max(0, x);
for (int i = 2; i <= n; i++) {
  int x;
  cin >> x;
  int tmp = max(maks_zbir, maks_zbir_p + x);
  maks_zbir_p = maks_zbir;
  maks_zbir = tmp;
}
cout << maks_zbir << endl;
```

Напомена. Систематичан начин ослобађања неефикасности проузроковане рекурзијом овог типа долази у облику техника динамичког програмирања, које је описано у засебном поглављу.

Задатак: Максимални збир сегмента

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Максимални суфикси

Сваки сегмент је суфикс неког префикса низа. Пошто се суфикси могу анализирати инкрементално (сви суфикси $i+1$-вог префикса низа се добијају проширивањем свих суфикса i-тог префикса низа додатним елементом), проблем анализе свих сегмената је пожељно свести на проблем анализе суфикса.

Задатак може да се сведе на то да се за сваки префикс у низу одреди максималан суфикс, а да се онда међу максималним суфиксима за сваки префикс пронађе онај суфикс који је глобално максималан.

Једини суфикс првих нула елемената низа је празан и његов збир је по дефиницији 0. Сви суфикси првих $i+1$ елемената низа, изузев празног суфикса добијају се додавањем елемента на позицији i на крај неког суфикса првих i елемената низа. Међу непразним суфиксима највећи збир има онај који је добијен додавањем последњег елемента управо на суфикс првих i елемената низа који има максимални збир. Од њега једино може бити већи збир празног суфикса (и то када се након проширивања претходно максималног суфикса последњим елементом добије негативан збир). Ако вредности максималног збира суфикса памтимо у низу, тада низ лако попуњавамо на основу веза $S_0 = 0$ и $S_{i+1} = \max(S_i + a_i, 0)$, где је са S_i обележена вредност максималног збира суфикса првих i елемената низа a.

На крају налазимо максимум низа S_i.

Пример. Прикажимо рад алгоритма на примеру низа `-2 3 2 -3 -3 -2 4 5 -8 3`. У таблици попуњавамо вредности S_i.

```
i    a_{i+1}   S_i
0              0
1    -2        0 = max(0+(-2), 0)
2     3        3 = max(0+3, 0)
3     2        5 = max(3+2, 0)
4    -3        2 = max(5+(-3), 0)
5    -3        0 = max(2+(-3), 0)
6    -2        0 = max(0+(-2), 0)
7     4        4 = max(0+4, 0)
8     5        9 = max(4+5, 0)
9    -8        1 = max(9+(-8), 0)
10    3        4 = max(1+3, 0)
```

Максимална вредност у колони S_i је 9.

Анализа сложености. Пошто користимо низ максималних збирова суфикса, меморијска сложеност је $O(n)$. Низ попуњавамо елемент по елемент, инкрементално, у једном пролазу за шта је довољно n корака, а затим максимум налазимо у новом пролазу тј. у нових n корака. Укупна сложеност је, дакле, линеарна тј. $O(n)$.

```
// maksimalni sufiks prvih i elemenata niza
vector<int> maxSufiks(n+1);
maxSufiks[0] = 0;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  maxSufiks[i+1] = max(maxSufiks[i] + x, 0);
}

// maksimalni segment je maksimalni od svih maksimalnih sufiksa
cout << *max_element(begin(maxSufiks), end(maxSufiks)) << endl;
```

Каданов алгоритам

Максималне вредности збирова суфикса не морамо да памтимо у низу, ако њихов максимум одређујемо истовремено са одређивањем вредности максималних збирова суфикса. Овај алгоритам познат је под именом *Каданов алгоритам*.

Један начин да се дође до тог алгоритма је следећи. Покушавамо да алгоритам заснујемо на индуктивној конструкцији.

- За празан низ, једини сегмент је празан и његов је збир нула (то је уједно највећи збир који се може добити).
- Сматрамо да умемо да проблем решимо за произвољан низ дужине n и на основу тога покушавамо да решимо задатак за низ дужине $n + 1$ (полазни низ проширен једним додатним елементом).

  Сегмент највећег збира у проширеном низу се или цео садржи у полазном низу дужине n или чини суфикс проширеног низа, тј. завршава се на последњој позицији (укључујући и могућност да је ту и празан сегмент).

  На основу индуктивне хипотезе знамо да израчунамо највећи збир сегмента низа дужине n и потребно је да још одредимо максимални збир суфикса проширеног низа. Један начин да се то уради је да приликом сваког проширења низа изнова анализирамо све сегменте који се завршавају на текућој позицији, али чак иако то радимо инкрементално (кренувши од празног суфикса, па додајући уназад један по један елемент) највише што можемо добити је алгоритам квадратне сложености (пробајте да се уверите да је то заиста тако). Кључни увид је то да највећи збир суфикса који се завршава на текућој позицији можемо инкрементално израчунати знајући највећи збир суфикса низа пре проширења. Највећи збир неког непразног суфикса који се завршава на текућој позицији је збир текућег елемента низа и највећег збира неког суфикса који се завршава на претходној позицији. Од њега може бити повољнији само празан суфикс (и то само ако је претходни збир негативан).

Дакле, ако са S_i обележимо максимални збир неког суфикса првих i елемената низа, а са M_i максимални збир неког сегмента прих i елемената низа, важи да је $M_0 = S_0 = 0$, да је $S_{i+1} = \max(S_i + a_i, 0)$ и $M_{i+1} = \max(M_i, S_{i+1})$.

Имплементацију можемо направити итеративним алгоритмом коме је инваријанта да у сваком кораку петље знамо ове две вредности (максимум сегмента и максимум суфикса).

Пример. Прикажимо рад алгоритма на примеру низа `-2 3 2 -3 -2 4 5 -8 3`. У таблици попуњавамо вредности S_i и M_i.

i	a_{i+1}	S_i	M_i
0		0	0
1	-2	0 = max(0+(-2), 0)	0 = max(0, 0)
2	3	3 = max(0+3, 0)	3 = max(0, 3)
3	2	5 = max(3+2, 0)	5 = max(3, 5)
4	-3	2 = max(5+(-3), 0)	5 = max(5, 2)
5	-3	0 = max(2+(-3), 0)	5 = max(5, 0)
6	-2	0 = max(0+(-2), 0)	5 = max(5, 0)
7	4	4 = max(0+4, 0)	5 = max(5, 4)
8	5	9 = max(4+5, 0)	9 = max(5, 9)
9	-8	1 = max(9+(-8), 0)	9 = max(9, 1)
10	3	4 = max(1+3, 0)	9 = max(9, 4)

Анализа сложености. Пошто елементе учитавамо један по један и не памтимо их истовремено, меморијска сложеност је $O(1)$. Максимални збир сегмента и суфикса инкрементално израчунавамо једним проласком кроз задате елементе и временска сложеност је линеарна тј. $O(n)$.

Напомена. Приметимо да смо у претходном разматрању проширили индуктивну хипотезу претпостављајући да поред тражене вредности тј. максимума неког сегмента првих n елемената низа знамо додатно и вредност максималног суфикса првих n елемената низа.

```
int maxSufiks = 0, max = maxSufiks;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  maxSufiks += x;
  if (maxSufiks < 0)
    maxSufiks = 0;
  if (maxSufiks > max)
    max = maxSufiks;
}
cout << max << endl;
```

Види другачија решења овог задатка.

Задатак: Број растућих сегмената

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Индуктивна конструкција

Најлепше решење можемо добити ако употребимо инкрементално проширивање сегмената једним по једним елементом здесна. Укупан број растућих сегмената у низу можемо добити тако што за сваку позицију низа израчунамо број R растућих сегмената који се завршавају на тој позицији и тако добијене бројеве саберемо.

На позицији 0 се не завршава ни један растући сегмент (јер они по дефиницији морају да буду бар двочлани). Дакле, важи $R_0 = 0$. Ако знамо број растућих сегмената који се завршавају на позицији j, тада веома једноставно можемо израчунати број растућих сегмената који се завршавају на позицији $j + 1$.

- Ако је $a_{j+1} > a_j$, онда се сваки растући сегмент који се завршава на позицији j може проширити елементом a_{j+1} и тако добити нови растући сегмент. Додатно, растући сегмент је и $[a_j, a_{j+1}]$, тако да је укупан број сегмената који се завршавају на позицији $j + 1$ за један већи од укупног броја растућих сегмената који се завршавају на позицији j и важи $R_{j+1} = R_j + 1$.
- Ако је $a_{j+1} \leq a_j$ онда се на позицији j не завршава ни један растући сегмент тј. важи $R_{j+1} = 0$.

Анализа сложености. Број растућих сегмената за сваки десни крај и њихов укупан број одређујемо једним проласком кроз низ, у сложености $O(n)$. Пошто памтимо само да узастопна члана низа, меморијска сложеност је $O(1)$.

```
int n;
cin >> n;
int prethodni;
cin >> prethodni;
// ukupan broj rastucih serija
long long ukupanBrojRastucih = 0;
// broj rastucih koji se zavrsavaju na tekucoj poziciji
long long brojRastucih = 0;
for(int i = 1; i < n; i++) {
  int tekuci;
  cin >> tekuci;
  if (tekuci > prethodni) {
    // tekuci element produzava sve rastuce segmente koji su se zavrsili na
```

```
    // prethodnoj poziciji i dodaje jos jedan nov dvoclan rastuci segment
    brojRastucih++;
    // dodajemo broj rastucih koji se zavrsavaju na poziciji i na
    // ukupan broj rastucih segmenata
    ukupanBrojRastucih += brojRastucih;
  } else {
    // na tekucoj poziciji se ne zavrsava ni jedna rastuca serija
    brojRastucih = 0;
  }
  prethodni = tekuci;
}
cout << ukupanBrojRastucih << endl;
```

Задатак: Фактори неравнотеже бинарног дрвета

Дефинишимо висину празног дрвета као 0, а непразног дрвета као број чворова од корена тог дрвета до њему најудаљенијег листа (рачунајући и корен и тај лист). Дефинишимо фактор равнотеже чвора бинарног дрвета као апсолутну разлику између висина његовог левог и десног поддрвета. Напиши програм који одређује највећи фактор равнотеже неког чвора учитаног бинарног дрвета.

Улаз: Празно бинарно дрво се задаје карактером `-`, а непразно у облику `[k <levo> <desno>]`, где је `k` цео број (вредност у корену), а `<levo>` и `<desno>` су лево и десно поддрво записани у истом формату.

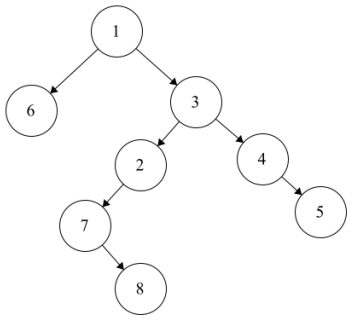
Излаз: На стандардни излаз исписати вредност највећег фактора равнотеже.

Пример 1

Улаз	*Излаз*
`[1 [6 - -] [3 [2 [7 - [8 - -]] -] [4 - [5 - -]]]]`	`3`

Објашњење

Учитано дрво је приказано на слици.



Слика 3.1: Бинарно дрво

Највећи фактор равнотеже је у корену – лево подрво је висине 1, а десно висине 4, па је фактор једнак 3.

Пример 2

Улаз

```
[1 [2 [3 [4 [5 [6 [7 [8 [9 [10 - -] -] -] -] -] -] -] -] -] -]
```

Излаз

```
9
```

Решење

Директна индуктивно-рекурзивна конструкција у овом примеру не даје резултате јер фактор равнотеже чвора не зависи од фактора равнотеже левог и десног поддрвета (тј. њихових корених чворова), већ од висине

тих дрвета. Са друге стране, висину дрвета веома једноставно можемо израчунати на основу индуктивно-рекурзивне конструције (висина празног дрвета је 0, а висина непразног дрвета је за један већа од максимума висина левог и десног поддрвета).

Са функцијом за израчунавање висине на располагању лако можемо дефинисати рекурзивну функцију која израчунава највећи фактор равнотеже. Највећи фактор равнотеже празног дрвета је 0, а највећи фактор равнотеже непразног дрвета је максимум највећег фактора равнотеже левог подрвета, највећег фактора равнотеже десног поддрвета и фактора равнотеже корена, који израчунавамо као апсолутну разлику висина левог и десног поддрвета.

Анализа сложености. Ако претпоставимо да је дрво око сваког свог чвора уравнотежено тј. да је у сваком поддрвету пола чворова лево, а пола чворова десно од корена, функција која израчунава висину дрвета са n чворова задовољава једначину $T(n) = 2 \cdot T(n/2) + O(1)$, чије је решење на основу мастер теореме $O(n)$. Сложеност је таква и без те претпоставке (ако је дрво дегенерисано у листу, сложеност задовољава једначину $T(n) = T(n-1) + O(1)$ чије је решење такође $O(n)$). И интуитивно је јасно да функција у сваком позиву мора да обиђе и лево и десно подстабло да би одредила висину, тако да функција обилази све чворове (једном) и стога је линеарне сложености у односу на број чворова стабла. У општем случају време извршавања задовољава једначину $T(n) = T(k) + T(n-1-k) + O(1)$ тј. $T(n) = T(k) + T(n-1-k) + C$, па се индукцијом може показати да важи $T(n) \leq Cn$. Заиста, на основу индуктивне хипотезе важи $T(n) = T(k) + T(n-1-k) + C \leq Ck + C(n-1-k) + C = Cn$.

Под претпоставком да је дрво око сваког чвора уравнотежено, сложеност функције која проналази највећи фактор равнотеже задовољава једначину $T(n) = 2 \cdot T(n/2) + O(n)$ и њена је сложеност, знамо на основу мастер теореме, $O(n \log n)$. Ако је дрво издегенерисано у листу (што се може десити у најгорем случају), добија се да време извршавања задовољава једначину $T(n) = T(n-1) + O(n)$ чије је решење $O(n^2)$. Рецимо да кључни проблем настаје услед тога што се висина делова дрвета израчунава изнова и изнова.

Нагласимо и да је сложеност учитавања и креирања дрвета, као и ослобађања линеарна и износи $O(n)$, али време може бити значајно услед великих константних фактора насталих услед коришћења релативно скупих операција алокације и деалокације меморије (додуше, то је могуће оптимизовати).

```
int visina(cvor* koren) {
  // visina praznog stabla je nula
  if (koren == nullptr)
    return 0;
  // visina untrašnjeg čvora
  return max(visina(koren->levo), visina(koren->desno)) + 1;
}

// izracunava faktore ravnoteze svih cvorova
int maxFaktorRavnoteze(cvor* koren) {
  if (koren == nullptr)
    return 0;

  int maxFaktorLevo = maxFaktorRavnoteze(koren->levo);
  int maxFaktorDesno = maxFaktorRavnoteze(koren->desno);
  int faktorKoren = abs(visina(koren->levo) - visina(koren->desno));

  return max({maxFaktorLevo, maxFaktorDesno, faktorKoren});
}
```

Алгоритам се може побољшати ако се висина и фактор равнотеже рачунају истовремено, тј. ако појачамо индуктивну хипотезу и претпоставимо да рекурзивним позивима можемо да израчунамо и факторе равнотеже и висину поддрвета. Базу чини случај празног стабла чија је висина нула и које не садржи чворове чији фактор равнотеже треба израчунати. Ако претпоставимо да умемо да израчунамо факторе равнотеже свих чворова поддрвета, као и њихове висине, тада фактор равнотеже корена можемо једноставно израчунати као апсолутну вредност разлике тих висина, а висину целог дрвета као број за један већи од максимума висине левог и десног поддрвета.

Анализа сложености. Под претпоставком да је дрво балансирано, број корака у извршавању овог алгоритма задовољава једначину $T(n) = 2 \cdot T(n/2) + O(1)$, чије је решење $O(n)$. Чак и када дрво није балансирано, сложеност је линеарна (јер је решење једначине $T(n) = T(n-1) + O(1)$ такође $O(n)$). Заиста, какав год да је

облик дрвета, сваки чвор се посећује само једном. Време извршавања задовољава једначину $T(n) = T(k) + T(n-1-k) + O(1)$ тј. $T(n) = T(k) + T(n-1-k) + C$, па се индукцијом може показати да важи $T(n) \leq Cn$. Заиста, на основу индуктивне хипотезе важи $T(n) = T(k) + T(n-1-k) + C \leq Ck + C(n-1-k) + C = Cn$.

```
cvor* ucitaj() {
  char c;
  cin >> c;
  if (c == '-')
    return nullptr;
  else {
    // ucitali smo [
    int vrednost;
    cin >> vrednost >> ws; // ucitavamo vrednost u korenu i preskacemo beline
    cvor* levo = ucitaj(); // rekurzivno ucitvamo levo poddrvo
    cin >> ws; // preskacemo beline
    cvor* desno = ucitaj(); // rekurzivno ucitvamo desno poddrvo
    cin >> ws; // preskacemo beline
    // ucitavamo ]
    cin >> c;
    // kreiramo koreni cvor i vracamo celo ucitano drvo
    return napravi_cvor(vrednost, levo, desno);
  }
}
// izracunava faktore ravnoteze svih cvorova
void visinaIMaxFaktorRavnoteze(cvor* koren, int& visina, int& maxFaktor) {
  if (koren == nullptr) {
    visina = 0;
    maxFaktor = 0;
    return;
  }

  int visinaLevo, maxFaktorLevo;
  visinaIMaxFaktorRavnoteze(koren->levo, visinaLevo, maxFaktorLevo);
  int visinaDesno, maxFaktorDesno;
  visinaIMaxFaktorRavnoteze(koren->desno, visinaDesno, maxFaktorDesno);

  int faktorKoren = abs(visinaLevo - visinaDesno);

  maxFaktor = max({maxFaktorLevo, maxFaktorDesno, faktorKoren});
  visina = max(visinaLevo, visinaDesno) + 1;
}

int maxFaktorRavnoteze(cvor* koren) {
  int visina, maxFaktor;
  visinaIMaxFaktorRavnoteze(koren, visina, maxFaktor);
  return maxFaktor;
}
```

Задатак: Дијаметар бинарног дрвета

Дефинишимо дијаметар бинарног дрвета као број чворова на путањи између два најудаљенија чвора (то ће сигурно бити листови), рачунајући и те чворове. Напиши програм који учитава бинарно дрво и израчунава његов дијаметар.

Улаз: Празно бинарно дрво се задаје карактером `-`, а непразно у облику `[k <levo> <desno>]`, где је `k` цео број (вредност у корену), а `<levo>` и `<desno>` су лево и десно поддрво записани у истом формату.

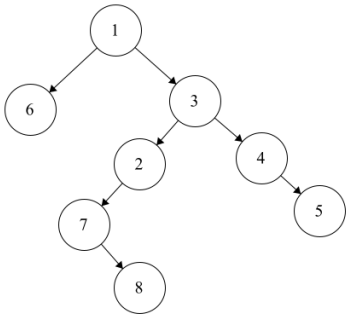
Излаз: На стандардни излаз исписати тражени дијаметар учитаног дрвета.

Пример

Улаз	*Излаз*
`[1 [6 - -] [3 [2 [7 - [8 - -]] -] [4 - [5 - -]]]]`	`6`

Објашњење

Учитано дрво је приказано на слици.



Слика 3.2: Бинарно дрво

Највеће растојање је 6 и оно се достиже између чворова 8 и 5.

Решење

Два најудаљенија чвора (листа) могу да буду оба у левом поддрвету, оба у десном подрвету или један може да буде у левом, а један у десном поддрвету. У прва два случаја пут између њих не садржи корен, а у трећем случају пут између њих садржи корен. У прва два случаја дијаметар дрвета једнак је дијаметру левог односно десног дрвета и може бити израчунат рекурзивно. У трећем случају знамо да најдужи пут садржи корен дрвета, као и путеве од корена левог дрвета до неког његовог листа и од корена десног дрвета до неког његовог листа, при чему ти путеви треба да буду што дужи. Међутим, најдужи пут од корена до листа у дрвету је управо висина тог дрвета, па је најдужи пут који пролази кроз корен дрвета за један већи од збира висина левог и десног поддрвета.

Висину дрвета лако можемо израчунати посебном рекурзивном функцијом (висина празног дрвета је 0, а висина непразног дрвета је за један већа од максимума висина левог и десног поддрвета).

Са функцијом за израчунавање висине на располагању, лако је дефинисати рекурзивну функцију која израчунава дијаметар. Дијаметар празног дрвета је 0, а у супротном је максимум дијаметра левог подрвета, дијаметра десног поддрвета и збира висина тих поддрвета увећаног за 1.

Анализа сложености. Ако претпоставимо да је дрво око сваког свог чвора уравнотежено тј. да је у сваком поддрвету пола чворова лево, а пола чворова десно од корена, функција која израчунава висину дрвета са n чворова задовољава једначину $T(n) = 2 \cdot T(n/2) + O(1)$, чије је решење на основу мастер теореме $O(n)$. Сложеност је таква и без те претпоставке (ако је дрво дегенерисано у листу, сложеност задовољава једначину $T(n) = T(n-1) + O(1)$ чије је решење такође $O(n)$). И интуитивно је јасно да функција у сваком позиву мора да обиђе и лево и десно подстабло да би одредила висину, тако да функција обилази све чворове (једном) и стога је линеарне сложености у односу на број чворова стабла.

Под претпоставком да је дрво око сваког чвора уравнотежено, сложеност функције која проналази дијаметар задовољава једначину $T(n) = 2 \cdot T(n/2) + O(n)$ и њена је сложеност, знамо на основу мастер теореме, $O(n \log n)$. Ако је дрво издегенерисано у листу (што се може десити у најгорем случају), добија се да време извршавања задовољава једначину $T(n) = T(n-1) + O(n)$ чије је решење $O(n^2)$. Рецимо да кључни проблем настаје услед тога што се висина делова дрвета израчунава изнова и изнова.

```
int visina(cvor* koren) {
  if (koren == nullptr)
    return 0;
  return 1 + max(visina(koren->levo), visina(koren->desno));
}

int dijametar(cvor* koren) {
  if (koren == nullptr)
```

```
    return 0;
  int dijametar_levo = dijametar(koren->levo);
  int dijametar_desno = dijametar(koren->desno);
  return max({dijametar_levo, dijametar_desno, 1 + visina(koren->levo) + visina(koren->desno)});
}
```

Ефикасније решење се добија ако се уместо засебних функција за израчунавање висине и дијаметра, индуктивна хипотеза ојача и дефинише се функција која истовремено израчунава висину и дијаметар.

Анализа сложености. Под претпоставком да је дрво балансирано, број корака у извршавању овог алгоритма задовољава једначину $T(n) = 2 \cdot T(n/2) + O(1)$, чије је решење $O(n)$. Чак и када дрво није балансирано, сложеност је линеарна (јер је решење једначине $T(n) = T(n-1)+O(1)$ такође $O(n)$). Заиста, какав год да је облик дрвета, сваки чвор се посећује само једном. Време извршавања задовољава једначину $T(n) = T(k) + T(n-1-k)+O(1)$ тј. $T(n) = T(k)+T(n-1-k)+C$, па се индукцијом може показати да важи $T(n) \leq Cn$. Заиста, на основу индуктивне хипотезе важи $T(n) = T(k)+T(n-1-k)+C \leq Ck+C(n-1-k)+C = Cn$.

```
void dijametar_i_visina(cvor* koren, int& dijametar, int& visina) {
  if (koren == nullptr) {
    dijametar = 0;
    visina = 0;
  } else {
    int dijametar_levo, visina_levo;
    dijametar_i_visina(koren->levo, dijametar_levo, visina_levo);
    int dijametar_desno, visina_desno;
    dijametar_i_visina(koren->desno, dijametar_desno, visina_desno);
    dijametar = max({dijametar_levo, dijametar_desno, 1 + visina_levo + visina_desno});
    visina = 1 + max(visina_levo, visina_desno);
  }
}

int dijametar(cvor* koren) {
  int d, v;
  dijametar_i_visina(koren, d, v);
  return d;
}
```

Глава 4

Структуре података

Ефикасна организација података у меморији, тако да се подаци могу лако претраживати и мењати представља важан предуслов писања ефикасних програма. Поред примитивних типова података (попут, на пример, статички алоцираних низова или кориснички дефинисаних структурних типова), савремени програмски језици нуде велики број **библиотечких структура података** које олакшавају процес програмирања. У наставку ћемо приказати структуре података, које се толико често употребљавају да су директно подржане у библиотекама већине програмских језика.

Структуре података можемо посматрати преко њихових операција, тј. преко функционалности које пружају својим корисницима. У том светлу за коришћење неке библиотечке структуре података потребно је знати њен апстрактни интерфејс (функције тј. методе које се позивају да би се извршиле операције над подацима складиштеним у склопу структуре), док њена имплементација може остати у потпуности сакривена. При том, веома је важно имати осећај и о сложености сваке операције.

Језик C++ пружа подршку за велики број структура података, било примитивних, подржаних кроз сам језик, било подржаних кроз стандардну библиотеку (STL).

По начину како су имплементиране, библиотечке структуре података (каже се и **контејнери**, енгл. containers) се могу груписати на следећи начин.

- У групу секвенцијалних контејнера спадају `array`, `string`, `vector`, `list`, `forward_list` и `deque`.
- У групу адаптора контејнера спадају `stack`, `queue` и `priority_queue`.
- У групу уређених асоцијативних контејнера спадају `set`, `multiset`, `map` и `multimap`.
- У групу неуређених асоцијативних контејнера спадају `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`.

Тесно повезани са контејнерима су **итератори** који су уопштење показивача и представљају објекте који показују на одређено место (на одређен елемент) унутар контејнера или евентуално на део меморије непосредно иза елемената контејнера (такав је специјални итератор који се добија методом `end`). Итератори се могу *дереференцирати* применом оператора `*`, што значи да на основу итератора можемо очитати или изменити елемент контејнера на који тај итератор показује. Уз то, итератори се могу померати (у зависности од контејнера, само у једном или у оба смера) и поредити (поређење једнакости и различитости итератора операторима `==` и `!=` је увек подржано, а понекад је могуће упоредити да ли је неки оператор испред или другог, операторима `<`, `<=`, `>` и `>=`) . Многе библиотечке функције које се примењују над елементима контејнера захтевају пар итератора који ограничава део контејнера на који ће се функција примењивати (најчешће се наводи пар итератора добијених методама `begin` који `end` ограничавају целокупан контејнер тј. све његове елементе).

Секвенцијални контејнери се користе за складиштење серија елемената. Елементи се увек складиште секвенцијално, један иза другог, па сви секвенцијални контејнери представљају одређена уопштења примитивних, статички алоцираних низова. Карактерише их могућност обиласка свих елемената редом (понекад у оба правца), као и могућност индексног приступа елементу на основу његове позиције (додуше, та операција не мора увек да буде константне сложености). Са изузетком `array`, који представља само танак омотач око примитивног статичког низа и чија димензија мора бити позната приликом превођења програма, сви остали секвенци-

јални контејнери су динамички алоцирани и допуштено је и уметање и брисање елемената (и на почетак и на средину и на крај), при чему сложеност тих операција зависи од одабира секвенцијалног контејнера (стога их треба користити веома обазриво). Иако имају доста сличан интерфејс, секвенцијални контејнери су имплементирани на различите начине, па им се разликује сложеност извођења одређених операција. Сваки од њих има одређене предности и мане и сценарије у којима је пожељно и сценарије у којима га није пожељно користити.

Адаптори контејнера само представљају слој изнад неког од постојећих секвенцијалних контејнера и пружају апстрактни интерфејс изнад имплементације секвенцијалног контејнера, имплементирајући функције тог интерфејса коришћењем секвенцијалног контејнера за складиштење података. Адаптори имају свој подразумевани секвенцијални контејнер, који се може променити приликом декларисања променљивих.

- `stack` имплементира функције стека (LIFO) где се елементи могу додавати и уклањати са једног краја.
- `queue` имплементира функције реда (FIFO) где се елементи додају на један, а узимају са другог краја.
- `priority_queue` имплементира функције реда са приоритетом где се елементи додају у произвољном редоследу, а ваде у нерастућем редоследу вредности (из реда се увек може прочитати и извадити само највећи елемент реда).

Асоцијативни контејнери подразумевају да се уметање, претрага и брисање елемената врши на основу њихове вредности, а не на основу позиције на којој се налазе — приступ елементима је на основу кључа, а не на основу индекса тј. позиције. Код секвенцијалних контејнера однос елемената се природно успоставља на основу њиховог положаја унутар контејнера (без обзира на вредности). На пример, у низу је могуће и да се неки мањи елемент налази испред већег и да се неки већи елемент налази испред мањег. Са друге стране положај елемената у асоцијативним контејнерима се успоставља искључиво на основу њихове вредности (на пример, у уређеном скупу мањи елементи увек претходе већим).

Основни асоцијативни контејнери су *скупови* и *мапе* тј. *речници*. Скупови одговарају скуповима у математици и пружају ефикасно додавање и избацивање елемената, као и проверу да неки елемент припада скупу. Мапе тј. речници чувају коначна пресликавања у којима се неким кључевима (елементима неког домена) придружују неке вредности (елементи неког кодомена).

На основу начина како су имплементирани, асоцијативни контејнери се деле на *уређене* и *неуређене*. Уређени асоцијативни контејнери корисницима нуде одређене додатне функционалности (на пример, испис свих елемената у растућем редоследу вредности или проналажење најмањег елемената који је већи од дате вредности), али по цену да су понекад мало спорији од неуређених.

Уређени асоцијативни контејнери су следећи:

- `set<T>` – скуп елемената типа `T`
- `multiset<T>` – мултискуп елемената типа `T` (допуштена су вишеструка појављивања елемената)
- `map<K, V>` — пресликавање кључева типа `K` у вредности типа `V`
- `multimap<K, V>` — пресликавање кључева типа `K` у вредности типа `V` при чему се сваки оригинал може сликати у више различити слика.

Сви они претпостављају да се њихови елементи (тј. кључеви у случају мапа и мултимапа) могу поредити (релацијским оператором `<`). Обично су имплементирани помоћу самобалансирајућих уређених бинарних дрвета и карактерише их логаритамска сложеност основних операција. Укратко, складиштње елемената у дрвету подразумева да су елементи складиштени у чворовима који осим релевантних вредности чувају и показиваче на лева и десна поддрвета. Уређена бинарна дрвета подразумевају да се је вредност у сваком чвору већа или једнака од вредности у свим чворовима левог и строго мања од вредности од свих чворова у десном поддрвету, што значи да се се уметање и претрага остварују веома слично поступку бинарне претраге (поређењем са вредношћу у корену и затим преласком на лево или десно поддрво). Стога сложеност зависи од висине бинарног дрвета, која у случају да је дрво балансирано (да се висина левог и десног поддрвета не разликују превише), логаритамски зависи од укупног броја чворова у дрвету.

Неуређени асоцијативни контејнери су следећи:

- `unordered_set<T>`
- `unordered_multiset<T>`
- `unordered_map<K, V>`
- `unordered_multimap<K, V>`

Обично су имплементирани помоћу *хеш-табела* и карактерише их амортизована константна сложеност основних операција. Неуређени контејнери претпостављају да постоји дефинисана *хеш-функција* `hash` која елементе скупа тј. кључеве мапе слика у целобројне вредности (тзв. хеш-кодове) које одређују позицију унутар низа вредности на којој елемент треба да се нађе, ако је елемент скупа тј. кључ у мапи. Могуће је и да наступе *колизије* тј. да више елемената има исте хеш-кодове тј. да се хеш-функцијом сликају на исту позицију. Постоји неколико начина разрешавања колизија (најједноставнија подразумева да се на тој позицији чува листа свих елемената који су хеш-функцијом придружени тој позицији).

4.1 Скупови и мапе (речници)

4.1.1 Скупови

У програмима често имамо потребе да одржавамо скуп елемената (без дупликата), у који ефикасно можемо да додајемо елементе, из кога ефикасно можемо да избацујемо елементе и за који ефикасно можемо да проверавамо да ли је нека задата вредност елемент скупа. Савремени програмски језици у својим библиотекама пружају структуре података које нуде баш ове операције.

У језику C++ скуп је подржан кроз две класе: `set<T>` и `unordered_set<T>`, где је `T` тип елемената скупа (за њихово коришћење је потребно укључити заглавље `<set>` тј. `<unordered_set>`). Имплементација је различита (прва је заснована на балансираним бинарним дрветима, а друга на хеш-табелама), па су им временске и просторне карактеристике донекле различите.

Скупови подржавају следеће основне операције (за преглед свих операција упућујемо читаоца на документацију):

- `insert` - умеће нови елемент у скуп (ако је елемент већ у скупу, операција нема ефекта). Када се користи `set`, сложеност уметања је $O(\log k)$, где је k број елемената у скупу, а када се користи `unordered_set`, сложеност најгорег случаја је $O(k)$, док је просечна сложеност $O(1)$, при чему је амортизована цена сваког додавања у склопу већег броја узастопних додавања такође $O(1)$. Нагласимо и да константе код сложености $O(1)$ могу бити релативно велике и да уметање не можемо сматрати јако брзом операцијом. Честа операција је додавање n елемената у скуп. Најгора сложеност је ако су сви елементи различити и у случају уређеног скупа износи приближно $\log 1 + \log 2 + \ldots + \log n$, за шта се може показати да је $O(n \log n)$.
- `erase` - уклања дати елемент из скупа (ако елемент не постоји у скупу, скуп се не мења). Сложеност је иста као у случају уметања.
- `find` - проверава да ли скуп садржи дати елемент и враћа итератор на њега или `end` ако је одговор негативан. Тако се провера припадности елемента `e` скупу `s` може извршити са `if (s.find(e) != s.end()) ...` Сложеност најгорег случаја ове операције ако се користи `set` је $O(\log k)$, а ако се користи `unordered_set` сложеност најгорег случаја је $O(k)$, али је просечна сложеност $O(1)$.
- `size` - враћа број елемената скупа.

Могућа је и итерација кроз елементе скупа коришћењем петље облика `for (T element : skup)`, при чему се елементи колекције `set` набрајају у сортираном, а `unordered_set` у прилично насумичном редоследу (редослед је одређен хеш-функцијом која се користи и на њега се, као што само име `unordered` говори, не треба ослањати).

Ако се у скуп стављају елементарни типови података (бројеви, ниске, …), тада се користи подразумевани поредак у случају уређених тј. подразумевана хеш-функција у случају неуређених скупова (на пример, у случају бројева подразумевани поредак је растући). Ово је могуће променити, али излази ван домена овог материјала. Да би се формирао скуп елемената неког типа над којим није дефинисан подразумевани поредак нити хеш-функција (најчешће скуп структура или објеката неке класе), потребно је посебно дефинисати функцију поретка у случају коришћења уређених тј. хеш-функцију у случају коришћења неуређених скупова. Издвојићемо само случај када желимо да направимо скуп у ком су елементи уређени нерастуће (при чему на типу података постоји подразумевани неопадајући поредак). Такав скуп је најједноставније дефинисати коришћењем `set<T, greater<T>>`, где је за употребу `greater` потребно укључити заглавље `<functional>`.

Уређени скупови (колекција `set`) подржавају и методе:

- `lower_bound(x)` - проналази најмањи елемент скупа који је већи или једнак од дате вредности `x` и враћа итератор који указује на њега (или `end`, ако такав елемент не постоји),

- `upper_bound(x)` - проналази најмањи елемент скупа који је строго већи од дате вредности `x` и враћа итератор који указује на њега (или `end`, ако такав елемент не постоји).

Подразумевани поредак елемената у скупу је растући и методе `lower_bound` и `upper_bound` раде у односу на тај поредак (при чему се поредак може променити приликом дефинисања скупа и примене ових функција, навођењем другачије функције поређења).

4.1.1.1 Мултискупови

Скупови, као и у математици, не могу да садрже дупликате. Када се елемент који већ постоји у скупу убацује у скуп методом `insert`, скуп се не мења. Једно уопштење скупова дају *мултискупови* у којима је допуштено понављање елемената. Мултискупови су подржани библиотечком структуром `multiset<T>`, која се користи на потпуно исти начин као и `set<T>` (за њено коришћење је такође довољно укључити заглавље `<set>`).

4.1.2 Мапе (речници)

Програмски језик C++ пружа подршку за креирање мапа (речника, асоцијативних низова) који представљају колекције података у којима се кључевима неког типа придружују вредности неког типа (не обавезно истог). На пример, именима месеци (подацима типа `string`) можемо доделити број дана (податке типа `int`). Речници се представљају објектима типа `map<TipKljuca, TipVrednosti>`, дефинисаном у заглављу `<map>`. На пример,

```
map<string, int> brojDana =
{
    {"januar", 31},
    {"februar", 28},
    {"mart", 31},
    ...
};
```

Приметимо да смо иницијализацију мапе извршили тако што смо навели листу парова облика `{kljuc, vrednost}`. Иницијализацију није неопходно извршити одмах током креирања, већ је вредности могуће додавати (а и читати) коришћењем индексног приступа (помоћу заграда `[]`).

```
map<string, int> brojDana;
brojDana["januar"] = 31;
brojDana["februar"] = 28;
brojDana["mart"] = 31;
...
```

Мапу, дакле, можемо схватити и као низ тј. вектор у коме индекси нису обавезно из неког целобројног интервала облика $[0, n)$, већ могу бити произвољног типа.

Претрагу кључа можемо остварити методом `find` која враћа итератор на пронађени елемент тј. итератор иза краја мапе (који добијамо методом или функцијом `end`), ако елемент не постоји. На пример,

```
string mesec;
cin >> mesec;
auto it = brojDana.find(mesec);
if (it != end(brojDana))
   cout << "Broj dana: " + *it << endl;
else
   cout << "Mesec nije korektno unet" << endl;
```

Све елементе речника могуће је исписати коришћењем петље `for`. На пример,

```
for (auto& p : brojDana)
     cout << p.first << ": " << p.second << endl;
```

Алтернативно, можемо експлицитно користити итераторе

```
for (auto it = brojDana.begin(); it != brojDana.end(); it++)
     cout << it->first << ": " << it->second << endl;
```

Приметимо да смо у првом случају користили референцу да се пар елемената не би копирао у сваком кораку петље. У другом случају се користе итератори и оригиналном пару елемената се приступа дереференцирањем итератора.

У случају уређених мапа, итерација се врши у сортираном редоследу кључева.

Постоји и облик неуређене мапе (`unordered_map` из истоименог заглавља), која може бити у неким ситуацијама мало бржа него уређена (сортирана) мапа, но то је обично занемариво. Кључеви сортиране мапе могу бити само они типови који се могу поредити релацијским операторима, док кључеви неуређене мапе могу бити само они типови који се могу лако претворити у број (тзв. хеш-вредност). Ниске, које ћемо најчешће користити као кључеве, задовољавају оба услова.

Задатак: Дупликати

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Скупови

Библиотеке савремених програмских језика обично нуде и колекције података за репрезентовање скупова. Једна могућност је да се користи колекција заснована на балансираном бинарном дрвету. У језику C++ таква је колекција `set`. Додавање елемента у скуп се врши методом `insert` (при том се аутоматски води рачуна да се скуп не мења додавањем елемента који већ постоји), док се број елемената одређује методом `size()`.

Анализа сложености. Са имплементацијом скупа базираном на балансираном бинарном дрвету, убацивање у скуп је обично сложености $O(\log k)$, где је k број елемената у скупу, па је укупна сложеност овог приступа највише $O(n \log n)$.

```
// ucitavamo elemente u skup
set<unsigned> a;
int n;
cin >> n;
for (int i = 0; i < n; i++) {
  unsigned x;
  cin >> x;
  a.insert(x);
}
// ispisujemo broj elemenata skupa
cout << a.size() << endl;
```

Једна могућност је да се користи колекција заснована на хеширању. У језику C++ таква је колекција `unordered_set` и она се користи на исти начин као и `set` (при чему тип елемената мора да буде такав да је за њега расположива хеш-функција).

Анализа сложености. Сложеност најгорег случаја додавања у скуп заснован на хеширању је $O(k)$, где је k број елемената у скупу, али је амортизована цена једног додавања у склопу већег броја додавања једнака $O(1)$. Зато је укупна сложеност алгоритма једнака $O(n)$ (уз релативно велики константни фактор који уметање у овакав скуп носи).

```
// ucitavamo elemente u skup
unordered_set<unsigned> a;
int n;
cin >> n;
for (int i = 0; i < n; i++) {
  unsigned x;
  cin >> x;
  a.insert(x);
}
// ispisujemo broj elemenata skupa
cout << a.size() << endl;
```

Задатак: Сортирање бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Сортирање уз помоћ мултискупа

Сортирање се може извршити коришћењем мултискупа. Користи се алгоритам *сортирања уметањем* (енгл. *insertion sort*), једино што се уместо уметања елемената у низ врши уметање елемената у мултискуп. На крају се испишу сви елементи мултискупа редом (они се чувају у сортираном редоследу). Пошто се мултискуп имплементира коришћењем сортираног бинарног дрвета, овај се алгоритам назива и *сортирањем коришћењем дрвета* (енгл. *tree sort*).

У језику C++ можемо користити библиотечку колекцију `multiset`.

Анализа сложености. Пошто се уметање у мултискуп извршава у сложености $O(\log k)$, где је k број елемената у мултискупу, а испис у времену $O(k)$, сложеност овог поступка сортирања је $O(n \log n)$. Заузеће меморије је $O(n)$, уз, додуше, мало већи константни фактор него када се елементи смештају у низ. Додатно, елементи нису поређани један уз други у меморији, што мало успорава приступ.

```
int n;
cin >> n;
multiset<int> a;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  a.insert(x);
}
for (int x : a)
  cout << x << endl;
```

Види другачија решења овог задатка.

Задатак: Број различитих дужина дужи

Дато је N парова тачака које представљају крајеве дужи у простору. Исписати колико различитих дужина дужи се појављује у задатом скупу дужи.

Улаз: У првом реду улаза налази се природан број N ($N \leq 50000$) који представља број дужи. У следећих N редова следи опис тих N дужи са 6 целих бројева ($-10^9 \leq X_1, Y_1, Z_1, X_2, Y_2, Z_2 \leq 10^9$) одвојених празним местима, који редом представљају крајеве сваке дужи.

Излаз: У једини ред излаза потребно је исписати колико различитих дужина се појављује у задатом скупу дужи.

Пример 1

Улаз	*Излаз*
7	3
0 0 0 0 0 1	
0 0 0 0 1 0	
0 0 0 1 0 0	
0 0 0 1 0 1	
0 0 0 1 1 0	
0 0 0 0 1 1	
0 0 0 1 1 1	

Решење

Скуп дужина дужи

Задатак можемо решити и помоћу библиотечких структура података. У језику C++ можемо употребити једну од две имплементације скупа (`set` или `unordered_set`). Квадрате дужина дужи убацујемо у скуп и на крају

очитавамо број елемената тог скупа. Јако је важно нагласити да проналажење самих дужина коренованьем, у облику реалних бројева, не би дало добро решење. Наиме, услед грешака у запису тј. заокруживању реалних бројева, дешава се да се једнаке вредности могу протумачити као различите, а различите вредности као једнаке. Стога реалне бројеве никада не би требало чувати као елементе скупа нити као кључеве у мапи.

Анализа сложености. Ако претпоставимо да је операција убацивања елемента у скуп који садржи k елемената сложености $O(k)$, укупна сложеност овог алгоритма је $O(n \log n)$.

```
// skup svih duzina duzi
unordered_set<long long> duzine;

// ucitavamo koordinate temena duzi i ubacujemo njihove duzine u skup
int n;
cin >> n;
for (int i = 0; i < n; i++) {
  int x1, y1, z1, x2, y2, z2;
  cin >> x1 >> y1 >> z1 >> x2 >> y2 >> z2;
  duzine.insert((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2));
}

// ispisujemo broj elemenata skupa
cout << duzine.size() << endl;
```

Задатак: Својство 132

Низ a дужине n задовољава 132-својство ако постоји тројка индекса $0 \leq i < j < k < n$, тако да је $a_i < a_k < a_j$. Напиши програм који испитује да ли низ задовољава 132-својство.

Улаз: Са стандардног улаза се учитава број елемената низа n ($3 \leq n \leq 10^5$), а затим и n елемената низа (раздвојених размаком).

Излаз: На стандардни излаз исписати `da` или `ne` у зависности од тога да ли низ задовољава 132-својство или не.

Пример 1

Улаз	*Излаз*
`4` `3 1 4 2`	`da`

Објашњење

На пример, елементи $a_1 = 1, a_2 = 4, a_3 = 2$ задовољавају 132-својство.

Пример 2

Улаз

```
7
9 11 8 9 10 7 9
```

Излаз

```
da
```

Објашњење

На пример, елементи $a_0 = 9, a_1 = 11, a_4 = 10$ задовољавају 132-својство.

Пример 3

Улаз

```
4
1 2 3 4
```

Излаз

```
ne
```

Решење

Највећи елемент десно од a_j строго мањи од a_j – скуп

За сваки интервал (a_i, a_j), где је a_i минимум префикса $a_0, \ldots, a_j$ треба проверити да ли у десном делу низа, иза позиције j, постоји елемент a_k који припада том интервалу. Постоје два приступа да се то ефикасно урадимо:

- Потребно и довољно је да десно од a_j постоји елемент који је строго мањи од a_j и да за највећи елемент десно од a_j који је строго мањи од a_j важи да је строго већи од a_i.
- Потребно је и довољно да десно од a_j постоји елемент који је строго већи од a_i и да за најмањи елемент десно од a_j који је строго већи од a_i важи да је строго мањи од a_j.

Илуструјмо решење засновано на првом приступу. Тврђење важи, јер ако је највећи елемент десно од a_j који је строго мањи од a_j мањи или једнак од a_i, онда су сви елементи десно од a_j строго мањи од a_j мањи или једнаки a_i и ниједан не припада интервалу (a_i, a_j).

Једно ефикасно решење можемо добити коришћењем структуре података у којој ћемо чувати све елементе десно од текућег елемента a_j, а која нам омогућава да међу њима ефикасно пронађемо највећи елемент који је строго мањи од дате вредности a_j. У језику C++ можемо користити библиотечку колекцију скуп (`set`) уређен опадајући и њену методу `upper_bound` (ефикасна метода `upper_bound`, која ради у логаритамској сложености, постоји код уређених скупова `set`, али не и код неуређених скупова `unordered_set`).

Елементе a_j можемо обилазити здесна налево, за сваког проверавати да ли је средишњи елемент неке 132-тројке и ако није, додавати га у скуп (јер ће он бити десно од свих елемената које ћемо у наредним итерацијама обрађивати). Обиласком слева надесно требало би елементе избацивати из скупа, што је мало неелегантније (мада је временска сложеност иста).

Обилазак низа здесна налево мало компликује проналажење минимума префикса који се израчунавају инкрементално, слева надесно, међутим, њих можемо одредити у посебном пролазу на самом почетку и сместити у помоћни низ.

Анализа сложености. Додавање елемената у скуп који садржи m елемената, као и одређивање највећег елемента већег од дате вредности су сложености $O(\log m)$. Пошто се и додавање и претрага врше n пута, сложеност овог приступа је $O(n \log n)$ (израчунавање максимума префикса које се врши у фази претпроцесирања је сложености $O(n)$).

```
bool svojstvo132(const vector<int>& a) {
  int n = a.size();
  // niz minimuma prefiksa - na poziciji i nalazi se minimum prefiksa
  // niza na pozicijama [0, i]
  vector<int> minP(n);
  minP[0] = a[0];
  for (int i = 1; i < n; i++)
    minP[i] = min(minP[i-1], a[i]);

  // elementi desno od aj
  set<int, greater<int>> elementi_desno;
  elementi_desno.insert(a[n-1]);

  // za svaki element aj proveravamo da li moze biti sredisnji u nekoj
  // 132-trojci
  for (int j = n-2; j > 0; j--) {
    // analiziramo interval (ai, aj) i proveravamo da li postoji ak
    // desno od j koji mu pripada
    int ai = minP[j], aj = a[j];
    // interval je prazan
    if (ai == aj)
      continue;
    // trazimo najveci element desno od aj koji je strogo manji od aj
    auto najveci_desno_manji_od_aj = elementi_desno.upper_bound(aj);
    // ako on postoji i strogo je veci od ai, pronadjena je 132-trojka
```

```
    if (najveci_desno_manji_od_aj != elementi_desno.end() &&
        *najveci_desno_manji_od_aj > ai)
      return true;

    // najveci element desno od aj koji je manji od aj je manji ili
    // jednak ai, pa aj ne moze biti sredisnji element trojke

    // aj ce biti desno od elemenata u narednim iteracijama
    elementi_desno.insert(aj);
  }
  // nije pronadjena 132-trojka
  return false;
}
```

Задатак: Фреквенција знака

Написати програм који чита једну реч текста коју чине само велика слова енглесне абецеде и исписује слово који се најчешће појављује и колико пута се појављује. Ако се више слова најчешће појављује, исписује се слово које се пре појавило у речи.

Улаз: У једној линији стандардног улаза налази се једна реч текста са не више од 20 слова.

Излаз: У првој линији стандарног излаза приказати слово које се најчешће појављује, а у другој линији стандардног улаза исписати и колико пута се појављује.

Пример 1

Улаз	*Излаз*
POPOKATEPETL	P 3

Пример 2

Улаз	*Излаз*
BACACB	B 2

Решење

Кључни део задатака је да се за свако велико слово енглеског алфабета изброји колико се пута појављује у датој речи. Потребно је, дакле, увести 26 различитих бројача - по један за свако велико слово енглеског алфабета. Јасно је да увођење 26 различитих променљивих не долази у обзир. Потребна је структура података која пресликава дати карактер у број његових појављивања.

Најбоља таква структура у овом задатку је низ бројача у којем се на позицији нула чува број појављивања слова `A`, на позицији један слова `B` итд., све до позиције 25 на којој се налази број појављивања слова `Z`. Централно питање је како на основу карактера одредити позицију његовог бројача у низу. За то се користи чињеница да су карактерима придружени нумерички кодови (ASCII у језику C++) и то на основу редоследа карактера у енглеском алфабету. Редни број карактера је зато могуће одредити одузимањем кода карактера `A` од кода тог карактера (на пример, кôд карактера `D` је 68, док је код карактера `A` 65, одузимањем се добија 3, што значи да се бројач карактера `D` налази на месту број 3 у низу).

Задатак можемо решавати тако што ћемо два пута проћи кроз низ унетих слова тј. реч. У првом пролазу бројач појављивања сваког великог слова на које наиђемо увећавамо за 1. Након тога (или паралелно са тим) одређујемо и максималан број појављивања неког слова. За то користимо уобичајене начине за одређивање максимума низа.

У другом пролазу кроз низ унетих слова тј. реч тражимо прво слово речи чији је број појава једнак већ нађеном највећем броју појава неког слова. Када га нађемо исписујемо га и прекидамо петљу (на пример, наредбом `break`). Приметимо да овде заправо вршимо једноставну линеарну претрагу.

```
string rec;
cin >> rec;

int brojPojavljivanja[26] = {0};
for (char c : rec)
  brojPojavljivanja[c - 'A']++;

int maxPojavljivanja = 0;
for (int i = 0; i < 26; i++)
```

```
  if (brojPojavljivanja[i] > maxPojavljivanja)
    maxPojavljivanja = brojPojavljivanja[i];

for (char c : rec)
  if (brojPojavljivanja[c - 'A'] == maxPojavljivanja) {
    cout << c << endl
         << maxPojavljivanja << endl;
    break;
  }
```

Пресликавање карактера у њихов број појављивања могуће је остварити и библиотечким структурама података које представљају тзв. асоцијативне низове (мапе, речнике).

У језику C++ бисмо могли употребити `map<char, int>` или `unordered_map<char, int>`.

У мапи `m` Изразом `m[c]` се приступа броју појављивања карактера `c`. У језику C++ се изразом `m[c]++` може увећати број појављивања било да карактер `c` постоји у мапи или не (ако не постоји, овим се број појављивања поставља на 1).

Пошто се приликом коришћењем мапа тј. речника бројеви појављивања придружују само оним карактерима који се заиста јављају у речи (а не унапред свим карактерима), мало се штеди меморија. Ипак, ове структуре података су примереније за ситуације у којима није тако једноставно на основу кључа добити нумеричку вредност позиције и када је скуп допустивих кључева шири.

```
// rec koja se analizira
string rec;
cin >> rec;

// broj pojavljivanja svakog slova od A do Z
map<char, int> brojPojavljivanja;
// uvecavamo broj pojavljivanja svakog slova iz reci
for (char c : rec)
  brojPojavljivanja[c]++;

// odredjujemo najveci broj pojavljivanja slova
int maxPojavljivanja = 0;
for (const auto& p : brojPojavljivanja)
  if (p.second > maxPojavljivanja)
    maxPojavljivanja = p.second;

// ispisujemo slovo koje se prvo pojavljuje u reci sa tim brojem
// pojavljivanja
for (char c : rec)
  if (brojPojavljivanja[c] == maxPojavljivanja) {
    cout << c << endl
         << maxPojavljivanja << endl;
    break;
  }
```

Задатак: Фреквенције речи

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Бројање речи

Централно место у задатку је избројати појављивања сваке речи која се јавила на улазу. Овај задатак је донекле сличан задатку Фреквенција знака, једино што се уместо појединачних карактера броје речи. За разлику од ситуације када смо на основу ASCII кода карактера могли одредити његову позицију и тако број

појављивања сваког карактера чувати у низу бројача, овај пут то није једноставно и у ефикасном решењу је потребно користити напредније структуре података.

У језику C++ пресликавање речи у њен број појављивања можемо реализовати или типом `map<string, int>` (који је заснован на претраживачком стаблу) или `unordered_map<string, int>` (који је заснован на хеш-таблици).

Учитавамо реч по реч док не дођемо до краја улаза. За сваку реч увећавамо њен број појављивања у мапи тј. у речнику.

Изразом `brojPojavljivanja[rec]` се приступа броју појављивања речи. У језику C++ се изразом `brojPojavljivanja[rec]++` може увећати број појављивања било да реч постоји у мапи или не (ако не постоји, овим се број појављивања поставља на 1).

Једном када је познат број појављивања сваке речи, тада је могуће одредити тражену реч са најмањим бројем појављивања. Пошто се у случају више речи са истим бројем појављивања тражи она која је лексикографски најмања, користимо лексикографско поређење (хијерархијско поређење на основу више критеријума).

```
string s;
map<string, int> brojPojavljivanja;
while (cin >> s)
  brojPojavljivanja[s]++;

int max = 0; string maxRec;
for (auto it = brojPojavljivanja.begin();
     it != brojPojavljivanja.end(); it++)
  if (it->second > max ||
      (it->second == max && it->first < maxRec)) {
    maxRec = it->first;
    max = it->second;
  }

cout << maxRec << " " << max << endl;
```

Задатак: Сегмент датог збира у низу целих бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Ефикасна претрага збирова префикса

Изражавање збира сегмента у облику разлике збира два префикса нам даје могућност да стигнемо до ефикаснијег решења. Та техника је објашњена, на пример, у задатку Највећи збир префикса. Проблем можемо формулисати и овако. За сваки збир z_{j+1} префикса $[0, j+1)$ потребно је да пронађемо да ли постоји збир z_i префикса $[0, i)$ за неко $i < j$ таква да је $z_{j+1} - z_i = z$, где је z тражени збир, тј. да се провери да ли се међу збировима претходних префикса налази вредност $z_i = z_{j+1} - z$. Ако се та претрага врши линеарно, долазимо до имплементације, која испитује сваки пар елемената $i < j$. Пошто међу елементима низа може бити и негативних, збирови префикса нису сортирани и не можемо применити ни бинарну претрагу. Остаје нам, међутим, могућност да у некој структури података која омогућава ефикасно претраживање чувамо све збирове префикса за индексе $i < j$. Ако алгоритам организујемо тако да j увећавамо од 0 до $n-1$, тада се на крају сваког корака у ту структуру може додати и збир текућег сегмента (z_{j+1}). Структура треба да реализује претрагу по кључу, тако да је најбоље употребити асоцијативни низ (мапу тј. речник). У језику C++ то може бити `map` или `unordered_map`.

Пошто се у задатку тражи само одређивање броја сегмената са датим збиром, кључеви могу бити збирови префикса, а вредност придружена сваком кључу може бити број префикса са тим збиром. Да се тражила само провера да ли постоји сегмент са датим збиром, могли смо уместо асоцијативног низа (мапе, речника) чувати само скуп раније виђених вредности збирова префикса, а да су се експлицитно тражили сви префикси, онда бисмо сваки кључ пресликавали у низ вредности i таквих да је z_i једнако том кључу.

Анализа сложености. Ако рачунамо да ће претрага бити реализована у $O(\log n)$ (што је најчешће случај ако се користе структуре података засноване на бинарним стаблима), тада ће укупна сложеност ове имплементације бити $O(n \log n)$. Напоменимо да смо добитак на ефикасности платили додатном меморијом коју смо ангажовали, међутим, у овом сценарију није потребно памтити учитан низ, тако да меморијска сложеност неће бити значајно повећана.

```
// ucitavamo trazeni zbir
int trazeniZbir;
cin >> trazeniZbir;

// zbir prefiksa
int zbirPrefiksa = 0;

// broj segmenata sa trazenim zbirom
int broj = 0;

// broj pojavljivanja svakog vidjenog zbira prefiksa
map<int, int> zbiroviPrefiksa;
// zbir pocetnog praznog prefiksa je 0 i on se za sada pojavio
// jednom
zbiroviPrefiksa[0] = 1;

// ucitavamo elemente niza niz
int n;
cin >> n;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  // prosirujemo prefiks tekucim elementom
  zbirPrefiksa += x;

  // trazimo broj pojavljivanja vrednosti zbirPrefiksa - trazeniZbir
  // i azuriramo broj pronadjenih segmenata
  auto it = zbiroviPrefiksa.find(zbirPrefiksa - trazeniZbir);
  if (it != zbiroviPrefiksa.end())
    broj += it->second;

  // povecavamo broj pojavljivanja trenutnog zbira
  zbiroviPrefiksa[zbirPrefiksa]++;
}

cout << broj << endl;
```

Задатак: Број сегмената са различитим елементима

Дат је низ природних бројева дужине n. Написати програм којим се одређује колико има сегмената у датом низу чији су сви елементи различити. Сегмент низа чине узастопни елементи низа (њих бар 2).

Улаз: Прва линија стандардног улаза садржи природан број n ($2 \leq n \leq 50000$), број елемената низа. У свакој од n наредних линија стандардног улаза, налази по један члан низа.

Излаз: На стандардном излазу приказати у једној линији број сегмената датог низа чији су сви елементи различити.

Пример

Улаз	*Излаз*
5 1 2 2 3 6	4

Решење

Максимални сегменти за фиксирани крај

Једно прилично елегантно решење се заснива на томе да за сваку позицију анализирамо све сегменте којима је крај управо на тој позицији, а којима су сви елементи различити. Приметимо да ако неки сегмент има то својство, онда и сви сегменти иза њега (његови суфикси) такође имају то својство, док ако неки сегмент нема то својство, онда то својство нема ни један сегмент испред њега (његов префикс). Зато је довољно да пронађемо најдужи могући сегмент који се завршава на позицији $kraj$ и којем су сви елементи различити. Ако је то сегмент $[pocetak, kraj]$ онда ће такви бити и сегменти $[pocetak+1, kraj]$, ..., $[kraj-1, kraj]$. Њих је укупно $kraj - pocetak$.

Остаје питање како одредити позицију $pocetak$. Претпоставимо да већ знамо решење за претходну позицију краја, тј. претпоставимо да знамо да је $[pocetak, kraj-1]$, најдужи сегмент који има све различите елементе и који се завршава на позицији $kraj-1$ (у старту можемо и $pocetak$ иницијализовати на нулу, а $kraj$ на јединицу, знајући да $[0,0]$ не садржи дупликате и најдужи је такав који се завршава на позицији нула). Ако се елемент на позицији $kraj$ не садржи у том сегменту, онда је сегмент $[pocetak, kraj]$ наш тражени. Ако се садржи, онда је он сигурно једини дупликат у сегменту $[pocetak, kraj]$. Ако претпоставимо да се елемент на позицији $kraj$ у том сегменту јавља и на позицији p, тада је сегмент који тражимо $[p+1, kraj]$, зато што сви сегменти који почињу од позиције $pocetak$, па све до позиције p садрже исти тај дупликат. Сегмент $[p+1, kraj]$ не садржи дупликате и најдужи је такав сегмент, тако да у тој ситуацији $pocetak$ треба поставити на вредност $p+1$.

На крају, остаје питање како утврдити да ли се a_{kraj} јавља у сегменту $[pocetak, kraj-1]$ и ако се јавља, како одредити позицију p на којој се јавља. Директно решење подразумева линеарну претрагу сегмента приликом сваког проширења низа, што би знатно деградирало сложеност целог алгоритма. Боље решење је да се чува асоцијативни низ (мапа, речник) у којем се елементи низа из сегмента позиција $[pocetak, kraj-1]$ пресликавају у њихове позиције. Тада се једноставном претрагом мапе тј. речника (чија је сложеност константна или највише логаритамска) утврђује да ли се нови крајњи елемент јавља у претходном сегменту и на исти начин се одређује и његова позиција. Рецимо и да се приликом померања почетка на позицију $p+1$ сегмент скраћује, што треба да се ослика и у мапи - зато је тада потребно из мапе уклонити све елементе који се јављају у низу, на позицијама од $pocetak$, па закључно са p.

Приметимо одређену сличност овог алгоритма са оним приказаним у задатку Најкраћа подниска која садржи све дате карактере, где смо такође анализирали сегменте са фиксираним крајем и ослањали се на познато решење у којем је крај био једну позицију пре текуће.

Анализа сложености. И почетак и крај се само увећавају (никада се не умањују), што значи да се укупно изврши највише $O(n)$ корака у којима се врши претрага мапе, што значи да је сложеност највише $O(n \log n)$ – ако се користи мапа заснована на хеширању, сложеност је $O(n)$, уз могуће мало веће заузеће меморије.

```
// broj nepraznih segmenata niza sa svim razlicitim elementima
int brojSegmenataSaRazlicitimElementima(const vector<int>& a) {
  // broj elemenata niza
  int n = a.size();

  // ukupan broj segmenata niza ciji su svi elementi razliciti
  int broj = 0;

  // za svaku poziciju kraj zelimo da pronadjemo najduzi segment
  // oblika [pocetak, kraj] koji ima sve razlicite elemente

  // za svaki element u tekucem segmentu [pocetak, kraj] pamtimo
```

```
  // poziciju na kojoj se pojavljuje
  unordered_map<int, int> prethodno_pojavljivanje;

  int pocetak = 0;
  for (int kraj = 0; kraj < n; kraj++) {
    // karakter a[kraj] se vec javio u segmentu [pocetak, kraj-1]?
    if (prethodno_pojavljivanje.find(a[kraj]) != prethodno_pojavljivanje.end()) {
      // nijedan segment koji se zavrsava na poziciji kraj, a pocinje
      // pre ranijeg pojavljivanja elementa a[kraj] ne moze da ima sve
      // razlicite elemente, pa zato razmatramo samo segmente koji se
      // zavrsavaju na poziciji kraj i pocinju iza pozicije tog
      // prethodnog pojavljivanja - najduzi takav pocinje na prvoj
      // poziciji iza te pozicije
      int novi_pocetak = prethodno_pojavljivanje[a[kraj]] + 1;
      // brisemo iz segmenta sve elemente od starog do ispred novog pocetka
      // i mapu uskladjujemo sa time
      for (int i = pocetak; i < novi_pocetak; i++)
        prethodno_pojavljivanje.erase(a[i]);
      // pomeramo pocetak
      pocetak = novi_pocetak;
    }
    // prosirujemo segment elementom a[kraj], pa pamtimo poziciju
    // njegovog pojavljivanja
    prethodno_pojavljivanje[a[kraj]] = kraj;

    // [pocetak, kraj] sadrzi sve razlicite elemente i on je najduzi
    // takav koji se zavrsava na poziciji kraj
    // sigurno su takvi i [pocetak+1, kraj], ..., [kraj-1, kraj]
    // njih ima (kraj - pocetak) i taj broj dodajemo na ukupan broj
    // trazenih segmenata
    broj += kraj - pocetak;
  }

  // vracamo ukupan broj pronadjenih segmenata
  return broj;
}
```

4.2 Стек

Стек представља колекцију података у коју се подаци додају по LIFO (енгл. last-in-first out) принципу - елемент се може додати само на врх и скинути само са врха стека.

У језику C++ стек се реализује класом `stack<T>` где `T` представља тип елемената на стеку. За њено коришћење потребно је укључити заглавље `<stack>`. Подржане су следеће методе (све сложености $O(1)$):

- `push` - поставља дати елемент на врх стека
- `pop` - скида елемент са врха стека (под претпоставком да стек није празан – ако је стек празан, понашање је недефинисано и може доћи до насилног прекида рада програма). Нагласимо да је ова метода типа `void` и да не враћа уклоњени елемент.
- `top` - очитава елемент на врху стека (под претпоставком да стек није празан)
- `empty` - проверава да ли је стек празан
- `size` - враћа број елемената на стеку.

Ако се на стеку чувају уређени парови или n-торке, тада се уместо методе `push`, може користити метода `emplace`, којој се само редом наводе елементи пара тј. n-торке (није потребно посебно позивати функцију за креирање пара тј. n-торке, што је неопходно када се користи `push`).

Стек у језику C++ је заправо само адаптер око неке колекције података (подразумевано вектора) који корисника тера да поштује правила приступа стеку и спречава да направи операцију која над стеком није допуштена (попут приступа неком елементу испод врха).

Задатак: Линије у обратном редоследу

Напиши програм који исписује све линије које се учитају са стандардног улаза у обратном редоследу од редоследа учитавања.

Улаз: Са стандардног улаза се учитавају линије текста, све до краја улаза.

Излаз: На стандардни излаз исписати учитане линије у обратном редоследу.

Пример

Улаз	*Излаз*
zdravo	dan
svete	dobar
dobar	svete
dan	zdravo

Решење

Једно од могућих решења је да се све учитане линије сместе на стек, а да се затим испишу узимајући једну по једну са стека. Пошто стек функционише по принципу LIFO (last in first out, тј. онај који последњи уђе, први излази), редослед ће бити обрнут (најкасније додата линија биће прва скинута и исписана, док ће прва постављена линија бити скинута и исписана последња).

```
stack<string> s;
string linija;
while (getline(cin, linija))
  s.push(linija);
while (!s.empty()) {
  cout << s.top() << endl;
  s.pop();
}
```

Задатак: Историја веб-прегледача

Прегледач веба памти историју посећених сајтова и корисник има могућност да се враћа унатраг на сајтове које је раније посетио. Написати програм који симулира историју прегледача тако што се учитавају адресе посећених сајтова (свака у посебном реду), а када се учита ред у коме пише `back` прегледач се враћа на последњу посећену страницу. Ако се наредбом `back` вратимо на почетну страницу, исписати `-`. Ако се на почетној страници изда наредба `back`, остаје се на почетној страници. Програм треба да испише све сајтове које је корисник посетио.

Улаз: Са стандардног улаза се учитавају веб-адресе, свака у посебној линији, њих највише 1000.

Излаз: На стандардни излаз исписати редом сајтове који се посећују.

Пример

Улаз	*Излаз*
http://www.google.com	http://www.google.com
http://www.rts.rs	http://www.rts.rs
back	http://www.google.com
http://www.petlja.org	http://www.petlja.org
http://www.matf.bg.ac.rs	http://www.matf.bg.ac.rs
back	http://www.petlja.org
back	http://www.google.com
back	-
back	-

Решење

Историја прегледача се понаша као стек јер се елементи само могу скидати и постављати на један крај историје (као последње посећени) и са тог краја се могу и скидати.

```
stack<string> istorija;
string linija;
while (getline(cin, linija)) {
  if (linija == "back") {
```

```
    if (!istorija.empty())
      istorija.pop();
    if (!istorija.empty())
      cout << istorija.top() << endl;
    else
      cout << "-" << endl;
  }
  else {
    cout << linija << endl;
    istorija.push(linija);
  }
}
```

Види другачија решења овог задатка.

Задатак: Push-pop реконструкција

Током рада са стеком укупно n пута је извршена операција `push` којом се нека вредност поставља на врх стека и укупно n пута је извршена операција `pop` којом је елемент скинут са врха стека. Ако је познат низ бројева који су редом били аргументи операције `push` и низ бројева који су редом добијани као резултат операције `pop`, напиши програм који одређује редослед операција `push` и `pop`.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^5$), а затим два низа од по n бројева раздвојених размацима. Претпоставити да су и у једном и у другом низу сви елементи различити.

Излаз: На стандардни излаз исписати редослед операција `push` и `pop` или `-`, ако такав редослед операција није могуће пронаћи за задате низове.

Пример 1

Улаз	*Излаз*
5	push
1 2 3 4 5	push
5 4 3 2 1	push
	push
	push
	pop
	pop
	pop
	pop
	pop

Пример 2

Улаз	*Излаз*
5	push
1 2 3 4 5	push
3 2 5 4 1	push
	pop
	pop
	push
	push
	pop
	pop
	pop

Пример 3

Улаз	*Излаз*
5	-
1 2 3 4 5	
5 4 3 1 2	

Решење

Задатак ћемо решити симулирањем операција са стеком. Пролазимо са два показивача кроз низове `push` и `pop`.

- Ако је текући елемент на врху стека којег одржавамо једнак текућем елементу у низу `pop`, тада вршимо операцију `pop`. Заиста, то је једина могућност јер ако бисмо извршили операцију `push` уместо `pop`, на врху стека би се појавио неки елемент различит од тог који је тренутно на врху и он би морао бити уклоњен пре овог који је тренутно на врху, што значи да наредна операција `pop` не би вратила елемент који је тренутно на реду у низу `pop`.
- Ако текући елемент на врху стека којег одржавамо није једнак текућем елементу у низу `pop` или ако је стек празан, тада проверавамо да ли постоји још елемената у низу `push`.
  - Ако постоји још елемената, вршимо операцију `push` и наредни елемент низа `push` постављамо на стек. Заиста, то је једина могућност, јер ако ако је стек празан, `pop` није могућ, а ако стек није празан и ако бисмо извршили операцију `pop` уместо `push`, тада би резултат примене операције `pop` био различит од онога наметнутог учитаним низом `pop`.
  - Ако је низ `push` празан, тада решење не постоји. Заиста, на основу претходне дискусије знамо да није могућа ни операција `pop`, нити је могућа операција `push`.

Пример. Прикажимо рад алгоритма на примеру у ком су операције `push` биле у редоследу `1 2 3 4 5`, а операције `pop` у редоследу `3 2 5 4 1`.

- Иницијално је стек празан и крећемо од почетка низа `push`.
- Пошто је стек празан на њега постављамо елемент `1`, исписујемо `push`, а стање стека је `[1]`.
- Пошто је елемент на врху стека (а то је `1`) различит од текућег елемента у низу `pop` (а то је елемент `3`), на стек постављамо наредни елемент низа `push` (а то је `2`), исписујемо `push`, а стање стека је `[1, 2]`.
- Пошто је елемент на врху стека (а то је `2`) различит од текућег елемента у низу `pop` (а то је елемент `3`), на стек постављамо наредни елемент низа `push` (а то је `3`), исписујемо `push`, а стање стека је `[1, 2, 3]`.
- Пошто је елемент на врху стека (а то је `3`) једнак текућем елементу у низу `pop`, скидамо тај елемент са врха стека, прелазимо на наредни елемент низа `pop`, исписујемо `pop`, а стање стека је `[1, 2]`.
- Пошто је елемент на врху стека (а то је `2`) једнак текућем елементу у низу `pop`, скидамо тај елемент са врха стека, прелазимо на наредни елемент низа `pop`, исписујемо `pop`, а стање стека је `[1]`.
- Пошто је елемент на врху стека (а то је `1`) различит од текућег елемента у низу `pop` (а то је елемент `5`), на стек постављамо наредни елемент низа `push` (а то је `4`), исписујемо `push`, а стање стека је `[1, 4]`.
- Пошто је елемент на врху стека (а то је `4`) различит од текућег елемента у низу `pop` (а то је елемент `5`), на стек постављамо наредни елемент низа `push` (а то је `5`), исписујемо `push`, а стање стека је `[1, 4, 5]`.
- Пошто је елемент на врху стека (а то је `5`) једнак текућем елементу у низу `pop`, скидамо тај елемент са врха стека, прелазимо на наредни елемент низа `pop`, исписујемо `pop`, а стање стека је `[1, 4]`.
- Пошто је елемент на врху стека (а то је `4`) једнак текућем елементу у низу `pop`, скидамо тај елемент са врха стека, прелазимо на наредни елемент низа `pop`, исписујемо `pop`, а стање стека је `[1]`.
- Пошто је елемент на врху стека (а то је `1`) једнак текућем елементу у низу `pop`, скидамо тај елемент са врха стека, прелазимо на наредни елемент низа `pop`, исписујемо `pop`, а стање стека је `[]`.
- На крају се стек испразнио, оба низа су у потпуности обрађена, па је реконструкција била успешна.

```
enum Naredba {PUSH, POP};

bool PushPop(const vector<int>& push, const vector<int>& pop, int n, vector<Naredba>& naredbe) {
  // stek ciji rad simuliramo
  stack<int> stek;
  naredbe.reserve(2*n);
  // dok ne obradimo kompletno oba niza
  int push_i = 0, pop_i = 0;
  while (push_i < n || pop_i < n) {
    if (!stek.empty() && stek.top() == pop[pop_i]) {
      // ako se na vrhu steka nalazi naredni element koga treba
      // skinuti, skidamo ga
      naredbe.push_back(POP);
      stek.pop();
      pop_i++;
    } else if (push_i < n) {
      // u suprotnom, ako postoji naredni element koji treba
      // postaviti na stek, postavljamo ga
      naredbe.push_back(PUSH);
      stek.push(push[push_i]);
      push_i++;
    } else {
      // posto ne mozemo ni postaviti ni skinuti element sa steka,
      // resenje ne postoji
      return false;
    }
  }
  return true;
}
```

Задатак: Линијски едитор

Едитор омогућава куцање једне линије текста. На почетку је линија празна и курсор се налази на почетку. Корисник може да куца слова, помера курсор лево и десно и брише слово (испред или иза курсора). Курсор никада не може да испадне ван граница текста (када се притисне тастер лево док је курсор на почетку или десно док је курсор на крају текста, он се не помера).

Улаз: Са стандардног улаза се уноси ниска карактера која описује акције корисника. Акције су следеће:

- `iX` – корисник је откуцао карактер `X` (insert `X`)
- `<` – корисник је притиснуо тастер лево
- `>` – корисник је притиснуо тастер десно
- `b` – корисник је притиснуо тастер `backspace` за брисање карактера иза курсора
- `d` – корисник је притиснуо тастер `delete` за брисање карактера испред курсора

Излаз: На стандардни излаз исписати линију текста добијену извршавањем свих команди.

Пример 1

Улаз *Излаз*

```
iaib<bic>>    cb
```

Објашњење

```
Текст           Наредба   Значење
|               ia        куцање карактера a
a|              ib        куцање карактера b
ab|             <         тастер лево
a|b             b         тастер за брисање карактера иза курсора
|b              ic        куцање карактера c
c|b             >         тастер десно
cb|             >         тастер десно
cb|
```

Пример 2

Улаз

```
izidiriaivio<<<<<dbib<<<i!>>>>>i.
```

Излаз

```
!bravo.
```

Решење

Решење грубом силом одржава ниску карактера у коју умеће и из које брише карактере. У језику C++ позицију курсора је најједноставније одржавати у виду итератора (он, на пример, може да указује на карактер који је непосредно десно од позиције курсора, тј. на `end` када је курсор на крају ниске). Уметање се може вршити методом `insert`, а брисање методом `erase`.

Анализа сложености. Сложеност операција брисања и уметања карактера је линеарна у односу на дужину ниске, па је укупна сложеност оваквог приступа у најгорем случају квадратна (најгори случај је ако се уметање и брисање врше негде близу почетка или бар средине ниске).

```
string str = "";

string naredbe;
cin >> naredbe;
int i = 0;
auto it = begin(str);
while (i < naredbe.size()) {
  char naredba = naredbe[i++];
  if (naredba == '<') {
    if (it > begin(str))
      it--;
  } else if (naredba == '>') {
```

```
    if (it < end(str))
      it++;
  } else if (naredba == 'i') {
    char c = naredbe[i++];
    it = str.insert(it, c);
    it++;
  } else if (naredba == 'b') {
    if (it > begin(str)) {
      it--;
      it = str.erase(it);
    }
  } else if (naredba == 'd') {
    if (it < end(str))
      it = str.erase(it);
  }
}

cout << str << endl;
```

Ефикасније решење се може добити ако се уместо ниске чува листа карактера. У језику C++ постоје две колекције реализоване помоћу повезаних листа. Колекција `forward_list<T>` представља једноструко повезане листе, док колекција `list<T>` представља двоструко повезане листе. За њихово коришћење је потребно укључити заглавља `<forward_list>` тј. `<list>`. Основне методе за рад са листама су следеће (све су сложености $O(1)$):

- `begin()` - враћа итератор на почетак листе
- `end()` - враћа итератор на крај листе
- `insert(it, x)` - умеће елемент на позицију која је непосредно лево од датог итератора и враћа итератор на елемент који је уметнут
- `erase(it)` - брише елемент на који указује дати итератор и враћа итератор који указује на позицију непосредно десно од обрисаног елемента (што може бити и `end`, ако је обрисан последњи елемент).

На итератор код колекције `forward_list` могуће је примењивати оператор `++`, чиме се он помера на наредни елемент листе, а код колекције `list` и оператор `--`, чиме се он помера на претходни елемент листе. И ове операције су сложености $O(1)$.

Позицију курсора у овом задатку је најједноставније одржавати у виду итератора (он, на пример, може да указује на карактер који је непосредно десно од позиције курсора, тј. на `end` када је курсор на крају ниске). Пошто се курсор помера у оба смера, потребно је користити колекцију `list<char>`, а не `forward_list<char>`. Уметање се онда може вршити методом `insert`, а брисање методом `erase`.

Анализа сложености. Пошто су код листе операције уметања и брисања, као и померања итератора за једно место константне сложености, овај алгоритам је линеарне сложености.

```
string naredbe;
cin >> naredbe;
int i = 0;
list<char> str;
auto it = begin(str);
while (i < naredbe.size()) {
  char naredba = naredbe[i++];
  if (naredba == '<') {
    if (it != begin(str))
      it--;
  } else if (naredba == '>') {
    if (it != end(str))
      it++;
  } else if (naredba == 'i') {
    char c = naredbe[i++];
    it = str.insert(it, c);
    it++;
```

```
  } else if (naredba == 'b') {
    if (it != begin(str)) {
      it--;
      it = str.erase(it);
    }
  } else if (naredba == 'd') {
    if (it != end(str))
      it = str.erase(it);
  }
}
cout << string(begin(str), end(str)) << endl;
```

Могуће је и веома једноставно и елегантно решење коришћењем два стека. У једном стеку се чувају карактери лево од тренутне позиције курсора (тако да се на врху налази карактер најближи курсору), а у другом карактери десно од тренутне позиције курсора (тако да се на врху поново налази карактер најближи курсору). Куцање карактера се тада реализује уметањем карактера на леви стек, померање налево се реализује пребацивањем карактера са врха левог, на врх десног стека, померање надесно ради обратно, док се брисање карактера десно од курсора своди на скидање елемента са врха левог, а лево од курсора на скидање елемената са врха десног стека. Ако је неки од стекова празан, операције скидања елемента са њега се просто прескачу.

Анализа сложености. Пошто су операције скидања и додавања елемената на врх стека константне сложености, укупна сложеност овог алгоритма је линеарна.

```
string stekUString(stack<char>& stek) {
  string str = "";
  while (!stek.empty()) {
    str += stek.top();
    stek.pop();
  }
  return str;
}


int main() {
  stack<char> levo, desno;
  string naredbe;
  cin >> naredbe;
  int i = 0;
  while (i < naredbe.size()) {
    char naredba = naredbe[i++];
    if (naredba == '<') {
      if (!levo.empty()) {
        desno.push(levo.top());
        levo.pop();
      }
    } else if (naredba == '>') {
      if (!desno.empty()) {
        levo.push(desno.top());
        desno.pop();
      }
    } else if (naredba == 'i') {
      levo.push(naredbe[i++]);
    } else if (naredba == 'b') {
      if (!levo.empty())
        levo.pop();
    } else if (naredba == 'd') {
      if (!desno.empty())
        desno.pop();
    }
  }
```

```
  string str = stekUString(levo);
  reverse(begin(str), end(str));
  str += stekUString(desno);
  cout << str << endl;
  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Сортирање бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Нерекурзивно брзо сортирање

Једна веома важна употреба стека је за реализацију рекурзије. Током извршавања рекурзивних функција, на стек се смештају вредности локалних променљивих и аргумената сваког активног позива функције. Рекурзију увек можемо уклонити и уместо системског стека можемо ручно одржавати стек са тим подацима. Прикажимо ову технику уклањања рекурзије на примеру нерекурзивне имплементације алгоритма брзог сортирања. Нагласимо да је код алгоритма QuickSort дубина рекурзије мала (она логаритамски зависи од броја елемената низа), па се њеном елиминацијом не добија ништа значајно.

На стеку ћемо чувати аргументе рекурзивних позива функције сортирања. На почетку је то пар индекса $(0, n-1)$. Главна петља се извршава све док се стек не испразни и у њој се обрађује пар индекса који се скида са врха стека. Уместо рекукрзивних позива њихове ћемо аргументе постављати на врх стека и чекати да они буду обрађени у некој од наредних итерација петље. Приметимо да се аргументи другог рекурзивног позива обрађују тек када се у потпуности реши потпроблем који одговара првом рекурзивном позиву, што одговара понашању функције када је заиста имплементирана рекурзивно.

Напомена. Рецимо и да је ова техника општа и да се рекурзија увек може елиминисати на овај начин. За разлику од тога, елиминисање специфичних облика рекурзије (попут, на пример, репне) није увек применљиво, али када јесте, доводи до боље меморијске (па и временске) ефикасности јер се не користи стек.

```
void quick_sort(vector<int>& a) {
  // duzina niza
  int n = a.size();
  // stek na kome čuvamo argumente rekurzivnih poziva
  stack<pair<int, int>> sortirati;
  // sortiranje kreće od obrade celog niza tj. pozicija (0, n-1)
  sortirati.emplace(0, n - 1);
  while (!sortirati.empty()) {
    // skidamo par (l, d) sa vrha steka
    auto p = sortirati.top();
    int l = p.first, d = p.second;
    sortirati.pop();
    // obrađujemo par (l, d) na isti način kao u rekurzivnoj implementaciji
    if (d - l < 1)
      continue;
    int k = l;
    for (int i = l+1; i <= d; i++)
      if (a[i] < a[l])
        swap(a[++k], a[i]);
    swap(a[k], a[l]);
    // umesto rekurzivnih poziva njihove argumente
    // postavljamo na stek
    sortirati.emplace(k+1, d);
    sortirati.emplace(l, k-1);
  }
}
```

Види другачија решења овог задатка.

Задатак: Вредност постфиксног израза

Префиксна нотација се понекад назива и пољска нотација, а постфиксна нотација се понекад назива и обратна пољска нотација (енгл. reverse polish notation, RPN) у част пољског логичара Јана Лукашијевича, који ју је изумео. Она подразумева да се бинарни оператори уместо између операнада записују након њих. На пример, уместо `3 + 5`, писаћемо `3 5 +`. Напиши програм који одређује вредност постфиксно записаног израза.

Улаз: Са стандардног улаза се учитава постфиксно записан израз који садржи једноцифрене бројеве и операторе `+` и `*` (без размака).

Излаз: На стандардни излаз исписати вредност учитаног израза.

Пример 1

Улаз	*Излаз*
`12+3*`	`9`

Објашњење

Постфиксно је записан израз `(1+2)*3`.

Пример 2

Улаз

```
11+2*345+*+
```

Излаз

```
31
```

Објашњење

Постфиксно је записан израз `(1+1)*2+3*(4+5)`.

Решење

Велика предност постфиксно записаних израза је то што им се вредност веома једноставно израчунава уз помоћ стека. Када наиђемо на број постављамо га на врх стека. Када наиђемо на оператор, скидамо две вредности са врха стека, примењујемо на њих одговарајућу операцију и резултат постављамо на врх стека. Вредност целог израза се на крају налази на врху стека.

Пример. Нпр. за израз `34+5*` на стек постављамо `3`, затим `4`, након тога те две вредности уклањамо и постављамо `7`, затим постављамо `5` и на крају уклањамо `7` и `5` и мењамо их са `35`.

```
bool jeOperator(char c) {
  return c == '+' || c == '*';
}

int primeniOperator(char op, int op1, int op2) {
  int v = 0;
  switch(op) {
     case '+': v = op1 + op2; break;
     case '*': v = op1 * op2; break;
  }
  return v;
}

int vrednost(const string& izraz) {
   stack<int> st;
   for (char c : izraz) {
      if (isdigit(c))
        st.push(c - '0');
      else if (jeOperator(c)) {
         int op2 = st.top(); st.pop();
```

```
        int op1 = st.top(); st.pop();
        st.push(primeniOperator(c, op1, op2));
      }
    }
    return st.top();
}
```

Задатак: Превођење потпуно заграђеног израза у постфиксни облик

Написати програм који исправан инфиксни аритметички израз који има заграде око сваке примене бинарног оператора преводи у постфиксни облик. Једноставности ради претпоставити да су сви операнди једноцифрени бројеви и да се јављају само операције сабирања и множења.

Улаз: Једина линија стандардног улаза садржи исправан, потпуно заграђен израз.

Излаз: На стандардни излаз исписати тражени постфиксни облик.

Пример

Улаз	*Излаз*
`(((3*5)+(7+(2*1)))*4)`	`35*721*++4*`

Решење

Чињеница да је израз потпуно заграђен олакшава израчунавање, јер нема потребе да водимо рачуна о приоритету и асоцијативности оператора. Такви изрази се описују наредном, веома једноставном граматиком.

```
<izraz> :: <cifra>
<izraz> :: '(' <izraz> '+' <izraz> ')'
<izraz> :: '(' <izraz> '*' <izraz> ')'
```

Један начин да се приступи решавању проблема је да се примени индуктивно-рекурзивни приступ. Обрада структурираног улаза рекурзивним функцијама се назива *рекурзивни спуст* и детаљно се изучава у курсевима превођења програмских језика. Дефинишемо рекурзивну функцију чији је задатак да преведе део ниске који представља исправан инфиксни израз. Он може бити или број, када је превођење тривијално јер се он само препише на излаз или израз у заградама. У овом другом случају читамо отворену заграду, затим рекурзивним позивом преводимо први операнд, након тога читамо оператор, затим рекурзивним позивом преводимо други операнд, након тога читамо затворену заграду и исписујемо оператор који смо прочитали (он бива исписан непосредно након превода својих операнада).

Променљива `i` мења своју вредност кроз рекурзивне позиве. Стога ћемо је преносити по референци тако да представља и улазну и излазну величину функције. Задатак функције је да прочита израз који почиње на позицији i, да га преведе у постфиксни облик и да променљиву `i` промени тако да њена нова вредност i' указује на позицију ниске непосредно након израза који је преведен.

```
// Prevodi deo izraza od pozicije i u postfiksni oblik i rezultat
// nadovezuje na nisku postfiks. Po zavrsetku rada funkcije,
// promenljiva i ukazuje na poziciju iza prevedenog izraza.
void prevedi(const string& izraz, int& i, string& postfiks) {
  if (isdigit(izraz[i]))
    postfiks += izraz[i++];
  else {
    // preskačemo otvorenu zagradu
    i++;
    // prevodimo prvi operand
    prevedi(izraz, i, postfiks);
    // pamtimo operator
    char op = izraz[i++];
    // prevodimo drugi operand
    prevedi(izraz, i, postfiks);
    // preskačemo zatvorenu zagradu
    i++;
    // ispisujemo upamćeni operator
    postfiks += op;
```

```
  }
}

// prevodi potpuno zagradjen izraz u postfiksni oblik
string prevedi(const string& izraz) {
  string postfiks = "";
  int i = 0;
  prevedi(izraz, i, postfiks);
  return postfiks;
}
```

Решење техником рекурзивног спуста се може прерадити тако да се добије нерекурзивна имплементација. Да бисмо се ослободили рекурзије, потребно је да употребимо стек. Кључна опаска је да се у стек оквиру функције, пре рекурзивног позива за превођење другог операнда памти оператор. Ово нам сугерише да нам је за нерекурзивну имплементацију неопходно да одржавамо стек на који ћемо смештати операторе. Када наиђемо на број преписујемо га на излаз, када наиђемо на оператор стављамо га на стек, а када наиђемо на затворену заграду скидамо и исписујемо оператор са врха стека.

Пример. Размотримо, на пример, израз `((3+4)*(5+2))`

- Први карактер је отворена заграда коју прескачемо.
- Наредни карактер је отворена заграда коју прескачемо.
- Наредни карактер је цифра, коју исписујемо (до сада је исписано `3`).
- Наредни карактер је оператор `+`, који постављамо на стек. Стек је сада `+`.
- Наредни карактер је цифра, коју исписујемо (до сада је исписано `34`).
- Наредни карактер је затворена заграда, па исписујемо оператор са врха стека (до сада је исписано `34+`). Стек је сада празан.
- Наредни карактер је оператор `*` који постављамо на стек. Стек је сада `*`.
- Наредни карактер је отворена заграда коју прескачемо.
- Наредни карактер је цифра, коју исписујемо (до сада је исписано `34+5`)
- Наредни карактер је оператор `+` који постављамо на стек. Стек је сада `*+`.
- Наредни карактер је цифра, коју исписујемо (до сада је исписано `34+52`)
- Наредни карактер је затворена заграда, па исписујемо оператор са врха стека (до сада је исписано `34+52+`). Стек је сада `*`.
- Наредни карактер је затворена заграда, па исписујемо оператор са врха стека (до сада је исписано `34+52+*`). Стек је сада празан.

```
bool jeOperator(char c) {
  return c == '+' || c == '*';
}

string prevedi(const string& izraz) {
  string postfiks;
  stack<char> operatori;
  for (char c : izraz) {
    if (isdigit(c))
      postfiks += c;
    else if (c == ')') {
      postfiks += operatori.top();
      operatori.pop();
    } else if (jeOperator(c))
      operatori.push(c);
  }
```

```
  return postfiks;
}
```

Задатак: Вредност израза

Написати програм којим се израчунавају и приказују вредности датих аритметичких израза. Сваки израз је исправно задат, састоји се од природних бројева и операција +, -, * и / (целобројно дељење). Коришћењем проширене Бекусове нотације (EBNF), синтаксу израза можемо описати на следећи начин:

```
<izraz>  ::= <term> {<operacija1> <term>}
<term>   ::= <faktor> {<operacija2> <faktor>}
<faktor> ::= <broj> | '(' <izraz> ')'
<broj>   ::= <cifra> {<cifra>}
<cifra> ::= '0' | '1' | ... | '9'
<operacija1> ::= '+' | '-'
<operacija2> ::= '*' | '/'
```

Улаз: У свакој линији стандарног улаза налази се исправан израз (израз не садржи размаке).

Излаз: Свака линија стандардног излаза садржи редом вредности израза датих на стандардном улазу, свака вредност у посебној линији. Ако израз није дефинисан, због дељења 0, приказати поруку `deljenje nulom`.

Пример

Улаз	*Излаз*
`1+2*3-4`	`3`
`2*3-5*(100-8*12)`	`-14`
`123-43*(12-3*5)/(17-35/2)`	`deljenje nulom`
`12/5+2`	`4`

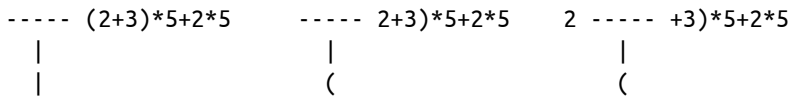
Решење

Израчунавање вредности израза је опет комбинација превођења у постфиксни облик и израчунавања вредности постфиксног израза.

Проблем се решава слично као код потпуно заграђених израза, али овај пут се мора обраћати пажња на приоритет и асоцијативност оператора. Решење се може направити рекурзивним спустом, али се тиме нећемо бавити у овом курсу. Кључна дилема је шта радити у ситуацији када се прочита `op2` у изразу облика `i1 op1 i2 op2 i3` где су `i1`, `i2` и `i3` три израза (било броја било израза у заградама), а `op1` и `op2` два оператора. У том тренутку на излазу ће се налазити израз `i1` преведен у постфиксни облик и иза њега израз `i2` преведен у постфиксни облик, док ће се оператор `op1` налазити на врху стека оператора. Уколико `op1` има већи приоритет од оператора `op2` или уколико им је приоритет исти, али је асоцијативност лева, тада је потребно прво израчунавати израз `i1 op1 i2` тиме што се оператор `op1` са врха стека пребаци на излаз. У супротном (ако `op2` има већи приоритет или ако је приоритет исти, а асоцијативност десна) оператор `op1` остаје на стеку и изнад њега се поставља оператор `op2`.

Ово је један од многих алгоритама које је извео Едсгер Дејкстра и назива се на енглеском језику *Shunting yard algortihm*, што би се могло слободно превести као алгоритам сортирања железничких вагона. Замислимо да израз треба да пређе са једног на други крај пруге. На прузи се налази споредни колосек (пруга је у облику слова T и споредни колосек је усправна црта). Делови израза прелазе са десног на леви крај (замислимо да иду по горњој ивици слова T). Бројеви увек прелазе директно. Оператори се увек задржавају на споредном колосеку, али тако да се пре него што оператор уђе на споредни колосек са њега на излаз пребацују сви оператори који су вишег приоритета у односу на текући или имају исти приоритет као текући а лево су асоцијативни. И отворене заграде се постављају на споредни колосек, а када наиђе затворена заграда са споредног колосека се уклањају сви оператори до отворене заграде. Када се исцрпи цео израз на десној страни, сви оператори са споредног колосека се пребацују на леву страну. Јасно је да споредни колосек има понашање стека, тако да имплементацију можемо направити коришћењем стека на који ћемо стављати операторе.

Пример. Илуструјмо ову железничку аналогију једним примером.



```
     2 ----- 3)*5+2*5     23 ----- )*5+2*5     23+ ----- *5+2*5
         +                    +                    |
         (                    (                    |


   23+ ----- 5+2*5      23+5 ----- +2*5      23+5* ----- 2*5
         |                    |                    |
         *                    *                    +

23+5*2 ----- *5       23+5*2 ----- 5       23+5*25 -----
         |                    *                    *
         +                    +                    +

                  23+5*25*+ -----
                               |
                               |
```

```
// provera da li je karakter aritmeticki operator
bool jeOperator(char c) {
  return c == '+' || c == '-' || c == '*' || c == '/';
}

// prioritet datog operatora
int prioritet(char c) {
  if (c == '+' || c == '-')
    return 1;
  else if (c == '*' || c == '/')
    return 2;
  // greska
  return 0;
}

// primenjuje datu operaciju na dve vrednosti na vrhu steka,
// zamenjujući ih sa rezultatom primene te operacije
// vracamo informaciju o tome da li operator uspesno primenjen ili je
// doslo do deljenja nulom
bool primeni(stack<char>& operatori, stack<int>& vrednosti) {
  // operator se nalazi na vrhu steka operatora
  char op = operatori.top(); operatori.pop();
  // operandi se nalaze na vrhu steka operatora
  int op2 = vrednosti.top(); vrednosti.pop();
  int op1 = vrednosti.top(); vrednosti.pop();

  // izracunavamo vrednost izraza
  int v = 0;
  if (op == '+') v = op1 + op2;
  else if (op == '-') v = op1 - op2;
  else if (op == '*') v = op1 * op2;
  else if (op == '/') {
    // deljenje nulom
    if (op2 == 0)
      return false;
    v = op1 / op2;
  }
  // postavljamo ga na stek operatora
  vrednosti.push(v);
  // operator je uspesno primenjen
  return true;
}
```

```
// izracunavamo vrednost izraza
// vracamo informaciju o tome da li je vrednost uspesno izracunata ili
// je doslo do deljenja nulom
bool vrednost(const string& izraz, int& v) {
  stack<int> vrednosti;
  stack<char> operatori;

  // analiziramo sve karaktere u ulaznom izrazu
  int i = 0;
  while (i < izraz.length()) {
    if (isdigit(izraz[i])) {
      // brojevne konstante postavljamo na stek
      v = izraz[i] - '0';
      i++;
      while (i < izraz.length() && isdigit(izraz[i]))
        v = 10 * v + (izraz[i++] - '0');
      vrednosti.push(v);
    } else if (izraz[i] == '(') {
      // otvorene zagrade postavljamo na stek
      operatori.push('(');
      i++;
    } else if (izraz[i] == ')') {
      // izračunavamo vrednost izraza u zagradi
      while (operatori.top() != '(')
        if (!primeni(operatori, vrednosti))
          return false;
      // uklanjamo otvorenu zagradu
      operatori.pop();
      i++;
    } else if (jeOperator(izraz[i])) {
       // obrađujemo sve prethodne operatore višeg prioriteta
       while (!operatori.empty() && jeOperator(operatori.top()) &&
              prioritet(operatori.top()) >= prioritet(izraz[i]))
         if (!primeni(operatori, vrednosti))
           return false;
       // stavljamo operator na stek
       operatori.push(izraz[i]);
       i++;
    }
  }

  // izračunavamo sve preostale operacije
  while (!operatori.empty())
    if (!primeni(operatori, vrednosti))
      return false;

  // vrednost izraza se nalazi na vrhu steka
  v = vrednosti.top();
  return true;
}
```

4.3 Ред

Ред представља колекцију података у коју се подаци додају по FIFO (енгл. first-in-first-out) принципу – елемент се увек узима са почетка, а додаје на крај реда.

У језику C++ ред се реализује класом `queue<T>` где `T` представља тип елемената на стеку. За њено коришћењем потребно је укључити заглавље `<queue>`. Подржане су следеће методе (све сложености $O(1)$):

- `push` - поставља дати елемент на крај реда
- `pop` - скида елемент са почетка реда (под претпоставком да ред није празан). Нагласимо да је ова метода типа `void` и да не враћа уклоњени елемент.
- `front` - очитава елемент на почетку реда (под претпоставком да ред није празан)
- `empty` - проверава да ли је ред празан
- `size` - враћа број елемената у реду

Ако се у реду чувају уређени парови или n-торке, тада се уместо методе `push`, може користити метода `emplace`, којој се само редом наводе елементи пара тј. n-торке (није потребно посебно позивати функцију за креирање пара тј. n-торке, што је неопходно када се користи `push`).

Ред у језику C++ је заправо само адаптер око неке колекције података (подразумевано реда са два краја) који корисника тера да поштује правила приступа реду и спречава да направи операцију која над редом није допуштена (попут приступа неком елементу који није на почетку).

4.3.1 Ред са два краја

Уопштење представља ред са два краја који допушта да се елементи и додају и узимају са било ког краја реда (та структура података заправо комбинује и функционалност стека и функционалност реда).

У језику C++ могуће је користити структуру `deque<T>`. За њено коришћење потребно је укључити заглавље `<deque>`. Подржане су следеће операције (све сложености $O(1)$).

- `push_front` - додавање елемента на почетак
- `push_back` - додавање елемента на крај
- `front` - читање елемента са почетка (под претпоставком да ред није празан)
- `back` - читање елемента са краја (под претпоставком да ред није празан)
- `pop_front` - уклањање елемента са почетка (под претпоставком да ред није празан). Нагласимо да је ова метода типа `void` и да не враћа уклоњени елемент.
- `pop_back` - уклањање елемента са краја (под претпоставком да ред није празан). Нагласимо да је ова метода типа `void` и да не враћа уклоњени елемент.
- `empty` - провера да ли је ред празан
- `size` - број елемената у реду

Интересантно, захваљујући специфичном начину имплементације, ова структура података подржава и оператор индексног приступа којим се елемент на датој позицији може прочитати или изменити у времену $O(1)$.

Задатак: Сегмент дужине k највећег просека

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Приметимо да је проблем налажења сегмента дужине k чији је просек највећи, еквивалентан проблему налажења сегмента дужине k највећег збира. Просек сегмента добијамо дељењем суме сегмента са дужином сегмента, која је у овом задатку константа и износи k, тако да је просек највећи када је збир највећи.

Смањивање употребљене меморије коришћењем реда

Решење у ком се сви елементи чувају у низу, а затим се инкрементално обрађује један по један сегмент дужине k је временски ефикасно, али се користи превише меморије. Наиме, није неопходно у меморији истовремено чувати све елементе низа, већ је довољно чувати само елементе текућег сегмента (ово може бити битно у случајевима када је меморија критичан ресурс и када је k знатно мање од n). Приликом учитавања сваког новог елемента почетни елемент тренутног сегмента се уклања, а последњи елемент се додаје на крај сегмента. Ово указује на то да је за чување текућег сегмента погодна структура података ред, која нам омогућава да додајемо елементе на крај и уклањамо елементе са почетка (први елемент који је додат први бива и уклоњен из реда). У језику C++ можемо употребити колекцију `queue`. Елементе у ред додајемо методом `push`, уклањамо их методом `pop` (она не враћа вредност). Елемент на почетку реда се може очитати методом `front`.

Анализа сложености. У реду се истовремено чува највише k елемената, па је меморијска сложеност $O(k)$, што може бити доста мање него $O(n)$. Пошто се сваки елемент највише једном убацује и највише једном избацује из реда, а убацивање и избацивање из реда је операција константне сложености, укупна временска сложеност алгоритма је $O(n)$.

```
// broj elemenata niza i duzina segmenta
int n, k;
cin >> k >> n;

// red u kojem u svakom trenutku cuvamo tekuci segment
queue<double> q;

// ucitavamo prvi segment duzine k, smestamo elemente u red i
// racunamo mu sumu
double suma = 0.0;
for (int i = 0; i < k; i++) {
  double x; cin >> x;
  q.push(x);
  suma += x;
}

// trenutna maksimalna suma segmenta i indeks njenog pocetka
int maxPocetak = 0;
double maxSuma = suma;

for (int i = 1; i <= n-k; i++) {
  // ucitavamo naredni element
  double x; cin >> x;
  // azuriramo sumu
  suma = suma - q.front() + x;
  // menjamo "najstariji" element u redu
  q.pop(); q.push(x);
  // ako je potrebno, azuriramo maksimum
  if (suma >= maxSuma) {
    maxSuma = suma;
    maxPocetak = i;
  }
}

// ispisujemo pocetak poslednjeg segmenta sa maksimalnom sumom
// (ujedno i prosekom)
cout << maxPocetak << endl;
```

Задатак: Јосифов проблем

Ђаци седе у кругу обележени бројевима од 0 до $n-1$ и играју се разбрајалице тако да у сваком бројању један ђак испадне. Бројање креће од ђака 0 и сваки m-ти ђак испада. Напиши програм који одређује који ђак ће остатати последњи.

Улаз: У првој линији стандардног улаза налази се почетни број ђака n ($1 \leq n \leq 10^5$), а у другом дужина бројалице m ($2 \leq m \leq n$).

Излаз: На стандардни излаз исписати број преосталог ђака.

Пример

Улаз	*Излаз*
8 3	6

Објашњење

Ђаци који седе у кругу на почетку и након сваког испадања су:

```
0 1 2 3 4 5 6 7
0 1 3 4 5 6 7
0 1 3 4 6 7
1 3 4 6 7
1 3 6 7
3 6 7
3 6
6
```

Решење

Структура података која допушта ефикасно избацивање елемената из средине је листа (нпр. двоструко повезана). Кружно кретање по листи можемо остварити тако што након сваког померања итератора проверимо да ли смо стигли до краја листе и ако јесмо, итератор ручно поново поставимо на почетак листе.

```
// efekat postfiksnog uvecavanja iteratora u kruznoj listi tj. efekat it++
// iterator se pomera na sledece mesto, ali se vraca polazna vrednost
list<int>::iterator uvecaj(list<int>& lista, list<int>::iterator& it) {
  list<int>::iterator polazni = it;
  it++;
  if (it == lista.end())
    it = lista.begin();
  return polazni;
}

int josif(int n, int m) {
  list<int> lista;
  for (int i = 0; i < n; i++)
    lista.emplace_back(i);

  auto it = lista.begin();
  while (lista.size() > 1) {
    for (int i = 0; i < m - 1; i++)
      uvecaj(lista, it);
    lista.erase(uvecaj(lista, it));
  }

  return *it;
}
```

Кружну листу можемо једноставно реализовати коришћењем реда. У сваком кораку бројања једног ђака са почетка реда ћемо пребацити на крај реда. Након пребацивања $m-1$ ученика, оног који је на почетку реда трајно избацујемо.

```
int josif(int n, int m) {
  queue<int> red;
  for (int i = 0; i < n; i++)
    red.push(i);
  while (red.size() > 1) {
    for (int i = 0; i < m - 1; i++) {
      red.push(red.front());
      red.pop();
    }
    red.pop();
  }
  return red.front();
}
```

Задатак: Максимална бијекција

Филмски продуцент организује вечеру на коју жели да позове глумце. Да би се глумци осећали пријатно на вечери, продуцент жели да обезбеди да је сваки глумац присутан на вечери омиљен глумац неког другог глумца присутног на вечери. Сваки од n глумаца, потенцијалних гостију, одабрао је свог омиљеног глумца из тог скупа глумаца (при чему није искључено и да је неки глумац одабрао сам себе). Напиши програм који одређује највећи подскуп тог скупа глумаца који садржи глумце које продуцент може позвати на вечеру.

Улаз: Са стандардног улаза уноси се број n ($1 \leq n \leq 50000$) који представља број глумаца који су гласали, а затим и редом редни бројеви омиљеног глумца сваког глумца (сви бројеви су између 0 и $n-1$).

Излаз: На стандардни излаз испиши највећи број глумаца који могу присуствовати вечери.

Пример

Улаз	*Излаз*
7 2 0 0 4 4 3 5	3

Објашњење

На вечеру могу бити позвани глумци са бројевима $0, 2, 4$. Глумац 0 је омиљени глумац глумца 2, глумац 2 је омиљени глумац глумца 0, док је глумац 4 сам себи омиљен.

Решење

Гласови глумаца одређују функцију f дефинисану на скупу $\{0, 1, \ldots, n-1\}$. Нека је скуп S скуп глумаца који су позвани на вечеру. Да би сваки глумац био омиљен глумац неком другом глумцу из скупа S, потребно је да рестрикција функције f на скуп S буде "на" (тј. да за сваку слику постоји оригинал који се слика у ту слику). Пошто је скуп S коначан, на основу Дирихлеовог принципа, она ће уједно бити и "1-1" (за сваку слику ће постојати тачно један оригинал који се у њу слика). Заиста, ако би неки глумац на вечери био омиљен за два различита глумца, неком глумцу на вечери би недостајао глумац коме би он био омиљен. Функија која је истовремено "на" и "1-1" зове се бијекција.

Елиминација елемената

Ефикасан алгоритам можемо направити ако пронађемо потребан услов да елемент буде део скупа S. Наиме, сваки елемент скупа S мора бити слика тачно једног елемента скупа S. Ако је сваки елемент скупа X (домена функције f) слика тачно једног елемента скупа X, тада је f бијекција на скупу X. У супротном мора да постоји елемент који није слика ни једног елемента скупа X и тај елемент не може бити део скупа S. Када тај елемент уклонимо (заједно са његовом сликом), добијамо скуп који је за један елемент мањи и на који можемо применити исти поступак (суштински, имамо описан индуктивни тј. рекурзивни поступак).

Овај приступ се може једноставно имплементирати тако што за сваки елемент скупа X израчунамо број елемената који се сликају у њега. То можемо урадити коришћењем једноставног, асоцијативног низа. Слична техника је показана као у задатку Фреквенције речи.

Можемо одржавати радну листу (ред) елемената у које се не слика ни један елемент (након израчунавања броја елемената који се сликају у сваки од елемената скупа X, све бројеве за које је вредност у асоцијативном низу нула, убацујемо у радну листу). Након тога, све док се радна листа не испразни, узимамо један по један елемент из радне листе, избацујемо га из скупа X и зато смањујемо број елемената који се сликају у слику тог избаченог елемента (умањујемо вредност у асоцијативном низу). Ако се установи да се након тога вредност слике у асоцијативном низу смањила на нулу, тада се слика убацује у радну листу. Иако редослед узимања елемената из радне листе може бити потпуно произвољан, за имплементацију се најчешће користи ред (јер даје некакав осећај правичности). Решење у ком би се уместо реда користио стек било би такође исправно.

```
vector<int> ulazniStepen(n, 0);
for (int i = 0; i < n; i++)
  ulazniStepen[f[i]]++;
```

```
queue<int> q;
for (int i = 0; i < n; i++)
  if (ulazniStepen[i] == 0)
    q.push(i);
int broj_elemenata = n;
while (!q.empty()) {
  int i = q.front(); q.pop();
  broj_elemenata--;
  if (--ulazniStepen[f[i]] == 0)
    q.push(f[i]);
}
cout << broj_elemenata << endl;
```

Задатак: Сортирање - сви испред мањи или сви испред већи

Бројеви у низу су такви да за сваки елемент важи или да су сви елементи испред њега мањи од њега или да су сви елементи испред њега већи од њега. Нпр. низ $5, 8, 12, 4, 2, 13, 19, 1$ задовољава то својство. Напиши програм који у линеарној сложености сортира тај низ.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^5$), а затим n елемената низа (елементи су дати у једној линији, раздвојени размацима).

Излаз: На стандардни излаз исписати сортиране елементе низа (раздвојене размаком).

Пример

Улаз	*Излаз*
`8` `5 8 12 4 2 13 19 1`	`1 2 4 5 8 12 13 19`

Решење

Ако се не узме у обзир специфичност улазних података, низ се може сортирати било којим алгоритмом за сортирање низа бројева (слично као у задатку Сортирање бројева).

Анализа сложености. Ако се користи библиотечка функција сортирања, временска сложеност овог алгоритма је $O(n \log n)$, док је меморијска сложеност $O(n)$.

```
// ucitavamo niz
int n;
cin >> n;
vector<int> a(n);
for (int i = 0; i < n; i++)
  cin >> a[i];

// sortiramo niz
sort(begin(a), end(a));

// ispisujemo sortiran niz
for (int i = 0; i < n; i++)
  cout << a[i] << " ";
cout << endl;
```

Решимо задатак индуктивном конструкцијом. Претпоставимо да обрађујемо један по један елемент и да смо већ сортирали префикс елемената испред текућег. Ако је он већи од свих елемената испред себе у полазном низу, тада га треба додати на крај сортираног префикса, а ако је мањи од свих елемената испред себе, треба га додати на почетак сортираног префикса.

Пример. Прикажимо сортирање низа `5, 8, 12, 4, 2, 13, 19, 1`.

- На почетку је ред празан, па у њега додајемо 5 (ред постаје `5`).
- 8 је веће од последњег елемента у реду, па га додајемо на крај (ред постаје `5, 8`).
- 12 је веће од последњег елемента у реду, па га додајемо на крај (ред постаје `5, 8, 12`).
- 4 је мање од последњег елемента у реду, па га додајемо на почетак (ред постаје `4, 5, 8, 12`).
- 2 је мање од последњег елемента у реду, па га додајемо на почетак (ред постаје `2, 4, 5, 8, 12`).

- 13 је веће од последњег елемента у реду, па га додајемо на крај (ред постаје `2, 4, 5, 8, 12, 13`).
- 19 је веће од последњег елемента у реду, па га додајемо на крај (ред постаје `2, 4, 5, 8, 12, 13, 19`).
- 1 је мање од последњег елемента у реду, па га додајемо на почетак (ред постаје `1, 2, 4, 5, 8, 12, 13, 19`).

За ефикасну имплементацију потребно је користити структуру података која допушта ефикасно додавање елемената на почетак и на крај и то може бити ред са два краја или двоструко повезана листа. У језику C++ можемо употребити `deque` или `list`. Да би се проверило да ли је елемент већи од свих испред себе или мањи од свих испред себе, довољно је упоредити га са било којим од тих елемената (на пример, са последњим елементом у реду). Када је ред празан, први елемент можемо убацити било на почетак, било на крај. Да бисмо избегли проверу да ли је ред празан приликом обраде сваког елемента, први елемент можемо убацити у празан ред пре него што почнемо са обрадом осталих елемената.

Анализа сложености. И меморијска и временска сложеност овог алгоритма је $O(n)$.

```
// red sa dva kraja u koji smestamo sortiran niz
deque<int> a;

// ucitavamo prvi element i ubacujemo ga u red
int x; cin >> x;
a.push_back(x);

// ucitavamo ostale elemente
for (int i = 1; i < n; i++) {
  cin >> x;
  // element poredimo sa poslednjim u redu i dodajemo ga na pocetak
  // ili na kraj reda
  if (a.back() < x)
    a.push_back(x);
  else
    a.push_front(x);
}

// ispisujemo sve elemente iz reda
for (int x : a)
  cout << x << " ";
cout << endl;
```

4.4 Ред са приоритетом

Ред са приоритетом је врста реда у коме елементи имају на неки начин придружен приоритет, додају се у ред један по један, а увек се из реда уклања онај елемент који има највећи приоритет од свих елемената у реду.

У језику C++ ред са приоритетом се реализује класом `priority_queue<T>`, где је `T` тип елемената у реду. Ред са приоритетом подржава следеће методе:

- `push` - додаје дати елемент у ред
- `pop` - уклања елемент са највећим приоритетом из реда (под претпоставком да ред није празан). Нагласимо да је ова метода типа `void` и да не враћа уклоњени елемент.
- `top` - очитава елемент са највећим приоритетом (под претпоставком да ред није празан)
- `empty` - проверава да ли је ред празан
- `size` - враћа број елемената у реду

Операције `push` и `pop` су сложености $O(\log k)$, где је k број елемената у реду, док су остале операције сложености $O(1)$.

Задатак: Сортирање бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Сортирање уз помоћ реда са приоритетом

Сортирање бројева се може извршити коришћењем реда са приоритетом. Користи се алгоритам *сортирања уз помоћ хипа* (енгл. *heap sort*) који је варијација алгоритма сортирања селекцијом (енгл. *selection sort*) у којем се, подсетимо се, у сваком кораку најмањи елемент доводи на почетак низа. Хип је структура података која се најчешће користи за имплементацију реда са приоритетом. Алгоритам хип сорт користи чињеницу да је одређивање и уклањање најмањег елемента из реда са приоритетом прилично ефикасна операција. Стога се сортирање може реализовати тако што се сви елементи уметну у ред са приоритетом (имплементиран помоћу структуре хип), из кога се затим проналази и уклања један по један најмањи елемент.

Анализа сложености. И убацивање елемената у ред са приоритетом и избацивање елемената из реда са приоритетом обично је сложености $O(\log k)$, где је k број елемената у реду са приоритетом. Стога је укупна сложеност овог алгоритма сортирања $O(n \log n)$.

```
// ovo je način da se u C++-u definiše red sa prioritetom u kome su
// elementi poređani u opadajućem redosledu prioriteta (ovde, vrednosti)
priority_queue<int, vector<int>, greater<int>> Q;
// učitavamo sve elemente niza i ubacujemo ih u red
int n;
cin >> n;
for (int i = 0; i < n; i++) {
  int ai;
  cin >> ai;
  Q.push(ai);
}
// vadimo jedan po jedan element iz reda i ispisujemo ga
while (!Q.empty()) {
  cout << Q.top() << endl;
  Q.pop();
}
```

Види другачија решења овог задатка.

Задатак: Збир k најбољих

Ученик је радио n задатака и за сваки задатак је добио одређени број поена. Одредити збир поена на k задатака које је најбоље урадио.

Улаз: У првој линији стандардног улаза унети природан број n ($1 \leq n \leq 10^6$) - број задатака које је ученик радио, у другој природан број k ($1 \leq k \leq n$) - број задатака које је најбоље урадио, а затим у следећих n линија број поена које је добио на задацима.

Излаз: Укупан број поена које је освојио на k најбоље оцењених задатака.

Пример

Улаз	*Излаз*
10 3 15 80 25 60 10 20 50 45 40 30	190

Решење

Модификовани алгоритам сортирања селекцијом

Једна могућа идеја која избегава сортирање елемената који су мањи од првих k је да се сортирање врши алгоритмом селекције (види задатак Сортирање бројева) који, подсетимо се, у сваком кораку на текуће место у низу доводи најмањи од преосталих елемената низа (у првом кораку се на прво место доводи највећи (или најмањи) елемент целог низа, у другом кораку се на друго место доводи највећи од свих елемената низа осим оног постављеног на прво место итд.). Приметимо да ће се након k корака на почетку низа наћи тачно k највећих елемената и ту се алгоритам може зауставити.

Анализа сложености. Пошто је за налажење најмањег елемента у низу потребно $O(n)$ операција, и то се ради k пута, временска сложеност овог алгоритма је $O(n \cdot k)$, док је меморијска сложеност $O(n)$.

```
// ucitavamo ulazne podatke
int n, k;
cin >> n >> k;
vector<int> a(n);
for (int i = 0; i < n; i++)
  cin >> a[i];

// sortiramo niz primenjujuci k rundi algoritam sortiranja
// selekcijom
for (int i = 0; i < k; i++) {
  int minPoz = i;
  for (int j = i + 1; j < n; j++)
    if (a[j] > a[minPoz])
      minPoz = j;
  swap(a[i], a[minPoz]);
}

// izracunavamo i ispisujemo zbir elemenata niza
int s = 0;
for (int i = 0; i < k; i++)
  s += a[i];
cout << s << endl;
```

Уметање

Једно решење је да се k највећих елемената низа чувају у нерастуће сортираном низу и да се примењује техника сортирања уметањем (види задатак Сортирање бројева). Ради једноставности имплементације чуваћемо низ од $k + 1$ елемената. Када учитамо сваки следећи број поена, стављаћемо га иза последњег постављеног елемента у том низу док се низ не попуни, односно на последњу позицију (ону са индексом k), и затим применом алгоритма уметања померати елемент улево док не дође на своју позицију у том низу (ако је тај елемент није међу највећих k у до сада учитаним елементма, он ће остати на позицији $k + 1$ и у следећем учитавању бити замењен). Уметање можемо реализовати тако што у привремену променљиву учитамо тај последњи елемент низа, затим све елементе који су мањи од њега померимо за једно место удесно и на крају њега упишемо на место елемента који је последњи померен. На крају, сабирамо елементе на првих k позиција тог низа ($k + 1$-ви елемент на позицији k није релевантан).

Анализа сложености. Пошто се у сваком кораку елементи могу померати k пута, а имамо укупно n корака и временска сложеност овог алгоритма је лоша и износи $O(n \cdot k)$, при чему је просторна сложеност боља и износи $O(k)$.

```
int n, k;
cin >> n >> k;

vector<int> a(k + 1);
cin >> a[0];
for (int i = 1; i < n; i++) {
  int x;
  cin >> x;
  // umece x na odgovarajuce mesto u prvih min(i, k) nerastuce
```

```
  // uredjenih elemenata tako da je niz i dalje u nerastucem poretku
  int j;
  for (j = min(i, k) - 1; j >= 0 && x > a[j]; j--)
    a[j + 1] = a[j];
  a[j + 1] = x;
}

// izracunavamo i ispisujemo zbir elemenata niza
int s = 0;
for (int i = 0; i < k; i++)
  s += a[i];

cout << s << endl;
```

Ред са приоритетом

Највећих k до сада виђених елемената низа можемо чувати у структури података која нам омогућава да пронађемо најмањи елемент у њој и да га евентуално заменимо оним који је тренутно учитан (ако је тренутно учитани елемент већи од њега). Идеална структура за то је хип тј. ред са приоритетом.

Ред са приоритетом у језику C++ можемо добити помоћу `priority_queue`. Елементе у ред можемо убацити методом `push`. Елемент који је најмањи можемо очитати методом `top` и избацити методом `pop`.

На почетку ред попуњавамо са k првих учитаних елемената, а затим сваки наредни учитани елемент поредимо са најмањим у реду и ако је већи од њега, најмањи избацујемо, а учитани елемент убацујемо.

Анализа сложености. Пошто је сложеност метода за убацивање и избацивање из реда са приоритетом логаритамска, а методе за очитавање најмањег елемента константна, временска сложеност овог алгоритма је $O(n \cdot \log(k))$, док је просторна сложеност $O(k)$.

Напомена. Приметимо да је овај алгоритам донекле сличан алгоритму сортирања уз помоћ хипа тј. алгоритма Хип-сорт (HeapSort).

```
int n, k;
cin >> n >> k;

// red sa prioritetom koji cuva k najvecih elemenata koristi se
// min-hip, koji omogucava brzo uklanjanje najmanjeg elementa
priority_queue<int, vector<int>, greater<int>> pq;

// ucitavamo prvih k elemenata i ubacujemo ih u red
for (int i = 0; i < k; i++) {
  int x;
  cin >> x;
  pq.push(x);
}

// ucitavamo preostale elemente
for (int i = k; i < n; i++) {
  int x;
  cin >> x;
  // ako je ucitani element veci od najmanjeg trenutno u redu
  // izbacujemo taj najmanji i menjamo ga ucitanim
  if (x > pq.top()) {
    pq.pop();
    pq.push(x);
  }
}

// izbacujemo elemente iz reda racunajuci njihov zbir i ispisujemo ga
int s = 0;
while (!pq.empty()) {
```

```
  s += pq.top();
  pq.pop();
}
cout << s << endl;
```

Види другачија решења овог задатка.

Задатак: K-ти највећи збир пара елемената два низа

Дата су два низа која садрже природне бројеве. Напиши програм који одређује k-ти највећи збир који се може добити када се сабере један елемент првог и један елемент другог низа.

Улаз: Са стандардног улаза се учитава број m ($1 \leq m \leq 5000$), а затим из наредног реда m елемената првог низа раздвојених размаком. Из наредног реда се учитава број n ($1 \leq n \leq 5000$), а затим из наредног реда n елемената другог низа раздвојених размаком. Елементи оба низа су природни бројеви између 0 и 10^6. На крају се учитава број k ($0 \leq k < mn$).

Излаз: На стандардни излаз исписати збир који се налази на позицији k у низу који би се добио када би се низ свих збирова парова једног елемента првог и једног елемента другог низа сортирао нерастуће (позиције се броје од нуле).

Пример 1

Улаз	*Излаз*
3 1 5 3 3 6 4 2 4	7

Објашњење

Збирови који се могу добити, поређани од највећег ка најмањег су $5+6=11$, $5+4=9$, $3+6=9$, $5+2=7$, $3+4=7$, $1+6=7$, $3+2=5$, $1+4=5$, $1+2=3$, па се на позицији 4 налази збир 7.

Пример 2

Улаз

```
5
5 3 8 6 1
6
1 10 9 7 12 2
9
```

Излаз

```
15
```

Решење

Груба сила

Решење грубом силом подразумева да се формира низ свих збирова, да се сортира нерастуће и да се прочита елемент са позиције k.

Анализа сложености. Парова елемената има $m \cdot n$, па је меморијска сложеност квадратна и износи $O(mn)$. Временском сложеношћу доминира сортирање и она износи $O(mn \log (mn))$.

Уместо сортирања, могуће је применити и мало ефикаснији алгоритам *QuickSelect*, који проналази елемент на позицији k и без сортирања низа, но тиме решење не би значајно било унапређено. Опис тог алгоритма дат је у задатку Збир k најбољих.

```
// formiramo sve zbirove parova elemenata dva niza
vector<int> zbirovi;
zbirovi.reserve(m*n);
for (int i = 0; i < m; i++)
```

```
  for (int j = 0; j < n; j++)
    zbirovi.push_back(a[i] + b[j]);

// sortiramo niz zbirova nerastuce
sort(begin(zbirovi), end(zbirovi), greater<int>());

// ispisujemo k-ti element sortranog niza zbirova
cout << zbirovi[k] << endl;
```

Обједињавање сортираних низова

Проблем се може свести на проблем обједињавања неколико сортираних низова, који се не морају истовремено чувати у меморији. Наиме, ако сортирамо оба низа опадајуће, можемо разматрати следеће низове:

$$\begin{gathered} a_0 + b_0, a_0 + b_1, \ldots, a_0 + b_{n-1}, \\ a_1 + b_0, a_1 + b_1, \ldots, a_1 + b_{n-1}, \\ \ldots \\ a_{m-1} + b_0, a_{m-1} + b_1, \ldots, a_{m-1} + b_{n-1}. \end{gathered}$$

Сви они су сортирани нерастуће и могу се објединити коришћењем реда са приоритетом. У почетку у ред убацујемо први елемент сваке листе тј. све збирове облика $a_i + b_0$. У сваком кораку избацујемо највећи збир из реда и додајемо у ред наредни елемент листе којој он припада. Да бисмо након вађења елемента из реда могли додати наредни елемент листе којој он припада, поред вредности збира $a_i + a_j$ (на основу којих је ред уређен) у реду морамо чувати и индексе i и j. Заправо довољно је чувати само индекс j, јер се на основу збира $z = a_i + b_j$, збир $a_i + b_{j+1}$ може добити као $z - b_j + b_{j+1}$.

Анализа сложености. У реду се у сваком тренутку налази највише m елемената. Пошто се чувају и оригинални низови (да би се сортирали), меморијска сложеност је $O(m + n)$. Ажурирање реда врши се k пута. Пошто је сложеност иницијалних сортирања низова $O(m \log m + n \log n)$, сложеност једног ажурирања реда је $O(\log m)$, а важи $k < mn$, временска сложеност је $O(m \log m + n \log n + mn \log m)$. Сложеношћу јасно доминира фаза обједињавања, па се временска сложеност може проценити и само са $O(mn \log m)$.

```
// sortiramo nizove opadajuce
sort(begin(a), end(a), greater<int>());
sort(begin(b), end(b), greater<int>());

// objedinjavamo sortirane nizove
// a[i] + b[0], ..., a[i] + b[n-1], za svako i od 0 do m-1
// u redu cuvamo zbirove a[i] + b[j] i pozicije j
priority_queue<pair<int, int>> pq;

// dodajemo u red prvi element svakog niza
for (int i = 0; i < m; i++)
  pq.emplace(a[i] + b[0], 0);

// k puta azuriramo red
for (int K = 0; K < k; K++) {
  // skidamo element sa vrha reda
  int j, z;
  tie(z, j) = pq.top(); pq.pop();
  // ako lista u kojoj je bio skinuti elemnt nije ispraznjena,
  // u red dodajemo njen naredni element
  if (j + 1 < n)
    pq.emplace(z - b[j] + b[j+1], j+1);
}

// element na poziciji k je trenutno na pocetku reda
cout << pq.top().first << endl;
```

Задатак: Ажурирање медијане

У заводу за статистику желе да што непристрасније процене која је просечна плата. Закључили су да израчунавање аритметичке средине може дати мало искривљену слику јер неколико људи са веома високим платама могу значајно повећати просек. Зато су одлучили да уместо аритметичке средине израчунају медијану, која се добија тако што се све плате поређају у неопадајући низ и онда се узме средишњи елемент тог низа. Ако у низу има паран број елемената, онда се за медијану узима аритметичка средина два средишња елемента. На пример, медијана низа бројева 1, 2, 4, 7, 9 је 4 (јер је он средишњи), а низа бројева 1, 2, 4, 5, 7, 9 је 4.5 (јер је то аритметичка средина бројева 4 и 5 који су средишњи елементи). Подаци о платама пристижу у завод, а софтвер мора да може да у сваком тренутку да податак о медијани до тада унетих плата.

Улаз: Са стандардног улаза се уносе линије, све до краја стандардног улаза. Линија или садржи слово `d` и затим износ плате раздвојен размаком (цео број), што значи да се уноси податак о новој плати или садржи слово `m` што значи да је потребно на стандардни излаз исписати податак о медијани до тада унетих плата. Прва линија сигурно садржи `d`.

Излаз: На стандардном излазу су исписане тражене медијане, свака у посебном реду, заокружене на једну децималу.

Пример

Улаз	*Излаз*
`d 5`	`6.0`
`d 7`	`6.5`
`d 6`	
`m`	
`d 8`	
`m`	

Решење

Два хипа

Ефикасно решење се може добити ако у сваком тренутку у једној (рећи ћемо левој) колекцији чувамо све елементе који су мањи од средишњег, а у другој (рећи ћемо десној) све оне који су већи или једнаки средишњем (ако постоји паран број елемената, те две колекције треба да садрже исти број елемената, а ако постоји непаран број елемената, десна колекција може да садржи један елемент више). Ако има непаран број елемената, тада је медијана једнака најмањем елементу десне колекције, а у супротном је једнака аритметичкој средини између највећег елемента леве и најмањег елемента десне колекције. Сваки нови елемент се пореди са најмањим елементом десне колекције и ако је мањи или једнак њему убацује се у леву колекцију, а ако је већи од њега, убацује се у десну колекцију. Тада се проверава да ли се средина променила. Ако се десило да лева колекција има више елемената од десне (што не допуштамо), највећи елемент леве колекције треба да пребацимо у десну. Ако се десило да у десној колекцији има два елемента више него у левој, тада најмањи елемент десне колекције пребацујемо у леву. Дакле, лева колекција треба да буде таква да лако можемо да пронађемо и избацимо њен највећи елемент, а десна да буде таква да лако можемо да пронађемо и избацимо њен најмањи елемент, при чему обе колекције морају да подрже ефикасно убацивање произвољних елемената. Јасно је да те колекције треба да буду хипови (у језику C++ можемо употребити `priority_queue`) у којима се најмања тј. највећа вредност може очитати у константном времену, уклонити у логаритамском, исто колико је потребно и да се уметне нови елемент.

Пример. Прикажимо рад овог алгоритма на једном примеру.

- На почетку су оба хипа празна.
- Претпоставимо да се на улазу појављује елемент 5. Њега убацујемо у десни хип (јер он може да садржи један елемент више). Стање је сада следеће:

  $levi : [] \quad desni : [5]$

  Медијана је елемент на врху десног хипа тј. 5.
- Претпоставимо са се на улазу сада појављује елемент 6. Пошто је он већи од елемента 5 који је на врху десног хипа, елемент 5 пребацујемо у леви, а елемент 6 убацујемо у десни хип. Стање је сада следеће:

  $levi : [5] \quad desni : [6]$

Медијана је просек елемената на врху оба хипа тј. 5,5.

- Претпоставимо да се сада на улазу појављује елемент 3. Пошто је он мањи од елемента 5 на врху левог хипа елемент 5 пребацујемо у десни хип, а елемент 3 убацујемо у леви. Стање је сада следеће:

  $levi : [3] \quad desni : [5, 6]$

  Медијана је елемент на врху десног хипа тј. 5.

- Претпоставимо да се сада на улазу појављује елемент 1. Пошто је он мањи од елемента 5 на врху десног хипа, додајемо га у левих хип.

  Стање је сада следеће:

  $levi : [1, 3] \quad desni : [5, 6]$

  Медијана је просек елемената на врху оба хипа тј. 4.

- Претпоставимо да се сада на улазу појављује елемент 8. Пошто је он већи од елемента 5 на врху десног хипа, додајемо га у десни хип.

  Стање је сада следеће:

  $levi : [1, 3] \quad desni : [5, 6, 8]$

  Медијана је елеменат на врху десног хипа, тј. 5.

Напомена. Идеја да се уместо у једне подаци чувају у две структуре података често може довести до ефикасног решења. На пример, уместо повезане листе у коју се елементи убацију на тренутну позицију итератора који се помера по листи, могу се користити два стека, као што је приказано у задатку Линијски едитор.

```
priority_queue<int, vector<int>, greater<int>> veci_od_sredine;
priority_queue<int, vector<int>, less<int>> manji_od_sredine;

double medijana() {
  if (manji_od_sredine.size() == veci_od_sredine.size())
    return (manji_od_sredine.top() + veci_od_sredine.top()) / 2.0;
  else
    return veci_od_sredine.top();
}

void dodaj(int x) {
  if (veci_od_sredine.empty())
    veci_od_sredine.push(x);
  else {
    if (x <= veci_od_sredine.top())
      manji_od_sredine.push(x);
    else
      veci_od_sredine.push(x);
    if (manji_od_sredine.size() > veci_od_sredine.size()) {
      veci_od_sredine.push(manji_od_sredine.top());
      manji_od_sredine.pop();
    } else if (veci_od_sredine.size() > manji_od_sredine.size() + 1) {
      manji_od_sredine.push(veci_od_sredine.top());
      veci_od_sredine.pop();
    }
  }
}
```

4.5 Апстрактне структуре података

Структуре података карактерише њихов апстрактни интерфејс тј. операције које треба да подрже (и допуштена сложеност тих операција). У наредних неколико примера биће приказано како се могу реализовати неке

структуре на основу задатог интерфејса. У многим случајевима довољно је на инвентиван начин употребити неку библиотечку структуру података или искомбиновати неколико библиотечких структура података.

Задатак: Мапа са увећањем

Мапа тј. речник је структура података која пресликава задате целобројне кључеве у задате целобројне вредности. Допуштене су следеће операције над мапом.

- `r k` — *read* — испис вредности придружене кључу `k` на стандардни излаз (ако том кључу није додељена вредност, потребно је исписати 0).
- `w k x` — *write* — кључу `k` придружује се вредност `x` (ако је раније била придружена нека друга вредност, она се занемарује).
- `e k` — *erase* — брише се вредност придружена кључу `k`.
- `i k x` — *increment* — вредност придружена кључу `k` се увећава за `x` (ако кључу `k` раније није била придружена вредност, њему се додељује вредност `x`).
- `a x` — *increment all* — све вредности које су придружене кључевима се увећавају за `x`.

Написати програм који чита низ овако описаних операција и спроводи их.

Улаз: Са стандардног улаза се учитавају операције, свака у посебном реду, њих највише 10^5.

Излаз: На стандардни излаз исписати резултат извршавања учитаних операција (оно што се исписује операцијама `r k`).

Пример

Улаз	*Излаз*
`w 0 2`	`2`
`w 1 1`	`3`
`r 0`	`5`
`i 1 2`	`6`
`r 1`	`0`
`a 3`	
`w 2 0`	
`r 0`	
`r 1`	
`r 2`	

Решење

Груба сила

Основу решења представља коришћење библиотечког асоцијативног низа (мапе тј. речника). Једина операција која није директно подржана мапама је увећање свих елемената за дату вредност. Ако се она имплементира грубом силом, потребан је обилазак свих кључева у мапи, што је прилично неефикасно.

Анализа сложености. Сложеност операције увећања свих вредности је линеарна у односу на број кључева у мапи, па је укупна сложеност квадратна у односу на број операција (најгори случај може бити када прва половина наредби упише све различите кључеве у мапу, а друга половина наредби буде увећање свих вредности).

```
map<int, int> m;

int vrednost(int k) {
   auto it = m.find(k);
   return it == m.end() ? 0 : it->second;
}

void umetni(int k, int v) {
   m[k] = v;
}

void obrisi(int k) {
   m.erase(k);
```

```
}

void uvecaj(int k, int v) {
   m[k] += v;
}

void uvecajSve(int v) {
  for (auto& p : m)
    p.second += v;
}

int main() {
  ios_base::sync_with_stdio(0); cin.tie(0);
  char c;
  while (cin >> c) {
    if (c == 'r') {
      int k;
      cin >> k >> ws;
      cout << vrednost(k) << '\n';
    } else if (c == 'w') {
      int k, x;
      cin >> k >> x >> ws;
      umetni(k, x);
    } else if (c == 'e') {
      int k;
      cin >> k >> ws;
      obrisi(k);
    } else if (c == 'i') {
      int k, x;
      cin >> k >> x >> ws;
      uvecaj(k, x);
    } else if (c == 'a') {
      int x;
      cin >> x >> ws;
      uvecajSve(x);
    }
  }
  return 0;
}
```

Чување збира свих увећања

Да би се та операција могла ефикасније имплементирати уз мапу чувамо и засебан цео број који представља збир свих увећања. Уместо да повећавамо све вредности у мапи, повећаваћемо само тај број. Пре исписа вредности придружене кључу увећаћемо је за тај број, а при придруживању вредности кључу умањићемо је за тај број.

Анализа сложености. У зависности од тога да ли се користи уређени или неуређени асоцијативни низ, сложеност свих операција је или $O(\log n)$ или амортизована константна. Укупна сложеност је зато у најгорем случају квазилинеарна у односу на број наредби.

```
map<int, int> m;
int uvecanje = 0;

int vrednost(int k) {
   auto it = m.find(k);
   return it == m.end() ? 0 : it->second + uvecanje;
}

void umetni(int k, int v) {
```

```
   m[k] = v - uvecanje;
}

void obrisi(int k) {
   m.erase(k);
}

void uvecaj(int k, int v) {
   auto it = m.find(k);
   if (it == m.end())
     m[k] = v - uvecanje;
   else
     it->second += v;
}

void uvecajSve(int v) {
   uvecanje += v;
}
```

Задатак: Ред без дупликата

Програм извршава следеће операције над одређеном колекцијом података.

- `i x` — *insert* — ако колекција већ не садржи елемент `x` он се у њу додаје.
- `r` — *remove* — из колекције се уклања и на стандардни излаз се исписује први елемент који је у њу додат (као код класичног реда). Ако је колекција празна, исписује се `-`.

Напиши програм који учитава низ наредби и извршава их.

Улаз: Са стандардног улаза се учитава низ наредби, свака је у посебном реду.

Излаз: На стандардни излаз исписати резултат извршавања програма.

Пример

Улаз	*Излаз*
`i 1`	`1`
`i 2`	`2`
`i 1`	`3`
`i 3`	`4`
`r`	`-`
`r`	
`i 4`	
`r`	
`r`	
`r`	

Решење

Груба сила - линеарна претрага

У имплементацији можемо употребити библиотечку имплементацију структуре ред. Пре додавања елемента на крај, морамо проверити да ли елемент већ постоји у реду. У језику C++ можемо употребити структуру података `deque` и функцију `find` која примењује линеарну претрагу (њу није могуће употребити када се користи `queue`).

Анализа сложености. Због линеарне претраге реда пре додавања новог елемента, сложеност додавања је линеарна у односу на тренутни број елемената у реду, па је укупна сложеност алгоритма квадратна (најгори случај може бити када се у свим наредбама уноси нови елемент у ред, различит од свих претходних).

```
ios_base::sync_with_stdio(false); cin.tie(0);
deque<int> q;

char c;
```

```
while (cin >> c) {
  if (c == 'i') {
    int x;
    cin >> x >> ws;
    if (find(q.begin(), q.end(), x) == q.end()) {
      q.push_back(x);
    }
  } else if (c == 'r') {
    if (!q.empty()) {
      cout << q.front() << '\n';
      q.pop_front();
    } else {
      cout << "-" << '\n';
    }
  }
}
```

Чување додатног скупа

Ефикасна имплементација је могућа ако се уз ред чува додатни скуп елемената који су садржани у реду. Пре додавања елемента у ред, проверава се да ли тај елемент постоји у скупу. Ако не постоји, додаје се и у скуп и у ред. Приликом избацивања елемената, први елемент реда се избацује и из реда и из скупа.

Анализа сложености. У зависности од тога да ли се користи уређен или неуређен скуп, сложеност операција додавања, брисања и претраге елемената у скупу може бити логаритамска или амортизована константнтна. Додавање у ред и читање елемената из реда има константну сложеност. Зато је укупна сложеност алгоритма линеарна ако се користи неуређен скуп тј. квазилинеарна ако се користи уређен скуп.

```
  return 0;
}
```

Задатак: Фреквенцијски стек

Програм извршава две врсте операција са структуром података која донекле подсећа на стек.

- Операција `0 x` поставља елемент `x` на врх стека.
- Операција `1` уклања елемент који се најчешће појављује од свих елемената који су тренутно на стеку. Ако је више таквих елемената, уклања се онај који је последњи додат. Нпр. ако су на стеку елементи `1 2 2 1 3`, уклања се елемент `1` и стек постаје `1 2 2 3`.

Улаз: Са стандардног улаза се учитава број операција n ($1 \leq n \leq 10^5$), а затим n операција (свака у посебном реду).

Излаз: Опис излазних података.

Пример

Улаз	*Излаз*
12 0 1 0 2 0 2 0 1 0 3 0 2 1 1 1 1 1 1	2 1 2 3 2 1

Решење

Основна идеја решења је да се уместо једниственог стека елементи чувају на више стекова организованих по фреквенцијама тј. броју појављивања елемента. Прво појављивање неког елемента стављаћемо на стек првих појављивања, друго појављивање на стек других појављивања итд. Да бисмо знали на који стек треба ставити наредно појављивање неког елемента, помоћу асоцијативног низа (речника, мапе) одржаваћемо број појављивања сваког елемента. Одржаваћемо и максималну фреквенцију. Ако се додавањем неког елемента повећава максимална фреквенција, додајемо нови стек. Приликом уклањања елемента, уклањаћемо врх последњег стека, тј. стека који одговара тој највећој фреквенцији. Ако се након тога тај стек испразни, смањујемо максималну фреквенцију.

Пример. Прикажимо рад алгоритма на примеру додавања бројева `1 2 2 1 3 2`.

- Прво се додаје број `1` на стек елемената који се појављују само једном. Максимална фреквенција постаје 1.

  ```
  1: 1
  ```

- Након тога се додаје број `2` на стек елемената који се појављују само једном.

  ```
  1: 1, 2
  ```

- Затим се додаје број `2` на стек елемената који се појављују два пута. Максимална фреквенција постаје 2.

  ```
  1: 1, 2
  2: 2
  ```

- Затим се додаје број `1` на стек елемената који се појављују два пута.

  ```
  1: 1, 2
  2: 2, 1
  ```

- Затим се додаје број `3` на стек елемената који се појављују једном.

  ```
  1: 1, 2, 3
  2: 2, 1
  ```

- На крају се додаје број `2` на стек елемената који се појављују три пута. Максимална фреквенција постаје 3.

  ```
  1: 1, 2, 3
  2: 2, 1
  3: 2
  ```

Прикажимо сада уклањање елемената док се стек не испразни.

- Пошто је максимална фреквенција 3, уклања се врх стека елемената са фреквенцијом 3 и исписује се број 2. Пошто се тај стек испразнио максимална фреквенција се смањује на 2.

  ```
  1: 1, 2, 3
  2: 2, 1
  ```

- Пошто је максимална фреквенција 2, уклања се врх стека елемената са фреквенцијом 2 и исписује се број 1.

  ```
  1: 1, 2, 3
  2: 2
  ```

- Пошто је максимална фреквенција 2, уклања се врх стека елемената са фреквенцијом 2 и исписује се број 2. Пошто се тај стек испразнио максимална фреквенција се смањује на 1.

  ```
  1: 1, 2, 3
  ```

- Након тога се редом скидају и исписује елементи 3, 2 и на крају 1 (тада се максимална фреквенција смањује на нулу).

```
ios_base::sync_with_stdio(false); cin.tie(0);
// frekvencija tj. broj pojavljivanja svakog elementa
map<int, int> frekv;
// niz stekova - na steku broj i čuvaju se i-ta pojavljivanja svih
```

```
// elemenata u originalnom steku
map<int, stack<int>> stekovi;
// najveca frekvencija nekog elementa u originalnom steku
int maksFrekv = 0;

// obradjujemo n naredbi
int n;
cin >> n;
for (int i = 0; i < n; i++) {
  int naredba;
  cin >> naredba;
  if (naredba == 0) {
    // ucitavamo element
    int x;
    cin >> x;
    // uvecavamo njegov broj pojavljivanja
    frekv[x]++;
    // u skladu sa tim ga postavljamo na njemu odgovarajuci stek
    stekovi[frekv[x]].push(x);
    // azuriramo najvecu frekvenciju nekog elementa
    maksFrekv = max(maksFrekv, frekv[x]);
  } else if (naredba == 1) {
    // uklanjamo element sa vrha steka elemenata koji se pojavljuju
    // najcesce
    int x = stekovi[maksFrekv].top();
    stekovi[maksFrekv].pop();
    cout << x << '\n';
    // smanjujemo njegov broj pojavljivanja
    frekv[x]--;
    // azuriramo najvecu frekvenciju nekog elementa
    if (stekovi[maksFrekv].empty())
      maksFrekv--;
  }
}
```

4.6 Имплементација структура података

Као што смо већ видели, језик C++ пружа подршку за велики број структура података, било примитивних, подржаних кроз сам језик (на пример, кориснички дефинисане структуре и уније и статички и динамички алоцирани низови), било подржаних кроз стандардну библиотеку (STL).

Подсетимо се, по начину како су имплементиране, библиотечке структуре података се могу груписати на следећи начин.

- У групу секвенцијалних структура података (контејнера) спадају `array`, `vector`, `string`, `list`, `forward_list` и `deque`.
- У групу адаптора контејнера спадају `stack`, `queue` и `priority_queue`.
- У групу уређених асоцијативних контејнера спадају `set`, `multiset`, `map` и `multimap`.
- У групу неуређених асоцијативних контејнера спадају `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`.

Секвенцијални контејнери се користе за складиштење серија елемената и представљају одређена уопштења класичних низова (додајући могућности динамичког проширивања, додавања и уклањања елемената и слично). Сваки секвенцијални контејнер има своју специфичну имплементацију која одређује сложеност операција и самим тим адекватне сценарије употребе.

- `array<T, size>` представља само танак **омотач око класичног статичког низа** чија је димензија `size` позната у тренутку декларације (не може се употребити промењива) и служи само да омогући

да се класични статички низови могу користити на исти начин као и други контејнери (на пример, допуштена је директна додела низа низу, поређење два низа оператором == и слично, што није могуће ако се користе примитивни статички низови).

- `vector<T>` је имплементиран преко **динамичког низа** који се по потреби реалоцира. Таква имплементација омогућава ефикасан индексни приступ и итерацију у оба смера, као и додавање елемената на крај (`push_back`) и скидање елемената са краја (`pop_back`). Пошто услед реалокација динамичког низа долази до честог копирања великог броја елемената, вектори који се током рада програма проширују, показују лоше перформансе када се у њима чувају велики објекти. `string` је имплементиран слично као вектор (уз додатне оптимизације и операције специфичне за податке текстуалног типа).
- `list<T>` је имплементиран преко **двоструко повезане листе**, док је `forward_list<T>` имплементиран преко **једноструко повезане листе**. Захваљујући таквој имплементацији, основна предност ових контејнера у односу на `vector` и `deque` је то што подржавају уметање и брисање елемената са било које позиције (не само на почетак или на крај) у константној сложености. Са друге стране, губи се могућност ефикасног индексног приступа. Двоструко имплементирана листа омогућава итерацију у оба смера, а једноструко повезана листа само унапред (над итераторима није могуће примењивати оператор `--` нити одузимање бројевних вредности од итератора). Пошто за разлику од вектора не долази до реалокације, у листама се могу складиштити и велики објекти.
- `deque<T>` је имплементиран као специфична структура података коју ћемо називати **дек** и која се може замислити као вектор у коме се налазе показивачи на блокове (низове) елемената фиксне величине (нешто попут `vector<array<T, size>*>`). Оваква специфична имплементација омогућава ефикасно додавање и уклањање елемената са оба краја (отуда и назив `deque` тј. double ended queue), али и ефикасну итерацију кроз све елементе, у оба смера, као и ефикасан индексни приступ. Једна од основних предности у односу на `vector` је то што се због чувања показивача приликом реалокације не копирају елементи већ само показивачи на њих, тако да се у дек који се током рада програма проширује могу слободно чувати и велики објекти.

Адаптори контејнера само представљају слој изнад неког од постојећих секвенцијалних контејнера и пружају апстрактни интерфејс изнад имплементације секвенцијалног контејнера, имплементирајући функције тог интерфејса коришћењем секвенцијалног контејнера за складиштење података. Адаптори имају свој подразумевани секвенцијални контејнер, који се може променити приликом декларисања променљивих.

- `stack` имплементира функције стека коришћењем вектора за складиштење података. Тип `stack<int>` подразумева заправо `stack<int, vector<int>>`, а могуће је користити и, на пример, `stack<int, forward_list<int>>` у ком се за имплементацију стека користи једноструко повезана листа.
- `queue` имплементира функције реда коришћењем дека за складиштење података. На пример, тип `queue<int>` подразумева заправо `queue<int, deque<int>>`. Уместо дека могуће је употребити и неки други контејнер (пре свега `list<int>`, док би `vector<int>` и `forward_list<int>` имали веома лошу сложеност, па их нема смисла користити).
- `priority_queue` имплементира функције реда са приоритетом коришћењем вектора за складиштење података. Тип `priority_queue<int>` заправо представља `priority_queue<int, vector<int>, less<int>>` где је `less<int>` функција која се користи за поређење елемената и прозрокује уређеност по опадајућем редоследу приоритета (на врху је елемент са највећим приоритетом). У вектору су смештени елементи специјалног дрвета који се назива *хип* и које ће бити објашњено касније.

Уређени асоцијативни контејнери су имплементирани помоћу **самобалансирајућих уређених бинарних дрвета** (обично су то **црвено-црна дрвета, RBT – red-black tree**).

- `set<T>` је имплементиран помоћу уређеног бинарног дрвета у коме се у чворовима налазе елементи скупа и у ком су у свим чворовима различите вредности.
- `multiset<T>` може бити имплементиран помоћу уређеног бинарног дрвета у коме се у чворовима налазе елементи мултискупа и у ком је могуће да постоји више чворова у коме су различите вредности или помоћу уређеног бинарног дрвета у чијим се чворовима налазе елементи мултискупа уз њихов број појављивања, без понављања елемената скупа.
- `map<K, V>` је имплементиран помоћу уређеног бинарног дрвета на основу кључева у коме се у чворовима налазе подаци о кључевима и њима придруженим вредностима и у ком су у свим чворовима различите вредности кључева.

- `multimap<K, V>` може бити имплементиран помоћу помоћу уређеног бинарног дрвета на основу кључева у коме се у чворовима налазе подаци о кључевима и њима придруженим вредностима и у ком више чворова може имати исту вредност кључа.

Неуређени асоцијативни контејнери су имплементирани помоћу **хеш-табела**. То су

- `unordered_set<T>`
- `unordered_multiset<T>`
- `unordered_map<K, V>`
- `unordered_multimap<K, V>`

4.6.1 Динамички низ - имплементација вектора

Вектори се имплементирају у облику динамичког низа, којем се димензија мења и који се реалоцира кад год број елемената достигне величину алоцираног простора.

Задатак: Вектор

Програм извршава три врсте наредби над низом целих бројева:

- `w p x` – *write* – у низ се на позицију `p` уписује вредност `x`.
- `r p` – *read* – на стандардни излаз се исписује вредност прочитана са позиције `p` (сматра се да се на позицијама на којима није вршен упис налазе нуле).
- `p x` – *push* – на крај низа, на позицију непосредно иза последњег уписаног елемента уписује се вредност `x`.

Написати програм који учивата ове наредбе, извршава их једну по једну и исписује резултат њиховог извршавања (водећи при том рачуна о томе да се наредбе изврше ефикасно).

Улаз: Свака линија стандардног улаза садржи једну наредбу.

Излаз: На стандардни излаз исписати резултат извршавања свих учитаних наредби.

Пример

Улаз	*Излаз*
`p 1`	`1`
`w 2 2`	`2`
`r 0`	`0`
`r 2`	`5`
`r 1`	`2`
`w 0 5`	
`r 0`	
`w 1 4`	
`w 1 2`	
`r 1`	

Решење

Библиотечка имплементација

Задатак се веома једноставно решава коришћењем библиотечке колекције `vector`.

```
ios_base::sync_with_stdio(false);
cin.tie(0);
vector<int> a;
char c;
while (cin >> c) {
  if (c == 'w') {
    int i, x;
    cin >> i >> x >> ws;
    if (i >= a.size())
      a.resize(i+1);
    a[i] = x;
  } else if (c == 'r') {
    int i;
```

```
    cin >> i >> ws;
    cout << (i < a.size() ? a[i] : 0) << '\n';
  } else if (c == 'p') {
    int x;
    cin >> x >> ws;
    a.push_back(x);
  }
}
```

Ручна имплементација

Прикажимо како се динамички низ може имплементирати када се не користи библиотечка подршка.

Основна идеја динамичког низа је то да се у старту алоцира неки претпостављени број елемената и да се, када се установи да тај број елемената није више довољан, изврши реалокација низа, тако што се алоцира нови, већи низ, затим се у тај нови низ ископирају елементи оригиналног низа и на крају се тај стари низ обрише, а показивач преусмери ка новом низу (у језику C++ не постоји оператор који би био еквивалентан C функцији `realloc`).

Имплементација основне функционалности динамичког низа у језику C++ могла би да изгледа како је приказано у коду. Једноставности ради користимо глобалне променљиве и функције – оне се једноставно избегавају, на пример, тако што може да се користи објектнооријентисано програмирање, али то није у фокусу овог курса. Улогу показивача `NULL` из језика C има показивач `nullptr`, улогу функције `malloc` има израз `new`, а улогу функције `free` има израз `delete` (постоје озбиљне разлике између ових C функција и C++ оператора, али ни то нам у овом тренутку није у фокусу). Израз `new T[n]` користимо да бисмо алоцирали низ од `n` елемената типа `T`, при чему број елемената тог типа наводимо у угластим заградама. За разлику од `malloc` која враћа `void*`, изразом `new T[n]` добијамо показивач типа `T*`. Ослобађање низа елемената алоцираног са `new T[n]` вршимо оператором `delete[]`.

Анализа сложености. Ако се динамичка реалокација врши на основу неке геометријске стратегије за повећање величине (нпр. сваки пут повећамо број елемената дупло), додавање на крај се врши у амортизованом константном времену. Индексни приступ елементу (приступ на основу позиције) захтева константно време (исто као код обичног низа).

```
// adresa početka niza
int* a = nullptr;
// broj alociranih i broj popunjenih elemenata
int alocirano = 0, n = 0;

// čitanje vrednosti elementa na datoj poziciji
// ne proverava se pripadnost opsegu niza
int procitajElement(int i) {
  return a[i];
}

// upis vrednosti elementa na datu poziciju
// ne proverava se pripadnost opsegu niza
void postaviElement(int i, int x) {
  a[i] = x;
}

// pomoćna funkcija koja vrši realokaciju niza na dati broj elemenata
void realociraj(int m) {
  // alociramo novi niz
  int* novo_a = new int[m];
  alocirano = m;
  // ako postoji stari niz, kopiramo njegove elemente u novi
  // i brišemo stari niz
  if (a != nullptr) {
    copy_n(a, min(m, n), novo_a);
    delete[] a;
```

```
  }
  // pokazivač usmeravamo ka novom nizu
  a = novo_a;
}

void postaviVelicinu(int nn) {
  if (alocirano <= nn)
    realociraj(2 * nn + 1);
  n = nn;
}

// dodavanje datog elementa na kraj dinamičkog niza
// niz se automatski realocira ako je potrebno
void dodajNaKraj(int x) {
  if (alocirano <= n)
    realociraj(2 * alocirano + 1);
  a[n++] = x;
}

// brisanje celog niza
void obrisi() {
  delete[] a;
}
```

4.6.2 Листе

Структуре листе подразумевају да се подаци чувају у меморији у чворовима којима се поред података чувају показивачи. У зависности од тога да ли се чувају само показивачи на следећи или и на претходни елемент разликују се:

- једноструко повезане листе
- двоструко повезане листе

Под претпоставком да су познати показивачи на почетак и на крај листе, једноструко повезане листе допуштају додавање на почетак и на крај, као и брисање са почетка у времену $O(1)$, док брисање елемента са краја захтева време $O(n)$ (јер је потребно изменити претпоследњи чвор, до кога се може доћи само поновним обиласком ниске од почетка). Уметање и брисање следећег елемента у односу на елемент на који указује познати показивач се може извршити у времену $O(1)$. Ипак, проналажење позиције на коју треба уметнути неки елемент често захтева пролаз кроз листу и захтева време $O(n)$. Уметање елемента тако да претходи елементу на који указује познати показивач, као и брисање елемента на који указује познати показивач захтева проналазак и измену претходног чвора, што захтева нови пролазак кроз листу и време $O(n)$.

Приступ елементу на датој позицији (индексни приступ) захтева време $O(n)$.

Под претпоставком да су познати показивачи на почетак и на крај листе, двоструко повезане листе допуштају и додавање и брисање и са почетка и са краја листе у времену $O(1)$. Уметање и брисање било испред, било иза елемента на који указује познати показивач врши се у времену $O(1)$. Остале операције се извршавају у истом времену као и код једноструко повезаних листа. Мане двоструко повезаних у односу на једноструко повезане листе су то што због чувања показивача на претходне елементе захтевају више меморије и појединачне операције могу бити мало спорије јер се захтева ажурирање више показивача. Предности су то што омогућују ефикасније извршавање неких операција (пре свега брисања са краја и итерација уназад).

Имплементација се често може учинити једноставнијом (па и донекле ефикаснијом) ако се уведе вештачки чвор (“сентинела”) којим се обезбеђује да је листа увек непразна и да показивач који се налази испред стварног почетка или иза стварног краја нема вредност NULL, већ указује на неки чвор листе (у овом случају, на сентинелу), тако да се његов претходни и следећи елемент могу добити на исти начин као и када показивач указује на неки регуларан чвор листе.

Употреба листа је све ређа и ређа на савременим системима. Алокација и деалокација појединачних чворова листе је скупа операција и у реалним имплементацијама потребно је користити неки специфичан механизам управљања меморијом (на пример, већи број допустивих чворова алоцирати одједном, а онда у листу уланчавати један по један од тих чворова). Додатни проблем је то што су чворови често раштркани по меморији, па

су промашаји кеш меморије много чешћи него у случају рада са структурама у којима су подаци у меморији смештени повезано (низовима и структурама у којима се користе већи блокови повезаних бајтова). Такође, због чувања показивача листе захтевају много више меморије него низови.

Предности листа у односу на низове и декове наступају пре свега у ситуацијама у којима се у листама чувају велики подаци. Тада се током реалокације динамичких низова копирају велике количине података, што може бити неефикасно, па је корисније употребити листе код којих се подаци не реалоцирају.

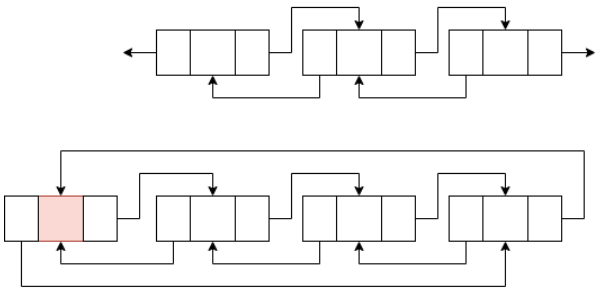
Задатак: Линијски едитор

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Задатак се може решити тако што се ручно имплементира двоструко повезана листа (пошто се курсор може кретати у оба смера, јасно је да листа мора бити двоструко повезана). Сваки чвор двоструко-повезане листе чува вредност (у нашем случају карактер), показивач на претходни и показивач на следећи чвор. Имплементација такве листе је једноставнија ако се уведе специјални чвор (понекад се назива *сентинела*) који истовремено чува позицију почетног и крајњег чвора двоструко повезане листе (његов показивач на следећи чвор указује на стварни почетак, а показивач на претходни чвор указује на стварни крај листе, при чему оба показују на сентинелу ако је листа празна). Вредност уписана у сентинелу се никада не користи. Овим се заправо одржава својеврсна кружна листа.



Слика 4.1: Двоструко повезана листа са NULL показивачима и са сентинелом

Постојање сентинеле значајно олакшава имплементације процедура за уметање и брисања елемената листе јер нема потребе испитивати специјалне случајеве да ли је неки показивач NULL (што је неопходно ако се користи репрезентација у коме су претходник првог и следебеник последњег чвора постављени на вредност NULL).

Иако курсор у едитору увек указује између два карактера, курсор ћемо у имплементацији представити показивачем на елемент листе (карактер) који се налази десно од стварне позиције курсора (карактер десно од курсора). Ако курсор указује на сентинелу, сматраћемо да је курсор на крају текста, а ако показује на први елемент листе (елемент који следи након сентинеле), сматраћемо да је курсор на почетку текста. Операције се тада изводе на следећи начин:

- Курсор можемо померити на лево, само ако му не претходи сентинела, а на десно само ако не указује на сентинелу.
- Уметање карактера се врши тако што се нови елемент убаци тако да претходи тренутној позицији курсора, а курсор се не помера (ово ради исправно и када је листа празна тј. када садржи само сентинелу).
- Брисање карактера лево од курсора је могуће кад год курсор није на почетку листе тј. када му не претходи сентинела и врши се тако што се обрише претходни елемент листе, а курсор се не помера.
- Брисање карактера десно од курсора је могуће када год курсор није на крају листе тј. када не указује на сентинелу и врши се тако што се обрише елемент на који курсор указује, а курсор се помери на следећу позицију.

Анализа сложености. Пошто су код двоструко повезане листе операције уметања и брисања, као и померања итератора за једно место константне сложености, овај алгоритам је линеарне сложености.

```
struct cvor {
  char c;
  cvor *prethodni, *sledeci;
};

cvor* napravi_cvor(char c, cvor* p = nullptr, cvor* s = nullptr) {
  cvor* novi = new cvor();
  // potrebna je provera da li je alokacija uspela
  novi->c = c;
  novi->prethodni = p;
  novi->sledeci = s;
  return novi;
}

// brise cvor na koji ukazuje pokazivac cv (pretpostavljamo da to nije
// sentinela)
void obrisi(cvor* cv) {
  cv->prethodni->sledeci = cv->sledeci;
  cv->sledeci->prethodni = cv->prethodni;
  delete cv;
}

// ubacuje novi cvor sa karakterom c ispred cvora na koji ukazuje
// pokazivac cv
void ubaci_ispred(cvor* cv, char c) {
  cvor* novi = napravi_cvor(c, cv->prethodni, cv);
  novi->prethodni->sledeci = novi;
  novi->sledeci->prethodni = novi;
}

// ispisuje celu listu
void ispisi(cvor* sentinela) {
  for (cvor* cv = sentinela->sledeci; cv != sentinela; cv = cv->sledeci)
    cout << cv->c;
  cout << endl;
}

int main() {
  string naredbe;
  cin >> naredbe;
  int i = 0;

  // vestacki ubacen ”prazan” cvor koji oznacava poziciju ispred
  // pocetka tj. iza kraja liste
  cvor* sentinela = napravi_cvor('_');
  sentinela->prethodni = sentinela->sledeci = sentinela;

  // tekuca pozicija kursora
  cvor* kursor = sentinela;

  while (i < naredbe.size()) {
    char naredba = naredbe[i++];
    if (naredba == '<') {
      if (kursor->prethodni != sentinela)
        kursor = kursor->prethodni;
    } else if (naredba == '>') {
      if (kursor != sentinela)
        kursor = kursor->sledeci;
```

```
    } else if (naredba == 'i') {
      char c = naredbe[i++];
      ubaci_ispred(kursor, c);
    } else if (naredba == 'b') {
      if (kursor->prethodni != sentinela)
        obrisi(kursor->prethodni);
    } else if (naredba == 'd') {
      if (kursor != sentinela) {
        cvor* sl = kursor->sledeci;
        obrisi(kursor);
        kursor = sl;
      }
    }
  }
  ispisi(sentinela);
  obrisi(sentinela);

  return 0;
}
```

4.6.3 Стек

Стек представља колекцију података у коју се подаци додају по LIFO (енгл. last-in-first out) принципу - елемент се може додати само на врх и скинути само са врха стека.

У језику C++ стек се реализује класом `stack<T>` где `T` представља тип елемената на стеку. За њено коришћење потребно је укључити заглавље `<stack>`. Подржане су следеће методе (све сложености $O(1)$):

- `push` - поставља дати елемент на врх стека
- `pop` - скида елемент са врха стека (под претпоставком да стек није празан – ако је стек празан, понашање је недефинисано и може доћи до насилног прекида рада програма). Нагласимо да је ова метода типа `void` и да не враћа уклоњени елемент.
- `top` - очитава елемент на врху стека (под претпоставком да стек није празан)
- `empty` - проверава да ли је стек празан
- `size` - враћа број елемената на стеку.

Ако се на стеку чувају уређени парови или n-торке, тада се уместо методе `push`, може користити метода `emplace`, којој се само редом наводе елементи пара тј. n-торке (није потребно посебно позивати функцију за креирање пара тј. n-торке, што је неопходно када се користи `push`).

Стек у језику C++ је заправо само адаптер око неке колекције података (подразумевано вектора) који корисника тера да поштује правила приступа стеку и спречава да направи операцију која над стеком није допуштена (попут приступа неком елементу испод врха).

Задатак: Историја веб-прегледача

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Пошто је број сајтова ограничен текстом задатка, стек се може имплементирати коришћењем статичког низа.

```
// istorija pregledaca - rucno implementiran stek koji moze da cuva
// najvise 1000 adresa (to je dato u tekstu zadatka)
string istorija[1000];
// pokazivac na naredno slobodno mesto na steku
int sp = 0;

string linija;
while (getline(cin, linija)) {
  if (linija == "back") {
```

```
    if (sp > 0)
      sp--;
    if (sp > 0)
      cout << istorija[sp-1] << endl;
    else
      cout << "-" << endl;
  }
  else {
    cout << linija << endl;
    istorija[sp++] = linija;
  }
}
```

Стек може бити имплементиран коришћењем једноструко повезане листе.

```
// cvor jednostruko povezane liste
struct cvor {
  string s;
  cvor* sledeci;
};

cvor* napravi_cvor(const string& s, cvor* sledeci) {
  cvor* novi = new cvor();
  // trebalo bi proveriti da li je alokacija uspela
  novi->s = s;
  novi->sledeci = sledeci;
  return novi;
}

// dodavanje elementa na pocetak
cvor* dodaj(cvor* pocetak, const string& s) {
  return napravi_cvor(s, pocetak);
}

// brisanje elementa sa pocetka
cvor* obrisi(cvor* pocetak) {
  cvor* sledeci = pocetak->sledeci;
  delete pocetak;
  return sledeci;
}

// brisanje cele liste
void obrisi_listu(cvor* pocetak) {
  while (pocetak != nullptr) {
    cvor* sledeci = pocetak->sledeci;
    delete pocetak;
    pocetak = sledeci;
  }
}

int main() {
  // istorija pregledaca - stek implementiran preko liste
  cvor* pocetak = nullptr;

  string linija;
  while (getline(cin, linija)) {
    if (linija == "back") {
      if (pocetak != nullptr)
        pocetak = obrisi(pocetak);
```

```
      if (pocetak != nullptr)
        cout << pocetak->s << endl;
      else
        cout << "-" << endl;
    }
    else {
      cout << linija << endl;
      pocetak = dodaj(pocetak, linija);
    }
  }

  obrisi_listu(pocetak);
  return 0;
}
```

4.6.4 Ред

Ред представља колекцију података у коју се подаци додају по FIFO (енгл. first-in-first-out) принципу – елемент се увек узима са почетка, а додаје на крај реда.

У језику C++ ред се реализује класом `queue<T>` где `T` представља тип елемената на стеку. За њено коришћењем потребно је укључити заглавље `<queue>`. Подржане су следеће методе (све сложености $O(1)$):

- `push` - поставља дати елемент на крај реда
- `pop` - скида елемент са почетка реда (под претпоставком да ред није празан). Нагласимо да је ова метода типа `void` и да не враћа уклоњени елемент.
- `front` - очитава елемент на почетку реда (под претпоставком да ред није празан)
- `empty` - проверава да ли је ред празан
- `size` - враћа број елемената у реду

Ако се у реду чувају уређени парови или n-торке, тада се уместо методе `push`, може користити метода `emplace`, којој се само редом наводе елементи пара тј. n-торке (није потребно посебно позивати функцију за креирање пара тј. n-торке, што је неопходно када се користи `push`).

Ред у језику C++ је заправо само адаптер око неке колекције података (подразумевано реда са два краја) који корисника тера да поштује правила приступа реду и спречава да направи операцију која над редом није допуштена (попут приступа неком елементу који није на почетку).

4.6.4.1 Ред са два краја

Уопштење представља ред са два краја који допушта да се елементи и додају и узимају са било ког краја реда (та структура података заправо комбинује и функционалност стека и функционалност реда).

У језику C++ могуће је користити структуру `deque<T>`. За њено коришћење потребно је укључити заглавље `<deque>`. Подржане су следеће операције (све сложености $O(1)$).

- `push_front` - додавање елемента на почетак
- `push_back` - додавање елемента на крај
- `front` - читање елемента са почетка (под претпоставком да ред није празан)
- `back` - читање елемента са краја (под претпоставком да ред није празан)
- `pop_front` - уклањање елемента са почетка (под претпоставком да ред није празан). Нагласимо да је ова метода типа `void` и да не враћа уклоњени елемент.
- `pop_back` - уклањање елемента са краја (под претпоставком да ред није празан). Нагласимо да је ова метода типа `void` и да не враћа уклоњени елемент.
- `empty` - провера да ли је ред празан
- `size` - број елемената у реду

Интересантно, захваљујући специфичном начину имплементације, ова структура података подржава и оператор индексног приступа којим се елемент на датој позицији може прочитати или изменити у времену $O(1)$.

Задатак: Последњих k линија

Напиши програм који исписује k последњих линија учитаних са стандардног улаза.

Улаз: Са стандардног улаза се учитава број k ($1 \leq k \leq 100$), а затим једна по једна линија текста (њих највише 10^6).

Излаз: На стандардни излаз исписати последњих k линија (претпоставити да је увек учитано барем k линија).

Пример

Улаз	*Излаз*

`2`
`ispisati`
`poslednjih k`
`linija`
	`poslednjih k`
	`linija`

Решење

Пошто знамо максимални капацитет реда (максимални број елемената који се у неком тренутку налазе у реду), ред је могуће имплементирати коришћењем низа у који се елементи смештају кружно (када се низ попуни до краја, смештање елемената се наставља од почетка). Одржаваћемо две позиције тј. два показивача: на први елемент у реду и на прву слободну позицију (иза последњег елемента у реду). На почетку ће обе бити иницијализоване на нулу. Када год су те две једнаке, знамо да је ред празан. Да бисмо разликовали ситуацију потпуно попуњеног реда и празног реда, низ ћемо алоцирати тако да може да чува један елемент више од максималног капацитета. Тада ћемо потпуно попуњен ред моћи да препознамо тако што ће се слободна позиција налазити непосредно испред првог елемента у реду (специјално, могуће је да први елемент буде на позицији 0, а да је последња слободна позиција последња позиција у низу). Додавање вршимо на слободну позицију и повећавамо је за 1 (наравно, по модулу дужине низа). Скидање са почетка вршимо тако што позицију првог елемента у реду увећамо за 1 (поново, по модулу дужине низа). Број елемената у реду можемо израчунати као разлику између два показивача (опет рачунату по модулу дужине низа, коришћењем везе $(k - p) \bmod n = (k - p + n) \bmod n$, да би се избегло рачунање остатка при дељењу негативних бројева).

Анализа сложености. Сложеност свих операција је $O(1)$, а меморијска сложеност је $O(k)$, где је k максимални капацитет.

Напомена. Приметимо да се на овај начин може имплементирати и ред са два краја, ако му је познат максимални капацитет.

```
vector<string> red;
int poc = 0, kraj = 0;

bool pun() {
  return (kraj + 1) % red.size() == poc;
}

bool prazan() {
  return kraj == poc;
}

void dodaj(const string& s) {
  red[kraj] = s;
  kraj = (kraj + 1) % red.size();
}

string& pocetak() {
  return red[poc];
}

void skini() {
  poc = (poc + 1) % red.size();
}

int velicina() {
  return (kraj - poc + red.size()) % red.size();
}
```

Ред је могуће имплементирати и ручно, коришћењем једноструко повезане листе. Чувају се показивач на први и последњи елемент (оба имају вредност NULL када је ред празан). Пошто уклањање последњег елемента једноструко повезане листе није могуће извршити ефикасно, чак и када је познат показивач на последњи елемент, додавање се врши на крај реда, а читање и скидање елемената са почетка. Број елемената одржавамо у посебној променљивој.

Анализа сложености. Сложеност свих операција је $O(1)$, а меморијска сложеност је $O(k)$, где је k максимални капацитет.

Напомена. Имплементација реда са два краја захтева коришћење двоструко повезане листе. Наиме, уклањање последњег елемента једноструко повезане листе није могуће извршити у времену $O(1)$, јер је потребна измена претпоследњег елемента.

```
struct cvor {
  string s;
  cvor* sledeci;
};

cvor* napravi_cvor(const string& s) {
  cvor* novi = new cvor();
  // trebalo bi proveriti da li je alokacija uspela
  novi->s = s;
  novi->sledeci = nullptr;
  return novi;
}

cvor* pocetak = nullptr;
cvor* kraj = nullptr;
int velicina = 0;

void dodaj(const string& s) {
  if (kraj == nullptr)
    pocetak = kraj = napravi_cvor(s);
  else {
    kraj->sledeci = napravi_cvor(s);
    kraj = kraj->sledeci;
  }
  velicina++;
}

bool prazan() {
  return velicina == 0;
}

const string& pocetni() {
  return pocetak->s;
}

void ukloni() {
  cvor* sledeci = pocetak->sledeci;
  delete pocetak;
  pocetak = sledeci;
  if (pocetak == nullptr)
    kraj = nullptr;
  velicina--;
}
```

4.6.5 Имплементација дека

Као што смо видели у поглављу о применама структура података, једна веома корисна структура података је ред са два краја која комбинује функционалност стека и реда.

Ако се за имплементацију користе једноструко повезане листе, тада се у времену $O(1)$ може вршити убацивање на почетак и брисање са почетка, као и убацивање на крај (под претпоставком да одржавамо показивач на крај). Брисање са краја је операција сложености $O(n)$, чак и када се чува показивач на последњи елемент (јер не можемо да пронађемо показивач на претпоследњи елемент).

Ако се за имплементацију користе двоструко повезане листе, тада се убацивање и брисање и са почетка и са краја може извршити у времену $O(1)$. Исто је и са уметањем елемента у средину (када се зна његова позиција). Међутим, индексни приступ (приступ члану на датој позицији) у најгорем случају захтева време $O(n)$. Једини могући начин претраге је линеарна претрага, чак и када су елементи у реду сортирани.

У наставку ћемо приказати структуру података **дек** (енгл. deque) која се често користи за имплементацију редова са два краја. Слично као двоструко повезане листе, дек омогућава додавање и на почетак и на крај у времену $O(1)$ (додуше амортизованом). Важна предност у односу на двоструко повезане листе је то што је индексни приступ могућ у времену $O(1)$. Овим је омогућена и бинарна претрага, што може некада бити веома важно. Додавање и брисање елемената са средине није могуће извршити ефикасно (те операције захтевају време $O(n)$).

Прикажимо и једну могућу имплементацију дека.

Задатак: Дек

Програм подржава следеће наредбе које се извршавају над низом целих бројева.

- `F x` — додавање елемента `x` на почетак низа
- `f` — уклањање и испис елемента са почетка низа на стандардни излаз
- `B x` — додавање елемента `x` на крај низа
- `b` — уклањање и испис елемента са краја низа на стандардни излаз
- `r p` — испис елемента на позицији `p` на стандардни излаз (позиција је између 0 и тренутног броја елемената низа, умањеног за 1).

Улаз: Стандардни улаз садржи низ наредби, сваку у посебном реду.

Излаз: На стандардни излаз исписати ефекат извшавања наредби.

Пример

Улаз	*Излаз*
`F 1`	`2`
`B 3`	`1`
`F 2`	`3`
`B 4`	`4`
`r 0`	`2`
`r 1`	`1`
`r 2`	`4`
`r 3`	`3`
`f`	
`r 0`	
`b`	
`r 1`	

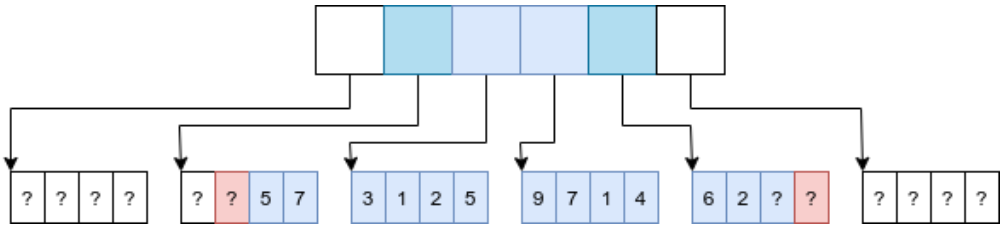
Решење

Задатак се једноставно може решити помоћу библиотечке имплементације дека.

```
#include <iostream>
#include <deque>

using namespace std;

int main() {
  deque<int> d;
```

```
  char c;
  while (cin >> c) {
    if (c == 'F') {
      int x;
      cin >> x >> ws;
      d.push_front(x);
    } else if (c == 'B') {
      int x;
      cin >> x >> ws;
      d.push_back(x);
    } else if (c == 'f') {
      cout << d.front() << endl;
      d.pop_front();
    } else if (c == 'b') {
      cout << d.back() << endl;
      d.pop_back();
    } else if (c == 'r') {
      int p;
      cin >> p >> ws;
      cout << d[p] << endl;
    }
  }
  return 0;
}
```

Дек можемо замислити као низ сегмената исте, фиксне величине. Сваки сегмент је структура која садржи низ елемената (било статички, било динамички алоциран) који представља неки део реда. На пример, ред чији су елементи 5, 7, 3, 1, 2, 5, 9, 7, 1, 4, 6, 2, може бити организован у 3 сегмента од по 4 елемента (у пракси је величина појединачних сегмената обично већа).



Слика 4.2: Пример имплементације дека

Дек чува низ показивача на појединачне сегменте. Два сегмента су карактеристична: леви сегмент у ком се налази почетак реда и десни сегмент у ком се налази крај реда. Оба могу бити само делимично попуњени. У сваком сегменту се чува прва слободна позиција на коју се може додати наредни елемент. Посебно се одржавају позиције та два сегмента. У празном деку леви и десни сегмент су суседни и празни (текући елемент левог сегмента је његов десни крај, а десног сегмента је његов леви крај).

Додавање елемента на почетак је веома једноставно ако леви сегмент није попуњен до краја. Елемент се само додаје на прву слободну позицију и она се помера налево. Када је леви сегмент потпуно попуњен, прелази се на попуњавање претходног сегмента, ако он постоји. Ако не постоји, онда се врши реалокација дека и проширује се његов број сегмената (обично је нови број сегмената неколико пута већи него полазни, како би се наредне реалокације дешавале све ређе и ређе и како би се обезбедило амортизовано константно време додавања). Приликом реалокације врши се само проширивање низа показивача на сегменте, а не самих сегмената, што је веома значајно ако се у деку чувају већи објекти (они се приликом алокације не копирају, а не копирају се ни сами сегменти). Приликом реалокације, показивачи на постојеће сегменте се у проширеном низу смештају на средину, а лево и десно од њих се смештају показивачи на ново алоциране сегменте који су иницијално празни. Реалокацијом се добијају сегменти лево од текућег потпуно попуњеног сегмента и додавање на почетак се врши у њих. Додавање на десни крај тече потпуно аналогно.

Брисање са левог краја се врши слично, уклањањем елемента из левог сегмента и преласком на наредни

сегмент ако се након брисања леви сегмент потпуно испразнио. Наравно, елементи сегмента се не уклањају физички, већ се само мењају индекси који указују на наредну слободну позицију. Брисање са десног краја је аналогно.

Индексни приступ елементу је могуће извршити у времену $O(1)$, тако што се прво одреди ком сегменту припада тражени елемент, а онда се прочита одговарајући елемент из тог сегмента. Једноставан трик је да се тражени индекс увећа за број празних елемената на левом крају левог сегмента. Тада се позиција сегмента у ком се елемент налази може једноставно израчунати као збир позиције левог сегмента и целобројног количника индекса i и величине једног сегмента, док се позиција унутар тог сегмента одређује као остатак у том дељењу.

Имплементација дека садржи велики број техничких детаља, па је приказујемо само информативно (читаоца упућујемо да покуша да самостално имплементира дек).

```
#include <iostream>

using namespace std;

// jedan segment
struct segment {
  // niz podataka
  int* podaci;
  // broj popunjenih elemenata niza
  int popunjeno;
  // pozicija u nizu na koju se moze ubaciti naredni levi element
  int levi;
  // pozicija u nizu na koju se moze ubaciti naredni desni element
  int desni;
};

// alokacija segmenta
segment* alocirajSegment(int velicina) {
  // alociramo novi segment
  segment* novi = new segment();
  // alociramo njegove podatke
  novi->podaci = new int[velicina];
  // nijedan podatak još nije upisan
  novi->popunjeno = 0;
  // trenutno nepopunjeni deo
  novi->levi = velicina - 1;
  novi->desni = 0;
  return novi;
}

// brisanje segmenta
void obrisiSegment(segment* s) {
  // brisemo podatke u segmentu
  delete[] s->podaci;
  // brisemo segment
  delete s;
}

struct dek {
  // niz pokazivača na segmente
  segment** segmenti;
  // broj segmenata u nizu
  int brojSegmenata;
  // velicina svakog segmenta
  int velicinaSegmenata;
  // pozicije na kojima se nalaze pokazivaci na krajnji levi
```

```
  // i krajnji desni segment (oba mogu biti polupopunjena)
  int levo, desno;
  int brojElemenata;
};


// vrši se realociranje deka tako da ima dati broj segmenata
void realocirajDek(dek& d, int brojSegmenata) {
  // pravimo novi niz pokazivača na segmente
  segment** noviSegmenti = new segment*[brojSegmenata];
  // postojeće pokazivače na segmente ćemo smestiti negde oko
  // sredine novog niza
  int uvecanje = (brojSegmenata - d.brojSegmenata) / 2;
  // ako je postojao stari niz pokazivača na segmente
  if (d.segmenti != nullptr)
    // kopiramo te pokazivače u novi niz pokazivača, krenuvši od
    // pozicije uvecanje
    for (int i = 0; i < d.brojSegmenata; i++)
      noviSegmenti[uvecanje + i] = d.segmenti[i];
  // alociramo nove segmente na početku i na kraju novog niza
  for (int i = 0; i < uvecanje; i++)
    noviSegmenti[i] = nullptr;
  for (int i = uvecanje + d.brojSegmenata; i < brojSegmenata; i++)
    noviSegmenti[i] = nullptr;

  // brišemo stari niz pokazivača na segmente
  delete[] d.segmenti;
  // preusmeravamo pokazivač na niz pokazivača
  d.segmenti = noviSegmenti;
  // ažuriramo broj segmenata
  d.brojSegmenata = brojSegmenata;
  // ažuriramo granice popunjenog dela niza
  d.levo += uvecanje;
  d.desno += uvecanje;
}


// vraća referencu na i-ti element u deku
int& iti(const dek& d, int i) {
  // trenutni broj elemenata u krajnjem levom segmentu
  int uLevom = d.segmenti[d.levo]->popunjeno;
  // pomeramo indeks tako da je pozicija 0 na početku krajnjeg levog
  // segmenta
  i += d.velicinaSegmenata - uLevom;
  // određujemo segment u kom se nalazi traženi element
  segment* s = d.segmenti[d.levo + i / d.velicinaSegmenata];
  // čitamo element iz tog segmenta
  return s->podaci[i % d.velicinaSegmenata];
}

// dodajemo element na početak deka
void dodajNaPocetak(dek& d, int x) {
  // ako je levi segment jos nealociran, alociramo ga
  if (d.segmenti[d.levo] == nullptr)
    d.segmenti[d.levo] = alocirajSegment(d.velicinaSegmenata);

  // ako je levi segment potpuno popunjen с леве стране
  if (d.segmenti[d.levo]->levi < 0) {
```

```
    // ako levo od njega nema više alociranih segmenata
    if (d.levo == 0)
      // realociramo dek i time alociramo nove segmente
      realocirajDek(d, d.brojSegmenata * 2);
    // prelazimo u prethodni segment
    d.levo--;
  }
  // segment u koji se može upisati element
  segment *s = d.segmenti[d.levo];

  // upisujemo element na tekuću slobodnu poziciju i pomeramo se
  // nalevo
  s->podaci[s->levi--] = x;
  // uvećavamo broj popunjenih elemenata
  s->popunjeno++;
  d.brojElemenata++;
}

// dodajemo element na kraj deka
void dodajNaKraj(dek& d, int x) {
  // ako je desni segment jos nealociran, alociramo ga
  if (d.segmenti[d.desno] == nullptr)
    d.segmenti[d.desno] = alocirajSegment(d.velicinaSegmenata);

  // ako je desni segment potpuno popunjen sa desne strane
  if (d.segmenti[d.desno]->desno >= d.velicinaSegmenata) {
    // ako desno od njega nema više alociranih segmenata
    if (d.desno == d.brojSegmenata - 1)
      // realociramo dek i time alociramo nove segmente
      realocirajDek(d, d.brojSegmenata * 2);
    // prelazimo na naredni segment
    d.desno++;
  }
  // segment u koji se može upisati element
  segment *s = d.segmenti[d.desno];

  // upisujemo element na tekuću slobodnu poziciju i pomeramo se
  // nalevo
  s->podaci[s->desni++] = x;
  // uvećavamo broj popunjenih elemenata
  s->popunjeno++;
  d.brojElemenata++;
}

// uklanjamo element sa pocetka deka
int ukloniSaPocetka(dek& d) {
  if (d.segmenti[d.levo]->levi >= d.velicinaSegmenata) {
    d.levo++;
    d.segmenti[d.levo]->levi = 0;
  }
  segment *s = d.segmenti[d.levo];
  s->popunjeno--;
  d.brojElemenata--;
  return s->podaci[++s->levi];
}


// uklanjamo element sa kraja deka
```

```
int ukloniSaKraja(dek& d) {
  if (d.segmenti[d.desno]->desni < 0) {
    d.desno--;
    d.segmenti[d.desno]->desni = d.velicinaSegmenata;
  }
  segment *s = d.segmenti[d.desno];
  s->popunjeno--;
  d.brojElemenata--;
  return s->podaci[--s->desni];
}

void ispisiDek(dek& d) {
  for (int i = 0; i < d.brojElemenata; i++)
    cout << iti(d, i) << " ";
  cout << endl;
}


// brisanje deka
void obrisiDek(dek& d) {
  // brisemo sve segmente
  for (int i = 0; i < d.brojSegmenata; i++)
    obrisiSegment(d.segmenti[i]);
  // brisemo niz pokazivaca na segmente
  delete[] d.segmenti;
}

void inicijalizujDek(dek& d, int brojElemenata, int velicinaSegmenata = 3) {
  d.velicinaSegmenata = velicinaSegmenata;
  d.segmenti = nullptr;
  d.brojElemenata = 0;
  d.brojSegmenata = 0;
  d.levo = -1;
  d.desno = 0;

  realocirajDek(d, brojElemenata);
}


int main() {
  // gradimo prazan dek i zatim ga inicijalizujemo tako da inicijalno
  // moze da primi 10 elemenata
  dek d;
  inicijalizujDek(d, 10);

  char c;
  while (cin >> c) {
    if (c == 'F') {
      int x;
      cin >> x >> ws;
      dodajNaPocetak(d, x);
    } else if (c == 'B') {
      int x;
      cin >> x >> ws;
      dodajNaKraj(d, x);
    } else if (c == 'f') {
      cout << ukloniSaPocetka(d) << endl;
    } else if (c == 'b') {
```

```
      cout << ukloniSaKraja(d) << endl;
    } else if (c == 'r') {
      int p;
      cin >> p >> ws;
      cout << iti(d, p) << endl;
    }
    ispisiDek(d);
  }

  // brisemo dek
  obrisiDek(d);

  return 0;
}
```

4.6.6 Бинарно дрво претраге - имплементација скупова и мапа

Дрвета су структуре података које се користе за представљање хијерархијских односа између делова (на пример, израза, система директоријума и датотека, организације елемената унутар HTML странице, синтаксе програма унутар преводиоца и слично). Бинарно дрво је рекурзивно дефинисани тип података: или је празно или садржи неки податак и лево и десно поддрво.

Уобичајени начин за представљање дрвета у језику C++ је преко чворова увезаних помоћу показивача.

```
struct cvor {
  int x;
  cvor *levo, *desno;
};
```

У случају да се дрвета користе за представљање скупова или мултискупова у чворовима се чува само један податак (елемент скупа). Ако се користе за представљање мапа или мултимапа, у чворовима се чувају два податка: кључ и вредност придружена кључу.

```
struct cvor {
  int k, v;
  cvor *levo, *desno;
}
```

Пошто је дрво рекурзивно-дефинисана структура података, најлакше је функције које оперишу са дрветима реализовати рекурзивно. У неким ситуацијама је могуће релативно лако елиминисати рекурзију, док је у неким другим ситуацијама имплементирање нерекурзивних операција компликовано (и захтева коришћење стека). Већина функција које ћемо ми имплементирати ће бити рекурзивна.

У наставку ћемо се бавити употребом дрвета за имплементацију асоцијативних контејнера (скупова, мултискупова, мапа и мултимапа). За то се користи посебна организација вредности у бинарном дрвету.

Дрво је **уређено бинарно дрво** (тј. **бинарно дрво претраге**, engl. binary search tree), ако је празно или ако је његово лево и десно поддрво уређено и ако је чвор у корену већи од свих чворова у левом поддрвету и мањи од свих чворова у десном поддрвету. Оваква дефиниција подразумева да у дрвету нема дупликата, што је случај у обичним скуповима и мапама. У мултискуповима и мултимапама је дозвољено постојање дупликата у дрвету и тада се дефиниција проширује тако што се допушта да је чвор у корену већи или једнак од свих чворова у десном поддрвету.

Нагласимо да није довољно проверити да је вредност у сваком чвору већа од вредности у корену левог поддрвета и вредности у корену десног поддрвета! На пример, то важи у наредном дрвету, али оно није претраживачко, јер је елемент 4 који је већи од елемента 3 завршио лево од њега.

```
    3
  2   5
1   4
```

4.6.6.1 Балансирано бинарно дрво

Мана класичних уређених бинарних дрвета је то што ако нису балансирана операције претраге, уметања и брисања могу захтевати линеарно време у односу на број чворова у дрвету. Балансирана бинарна дрвета гарантују да се то не може догодити и да је временска сложеност најгорег случаја ових операција логаритамска у односу на број чворова у дрвету. Две најчешће коришћене врсте балансираних бинарних дрвета.

- Адељсон-Вељски Ландисова дрвета (енгл. AVL tree)
- Црвено-црна дрвета (енгл. Red-black tree, RBT)

Њихов детаљан опис превазилази домен овог материјала.

Задатак: Мапа

Над речником који слика ниске карактера дужине највише 10 карактера у целе бројеве се могу вршити следеће наредбе:

- `w k v` — *write* — кључу `k` се придружује вредност `v` (ако је кључу `k` већ додељена нека вредност, она се занемарује)
- `r k` — *read* — на стандардни излаз се исписује вредност придружена кључу `k`. Ако кључу није придружена вредност, исписује се `-`.

Написати програм који чита и извршава низ оваквих наредби (наравно, водити рачуна о ефикасности).

Улаз: Свака линија стандардног улаза садржи једну наредбу.

Излаз: На стандардни излаз исписати резултат извршавања свих наредби.

Пример

Улаз	*Излаз*
`w pera 5`	`4`
`w ana 4`	`-`
`r ana`	`3`
`r mika`	`5`
`w mika 3`	
`r mika`	
`r pera`	

Решење

Задатак можемо решити преко ручно имплементираног уређеног бинарног дрвета. Једноставности ради, нећемо користити балансирано бинарно дрво (мада се тиме нарушава сложеност најгорег случаја, када кључеви пристижу у приближно сортираном редоследу).

```
#include <iostream>

using namespace std;

struct cvor {
  string kljuc;
  int vrednost;
  cvor *levo, *desno;
};

cvor* napravi_cvor(const string& kljuc, int vrednost) {
  cvor* novi = new cvor();
  // potrebno je proveriti da li je alokacija cvora uspela
  novi->kljuc = kljuc;
  novi->vrednost = vrednost;
  novi->levo = novi->desno = nullptr;
  return novi;
}

cvor* ubaci(cvor* koren, const string& kljuc, int vrednost) {
```

```
  if (koren == nullptr)
    return napravi_cvor(kljuc, vrednost);
  if (kljuc < koren->kljuc)
    koren->levo = ubaci(koren->levo, kljuc, vrednost);
  else if (kljuc > koren->kljuc)
    koren->desno = ubaci(koren->desno, kljuc, vrednost);
  else
    koren->vrednost = vrednost;
  return koren;
}

cvor* pronadji(cvor* koren, const string& kljuc) {
  if (koren == nullptr)
    return nullptr;
  if (kljuc < koren->kljuc)
    return pronadji(koren->levo, kljuc);
  else if (kljuc > koren->kljuc)
    return pronadji(koren->desno, kljuc);
  return koren;
}

bool pronadji(cvor* koren, const string& kljuc, int& vrednost) {
  cvor* c = pronadji(koren, kljuc);
  if (c == nullptr)
    return false;
  vrednost = c->vrednost;
  return true;
}

void obrisi_drvo(cvor* koren) {
  if (koren != nullptr) {
    obrisi_drvo(koren->levo);
    obrisi_drvo(koren->desno);
    delete koren;
  }
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  cvor* koren = nullptr;

  char c;
  while (cin >> c) {
    if (c == 'w') {
      string k; int v;
      cin >> k >> v >> ws;
      koren = ubaci(koren, k, v);
    } else if (c == 'r') {
      string k;
      cin >> k >> ws;
      int v;
      if (pronadji(koren, k, v))
        cout << v << '\n';
      else
        cout << "-" << endl;
    }
  }
```

```
  obrisi_drvo(koren);

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Скуп

Над скупом елемената могу се вршити следеће наредбе:

- `i x` — *insert* — уметање вредности `x` у скуп (ако се она већ налази у скупу, скуп се не мења)
- `e x` — *erase* — брисање вредности `x` из скупа елемената (ако се она не налази у скупу, скуп се не мења)
- `m` — *min* — на стандардни излаз исписује се минимални елемент скупа или `-` ако је скуп празан
- `M` — *max* — на стандардни излаз исписује се максимални елемент скупа или `-` ако је скуп празан
- `n x` — *next* — на стандардни излаз исписује се следбеник елемента `x` тј. најмањи број у скупу који је строго већи од `x` (ако такав елемент не постоји, исписује се `-`)

Написати програм који чита и извршава низ оваквих наредби (наравно, водити рачуна о ефикасности).

Улаз: Свака линија стандардног улаза садржи једну наредбу.

Излаз: На стандардни излаз исписати резултат извршавања свих наредби.

Пример

Улаз	*Излаз*
`i 3`	`3`
`i 5`	`5`
`i 4`	`4`
`m`	`5`
`M`	`8`
`e 3`	`-`
`m`	
`i 8`	
`n 4`	
`n 7`	
`n 8`	

Решење

Задатак се лако решава коришћењем библиотечке имплементације уређеног скупа. У језику C++ могуће је користити тип `set<int>` (али не и `unordered_set<int>`).

```
#include <iostream>
#include <set>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  char c;
  set<int> s;
  while (cin >> c) {
    if (c == 'i') {
      int x;
      cin >> x >> ws;
      s.insert(x);
    } else if (c == 'e') {
      int x;
      cin >> x >> ws;
      s.erase(x);
    } else if (c == 'm') {
```

```
      if (!s.empty())
        cout << *s.begin() << '\n';
      else
        cout << "-" << '\n';
    } else if (c == 'M') {
      if (!s.empty())
        cout << *prev(s.end()) << '\n';
      else
        cout << "-" << '\n';
    } else if (c == 'n') {
      int x;
      cin >> x >> ws;
      auto it = s.upper_bound(x);
      if (it != s.end())
        cout << *it << '\n';
      else
        cout << "-" << '\n';
    }
  }
  return 0;
}
```

Задатак можемо решити и ручном имплементацијом скупа коришћењем претраживачког (уређеног) бинарног дрвета.

Уметање у бинарно дрво претраге се дефинише веома једноставном рекурзивном конструкцијом. Уметање се увек врши на место листа. Вредност која се умеће се пореди са вредношћу у корену, ако је мања од ње умеће се лево, а ако је већа од ње умеће се десно. Ако је једнака вредности у корену, разликују се случај у ком се дупликати допуштају (тада дрво представља скуп) и случај у ком се не допуштају (тада дрво представља мултискуп). Ако се дупликати не допуштају, тада се приликом детекције да се вредност већ налази у корену дрвета, уметање просто прекида (у супротном би се уметање наставило у левом поддрвету).

Брисање је донекле компликованије реализовати него уметање. Можемо га реализовати наредним рекурзивним алгоритмом.

- ако је елемент који се брише мањи од корена, он се брише из левог поддрвета
- ако је елемент који се брише већи од корена, он се брише из десног поддрвета
- ако је једнак корену, брише се корен
  - ако десно поддрво не постоји, резултат је лево поддрво
  - у супротном се брише чвор који садржи најмањи елемент из десног поддрвета и та вредност се уписује у корен. Ова операције се може једноставно извести засебном рекурзивном конструкцијом.

Приликом брисања корена, у корен је могао бити доведен и највећи елемент левог поддрвета.

Минимум се одређује веома једноставно, као крајњи леви потомак корена (у овом случају није потребно користити рекурзивну имплементацију). Максимум се одређује аналогно.

Следбеник датог броја се лако може одредити рекурзивном конструкцијом.

- Ако је дрво празно, следбеник не постоји (у имплементацији је могуће вратити показивач NULL).
- Ако је вредност у корену мања или једнака од датог броја, тада се следбеник рекурзивно одређује у десном поддрвету (јер он мора бити строго већи од датог броја, а вредности лево од корена и у њему нису такве). Наравно, ако десно дрво не постоји, биће враћен показивач NULL.
- У супротном се следбеник рекурзивно одређује у левом поддрвету. Ако он постоји, то је коначан резултат, а ако не постоји, тада је следбеник корен.

Ову функцију је могуће имплементирати и нерекурзивно.

Анализа сложености. Под претпоставком да је дрво балансирано, сложеност свих описаних операција је $O(\log n)$, где је n број чворова у дрвету. Наиме, у свим случајевима се анализира или само лево или само десно поддрво, па је једначина којом се описује број корака $T(n) = T(n/2) + O(1)$.

```
#include <iostream>

using namespace std;

struct cvor {
  int x;
  cvor *levo, *desno;
};

cvor* napraviCvor(int x) {
  cvor* novi = new cvor();
  // potrebna je provera da li je alokacija uspela
  novi->levo = novi->desno = nullptr;
  novi->x = x;
  return novi;
}

cvor* ubaci(cvor* drvo, int x) {
  if (drvo == nullptr)
    return napraviCvor(x);
  if (x < drvo->x)
    drvo->levo = ubaci(drvo->levo, x);
  else if (x > drvo->x)
    drvo->desno = ubaci(drvo->desno, x);
  return drvo;
}

int minimum(cvor* drvo) {
  while (drvo->levo != nullptr)
    drvo = drvo->levo;
  return drvo->x;
}

int maksimum(cvor* drvo) {
  while (drvo->desno != nullptr)
    drvo = drvo->desno;
  return drvo->x;
}

// brisanje date vrednosti
// Iz datog drveta se ukljanja čvor sa najmanjom vrednošću
cvor* obrisiMin(cvor* drvo, int& x) {
   // iz praznog drveta nema šta da se briše
   if (drvo == nullptr) return nullptr;
   // ako je levo poddrvo prazno
   if (drvo->levo == nullptr) {
      // brišemo drvo i vraćamo desno poddrvo
      cvor* desno = drvo->desno;
      x = drvo->x;
      delete drvo;
      return desno;
    }
    // u suprotnom brišemo najmanji element levog poddrveta
    drvo->levo = obrisiMin(drvo->levo, x);
    return drvo;
```

```
}

// Iz datog drveta ukljanja dati čvor sa datom vrednošću
cvor* obrisi(cvor* drvo, int x) {
  // iz praznog drveta nema šta da se briše
  if (drvo == nullptr)
    return nullptr;
  // čvor se nalazi levo, pa se tamo i briše
  if (x < drvo->x)
    drvo->levo = obrisi(drvo->levo, x);
  // čvor se nalazi desno, pa se tamo i briše
  else if (x > drvo->x)
    drvo->desno = obrisi(drvo->desno, x);
  // čvor se nalazi u korenu
  else {
    if (drvo->desno == nullptr) {
      // ako je desno poddrvo prazno brišemo drvo i vraćamo levo poddrvo
      cvor* levo = drvo->levo;
      delete drvo;
      return levo;
    } else {
      // desno poddrvo nije prazno, pa brišemo najmanji čvor iz
      // njega i vrednost iz tog čvora upisujemo u drvo
      int min;
      drvo->desno = obrisiMin(drvo->desno, min);
      drvo->x = min;
    }
  }
  // fizički čvor u kome je drvo se nije promenio
  return drvo;
}


cvor* sledbenik(cvor* drvo, int x) {
  if (drvo == nullptr)
    return nullptr;
  if (drvo->x <= x)
    return sledbenik(drvo->desno, x);
  else {
    cvor* sl = sledbenik(drvo->levo, x);
    if (sl != nullptr)
      return sl;
    return drvo;
  }
}

// brisanje celog drveta
void obrisi(cvor* drvo) {
   if (drvo != nullptr) {
      obrisi(drvo->levo);
      obrisi(drvo->desno);
      delete drvo;
   }
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
```

```
  char c;
  cvor* drvo = nullptr;

  while (cin >> c) {
    if (c == 'i') {
      int x;
      cin >> x >> ws;
      drvo = ubaci(drvo, x);
    } else if (c == 'e') {
      int x;
      cin >> x >> ws;
      drvo = obrisi(drvo, x);
    } else if (c == 'm') {
      if (drvo != nullptr)
        cout << minimum(drvo) << '\n';
      else
        cout << "-" << '\n';
    } else if (c == 'M') {
      if (drvo != nullptr)
        cout << maksimum(drvo) << '\n';
      else
        cout << "-" << '\n';
    } else if (c == 'n') {
      int x;
      cin >> x >> ws;
      cvor* s = sledbenik(drvo, x);
      if (s != nullptr)
        cout << s->x << '\n';
      else
        cout << "-" << '\n';
    }
  }
  obrisi(drvo);
  return 0;
}
```

4.6.7 Хип - имплементација реда са приоритетом

Претраживачко бинарно дрво се може употребити за имплементацију редова са приоритетом, уз релативно добре перформансе, под претпоставком да је дрво балансирано (AVL или RBT). Подсетимо се, основне операције су уметање елемента у ред, читање елемента са највећим приоритетом и његово избацивање. Уметање произвољне вредности се врши у логаритамској сложености, максимални вредност се може пронаћи у логаритамској сложености спуштајући се надесно докле год је то могуће и тај чвор (лист) је могуће једноставно уклонити, поново у логаритамској сложености. Додатно, могуће је претражити дрво, па и обрисати произвољну вредност у логаритамској сложености.

Ако се задржимо само на три основне функционалности, могућа је другачија имплементација која има боље перформансе него претраживачко дрво. Поново се користи бинарно дрво, али овај пут организовану у структуру података која се назива **хип** (енгл. heap).

Макс-хип (енгл. max-heap) је бинарно дрво које задовољава услов да је сваки ниво осим евентуално последњег потпуно попуњен, као и да је вредност сваког чвора већа или једнака од вредности у његовој деци. Мин-хип се дефинише аналогно, једино што се захтева да је вредност сваког чвора мања или једнака од вредности у његовој деци. Макс-хип омогућава веома ефикасно одређивање максималног елемента у себи (он се увек налази у корену), а видећемо и веома ефикасно његово уклањање. Пошто је и операција уметања новог елемента у хип ефикасна, ова структура података је веома добар кандидат за имплементацију реда са приоритетом.

Задатак: Хип

Програм препознаје следеће наредбе које се извршавају над колекцијом података (у којој се могу јавити и дупликати):

- `i x` — *insert* — у колекцију бројева умеће се вредност `x`
- `m` — *maximum* — на стандардни излаз се исписује највећи елемент који се тренутно налази у колекцији
- `r` — *remove* — из колекције се избацује највећи елемент

Напиши програм који чита и извршава низ оваквих наредби (наравно, водити рачуна о ефикасности).

Улаз: Свака линија стандардног улаза садржи једну наредбу.

Излаз: На стандардни излаз се исписује резултат извршавања свих наредби.

Пример

Улаз	*Излаз*
`i 3`	`7`
`i 7`	`8`
`i 4`	`4`
`m`	
`i 8`	
`m`	
`r`	
`r`	
`m`	

Решење

Описане наредбе су основне три наредбе реда са приоритетом.

Задатак се веома једноставно може решити коришћењем библиотечке имплементације реда са приоритетом.

```
#include <iostream>
#include <queue>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  priority_queue<int> pq;
  char c;
  while (cin >> c) {
    if (c == 'i') {
      int x;
      cin >> x >> ws;
      pq.push(x);
    } else if (c == 'm') {
      cout << pq.top() << '\n';
    } else if (c == 'r') {
      pq.pop();
    }
  }
  return 0;
}
```

Задатак се може решити употребом уређеног мултискупа (који се може имплементирати коришћењем уређеног бинарног дрвета). У језику C++ најједноставније је употребити библиотечку имплементацију `multiset`.

Анализа сложености. Уметање, брисање и проналажење најмањег елемента у мултискупу извршавају се у времену $O(\log k)$, где је k број елемената у мултискупу, па је укупна сложеност алгоритма $O(n \log n)$, где је n број наредби које се извршавају.

Анализа сложености. Уметање, брисање и претрага уређеног речника се извршава у времену $O(\log k)$, где је k број елемената у мултискупу. Нажалост, проналажење најмањег елемента се извршава у времену $O(k)$,

па је стога, сложеност најгорег случаја алгоритма $O(n^2)$, где је n број наредби које се извршавају.

```
#include <iostream>
#include <set>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  multiset<int> pq;
  char c;
  while (cin >> c) {
    if (c == 'i') {
      int x;
      cin >> x >> ws;
      pq.insert(x);
    } else if (c == 'm') {
      cout << *(prev(pq.end())) << '\n';
    } else if (c == 'r') {
      pq.erase(prev(pq.end()));
    }
  }
  return 0;
}
```

Размотримо наредну имплементацију макс-хипа. Дрво смештамо у низ. Корен на позицију 0, наредна два елемента на позиције 1 и 2, затим наредне елементе на позиције 3, 4, 5 и 6 итд. На пример, хип

```
              11
         9              7
    4        3      5     6
 2    3   1
```

представљамо низом

```
11 9 7 4 3 5 6 2 3 1
```

Захтевамо да се хип попуњава редом и су сви нивои осим евентуално последњег комплетно попуњени, а да су у последњем нивоу попуњени само почетни елементи. Напишимо позиције у низу и њихов распоред у бинарном дрвету.

```
                  0
         1                 2
    3        4        5         6
 7    8    9   10  11   12  13    14
```

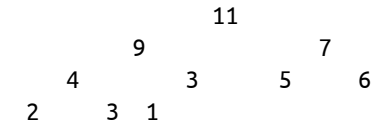
Лако је уочити да ако се корен налази на позицији i, онда се његово лево дете налази на позицији $2i + 1$, а десно дете на позицији $2i + 2$, док му се родитељ налази на позицији $\lfloor \frac{i-1}{2} \rfloor$.

Највећи елемент се увек налази у корену, тј. на позицији 0 (што се лако може доказати индукцијом имајући у виду да је вредност у сваком чвору већа од вредности његовој у деци). Време потребно за налажење максимума је очигледно $O(1)$.

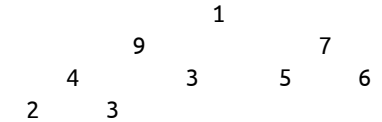
Размотримо како бисмо могли реализовати операцију уклањања највећег елемента из макс-хипа. Пошто се он налази у корену, а дрво мора бити потпуно попуњено (осим евентуално последњег нивоа) на место избаченог елемента је најједноставније уписати последњи елемент из хипа (крајњи десни елемент последњег нивоа). Овим је задовољен услов за распоред елемената у дрвету, али је својство хипа могуће нарушено, јер тај елемент не мора бити већи од свих осталих (и обично није). На срећу, поправку можемо извршити релативно једноставно. Потребно је да упоредимо вредност у корену са вредношћу његове деце (ако постоје). Ако је вредност у корену већа или једнака од тих вредности, онда корен задовољава услов макс-хипа и процедура може да се заврши (јер за све остале чворове знамо да задовољавају тај услов, јер је избацивање кренуло од исправног хипа). У супротном, разменимо вредност у корену са већом од вредности његове деце (тј. са вредношћу његовог детета, ако има само једно дете). Након тога корен задовољава услов макс-хипа, и

преостаје једино да проверимо (и евентуално поправимо) оно поддрво у чијем је корену завршила вредност корена дрвета. Ово је проблем истог облика, само мање димензије у односу на полазни и лако се, дакле, решава индуктивно-рекурзивном конструкцијом.

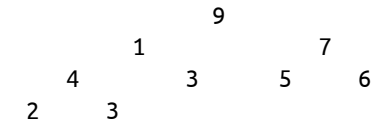
Прикажимо рад овог алгоритма на једном примеру.



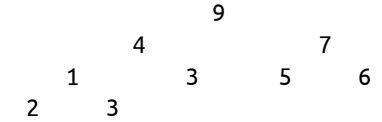
Избацујемо елемент са врха и на његово месту премештамо последњи елемент.



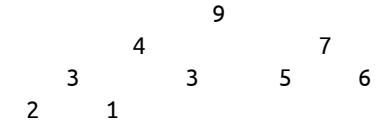
Мењамо вредност у корену, са већом од вредности његова два детета.



Исти поступак примењујемо на лево поддрво. Размењујемо вредност у корену са већом од вредности његова два детета.



Поступак се још једном примењује на поддрво са кореном 1.



Након овога, добијено дрво представља макс-хип.

Број корака померања наниже у најгорем случају одговара висини дрвета. Пошто у потпуном дрвету висине може да стане $2^{+1} - 1$ елемената, висина логаритамски зависи од броја елемената. Дакле, време потребно за уклањање највећег елемента из хипа у којем се налази n елемената је $O(\log n)$.

Уметање новог елемента функционише по сличном, али обрнутом принципу. Елемент је најједноставније убацити на крај низа. Ипак, могуће је да му ту није место, јер је можда већи од свог родитеља. У том случају могуће је извршити њихову размену. Након замене, поддрво T са кореном у новом чвору задовољава услов хипа и цело дрво без поддрвета T такође задовољава услов хипа. Сваким померањем новог елемента навише, остатак дрвета без поддрвета T се смањује и проблем се своди на проблем мање димензије и решава се индуктивно-рекурзивном конструкцијом.

Пошто је и у случању операције померања наниже и у случају операције померања навише рекурзија репна, она се може релативно једноставно уклонити.

Још један начин да имплементирамо померање наниже је да одредимо позицију већег од два детета и онда то дете упоредимо са родитељем.

```
#include <iostream>
#include <vector>

using namespace std;

int roditelj(int i) {
  return (i-1) / 2;
```

```
}

int levoDete(int i) {
  return 2*i + 1;
}

int desnoDete(int i) {
  return 2*i + 2;
}

int najveci(const vector<int>& hip) {
  return hip[0];
}

// element na poziciji k se pomera naniže, razmenom sa svojom decom
// sve dok se ne zadovolji uslov hipa
void pomeriNanize(vector<int>& hip, int k) {
  // pozicija najvećeg čvora (razmatrajući roditelja i njegovu decu)
  int najveci = k;
  int levo = levoDete(k), desno = desnoDete(k);
  if (levo < hip.size() && hip[levo] > hip[najveci])
    najveci = levo;
  if (desno < hip.size() && hip[desno] > hip[najveci])
    najveci = desno;
  // ako roditelj nije najveći
  if (najveci != k) {
    // menjamo roditelja i veće dete
    swap(hip[najveci], hip[k]);
    // rekurzivno obrađujemo veće dete
    pomeriNanize(hip, najveci);
  }
}

// izbacivanje najvećeg elementa iz hipa
void izbaciNajveci(vector<int>& hip) {
  // poslednji element izbacujemo iz hipa i
  // upisujemo ga na početnu poziciju
  hip[0] = hip.back();
  hip.pop_back();
  // pomeramo početni element naniže dok se ne
  // zadovolji uslov hipa
  pomeriNanize(hip, 0);
}

// element na poziciji k se pomera naviše, razmenom sa svojim
// roditeljem, sve dok se ne zadovolji uslov hipa
void pomeriNavise(vector<int>& hip, int k) {
  // pozicija roditelja čvora k
  int r = roditelj(k);
  // ako čvor k nije koren i ako je veći od roditelja
  if (k > 0 && hip[k] > hip[r]) {
     // razmenjujemo ga sa njegovim roditeljem
     swap(hip[k], hip[r]);
     // pomeramo roditelja navise
     pomeriNavise(hip, r);
  }
}
```

```
// ubacuje se element x u hip
void ubaci(vector<int>& hip, int x) {
  // element dodajemo na kraj
  hip.push_back(x);
  // pomeramo ga naviše dok se ne zadovolji uslov hipa
  pomeriNavise(hip, hip.size() - 1);
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  vector<int> hip;
  char c;
  while (cin >> c) {
    if (c == 'i') {
      int x;
      cin >> x >> ws;
      ubaci(hip, x);
    } else if (c == 'm') {
      cout << najveci(hip)  << '\n';
    } else if (c == 'r') {
      izbaciNajveci(hip);
    }
  }
  return 0;
}
```

Рекурзија у функцијама за померање елемената навише и наниже је репна, па се лако елиминише.

```
// element na poziciji k se pomera naniže, razmenom sa svojom decom
// sve dok se ne zadovolji uslov hipa
void pomeriNanize(vector<int>& hip, int k) {
  while (true) {
    // pozicija najvećeg čvora
    // (razmatrajući roditelja i njegovu decu)
    int najveci = k;
    int levo = levoDete(k), desno = desnoDete(k);
    if (levo < hip.size() && hip[levo] > hip[najveci])
      najveci = levo;
    if (desno < hip.size() && hip[desno] > hip[najveci])
      najveci = desno;
    // ako je roditelj najveći, uslov hipa je zadovoljen
    if (najveci == k)
      break;
    // menjamo roditelja i veće dete
    swap(hip[najveci], hip[k]);
    // obrađujemo veće dete
    k = najveci;
  }
}

// element na poziciji k se pomera naviše, razmenom sa svojim
// roditeljem, sve dok se ne zadovolji uslov hipa
void pomeriNavise(vector<int>& hip, int k) {
  int r = roditelj(k);
  while (k > 0 && hip[k] > hip[r]) {
     swap(hip[k], hip[r]);
     k = r;
     r = roditelj(k);
  }
```

```
}
```

Још један начин да имплементирамо померање наниже је да одредимо позицију већег од два детета и онда то дете упоредимо са родитељем.

```
// element na poziciji k se pomera naniže, razmenom sa svojom decom
// sve dok se ne zadovolji uslov hipa
void pomeriNanize(vector<int>& hip, int k) {
  int roditelj = k;
  // pretpostavljamo da je levo dete vece
  int veceDete = levoDete(k);
  // dok god čvor ima dece
  while (veceDete < hip.size()) {
    // poredimo da li je desno dete vece od levog
    int desno = veceDete + 1;
    if (desno < hip.size() && hip[desno] > hip[veceDete])
       veceDete = desno;
    // ako je roditelj veci ili jednak od oba deteta,
    // uslov hipa je zadovoljen
    if (hip[roditelj] >= hip[veceDete])
      break;
    // menjamo roditelja i vece dete
    swap(hip[veceDete], hip[roditelj]);
    // nastavljamo obradu od veceg deteta
    roditelj = veceDete;
    veceDete = levoDete(roditelj);
  }
}
```

Задатак: Сортирање бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Сортирање бројева може бити реализовано директном имплементацијом алгоритма *heap sort* у којој је ред са приоритетом реализован помоћу ручно имплементираног хипа.

Интересанто, исти низ (или вектор) се може користити и за смештање полазног низа и за смештање хипа и за смештање сортираног низа (елементи хипа се могу сместити у почетни део низа, док се сортирани део може сместити у крајњи део низа). Тиме се штеди меморијски простор, међутим, потребно је мало прилагодити функције за рад са хипом. Наиме, функција за померање наниже поред низа или вектора у коме су смештени елементи, мора као параметар да прими и број елемената низа или вектора који представља хип (јер стварна димензија може бити и већа, ако се простор иза хипа користи за смештање елемената сортираног низа). На основу тог броја је лако установити који елементи низа припадају, а који не припадају хипу. Поређење са `hip.size()` се мора заменити поређењем са тим бројем.

Један начин да се од низа формира хип (у истом меморијском простору) је да се крене од празног хипа и да се један по један елемент убацује у хип. Овај начин се назива **формирање хипа наниже** или **Вилијамсов метод**.

```
// formira hip od elemenata niza a dužine n
// hip se smešta u istom memorijskom prostoru kao i niz
void formirajHip(int a[], int n) {
  // ubacujemo jedan po jedan element u hip i pomeramo ga naviše
  for (int i = 1; i < n; i++)
     pomeriNavise(a, i);
}
```

Инваријанта спољне петље (тј. индуктивна хипотеза) је то да елементи на позицијама $[0, i)$ чине макс-хип. Пошто знамо да операција померања елемента навише исправно умеће последњи елемент у постојећи макс-

хип, лако је доказати да инваријанта остаје одржана. На крају петље је $i = n$, што значи да су сви елементи низа (елементи на позицијама $[0, n)$) сложени у макс-хип. Додатно, размене не мењају мултискуп елемената низа, па је мултискуп елемената у хипу једнак мултискупу елемената полазног низа.

Сложеност најгорег случаја формирања хипа једнака је $\Theta(n \log n)$. Наиме, сложеност операције уметања тј. померања елемента навише у хипу који има k елемената једнака је $O(\log k)$, па је укупна сложеност формирања хипа асимптотски једнака збиру свих логаритама од 1 до n, што је, како смо раније видели $\Theta(n \log n)$.

Други, ефикаснији начин формирања хипа назива се **формирање хипа навише** или **Флојдов метод**. Идеја је да се елементи оригиналног низа обилазе од позади и да се сваки елемент уметне у хип чији корен представља, тако што се спусти наниже кроз хип. Индуктивна хипотеза у овом приступу је то да су сви елементи из интервала (i, n) корени исправних макс-хипова. На пример, ако је дат низ

```
7 2 8 9 4 6 5 1 3
```

сви елементи осим прва два чине коренове исправних хипова. Заиста, у низу је смештено следеће дрво.

```
          7
      2       8
   9    4  6   5
 1   3
```

Елементи од 4, 6, 5, 1 и 3 немају наследнике и тривијално представљају исправне једночлане хипове. И у општем случају, сви елементи у другој половини низа представљају листове и за њих је ова инваријанта тривијално испуњена. Елемент 9 је већи од оба своја сина, па је корен исправног макс-хипа. Слично, елемент 8 је већи од оба своја сина, па је и он корен исправног макс-хипа.

Поставља се питање како проширити инваријанту. Елемент на позицији i не мора да буде већи од своје деце, али знамо да су оба његова детета корени исправних хипова. Стога је само потребно спустити елемент са врха на његово место, што је операција коју смо већ разматрали приликом операције брисања елемента из хипа. Дакле, сваки елемент од оног на позицији $\lfloor \frac{n}{2} \rfloor$, па оназад до оног на позицији 0 се спушта наниже колико је потребно тако да хип коме је корен на тој позицији буде исправан.

У примеру, се елемент 2 мења са својим већим сином (то је елемент 9).

```
          7
      9       8
   2    4  6   5
 1   3
```

```
7 9 8 2 4 6 5 1 3
```

Након тога се мења са елементом 3 (он је сада већи син елемента 2).q

```
          7
      9       8
   3    4  6   5
 1   2
```

```
7 9 8 3 4 6 5 1 2
```

На крају, потребно је спустити још елемент 7. Он се мења са елементом 9 и пошто је након тога већи од своја оба детета, поступак се завршава.

```
          9
      7       8
   3    4  6   5
 1   2
```

```
9 7 8 3 4 6 5 1 2
```

```
// svi elementu u nizu a na pozicijama [0, i) zadovoljavaju uslov hipa
// pomera se element na poziciji i navise tako da nakon toga svi elementi
// na pozicijama [0, i] zadovoljavaju uslov hipa
void pomeri_gore(vector<int>& a, int i) {
  while (i > 0) {
    // roditelj cvora na poziciji i
```

```
    int roditelj = (i - 1) / 2;
    // ako element nije veci od svog roditelja, postupak je zavrsen
    if (a[i] <= a[roditelj])
      break;
    // inace razmenjujemo element sa svojim roditeljem
    swap(a[i], a[roditelj]);
    // i postupak nastavljamo od pozicije roditelja
    i = roditelj;
  }
}


// svi elementi u nizu a na pozicijama od 0 do n-1 zadovoljavaju uslov
// hipa osim eventualno elementa na poziciji i koji moze biti manji od
// svojih potomaka (ali je svakako manji ili jednak svom roditelju,
// ako roditelj postoji) - element se pomera nanize, tako da se na
// pozicijama 0 do n-1 dobije ispravan hip
void pomeri_dole(vector<int>& a, int i, int n) {
  while (true) {
    // pozicija sa kojom treba zameniti element na poziciji i
    // vrednost i govori da element ne treba menjati
    int menjam = i;
    // pozicija levog i desnog potomka cvora i
    int levi = 2 * i + 1;
    int desni = 2 * i + 2;
    // ako element na poziciji i ima levog potomka i ako je manji od njega
    // trebalo bi da se zameni sa levim
    if (levi < n && a[i] < a[levi])
      menjam = levi;
    // medjutim, ako postoji i desni potomak i ako je on jos manji
    // trebalo bi da se zameni sa njim
    if (desni < n && a[menjam] < a[desni])
      menjam = desni;

    // ako je menjam ostalo na vrednosti i znaci da nije manje ni od
    // jednog potomka i posao je zavrsen
    if (menjam == i)
      break;

    // u suprotnom menjamo element sa svojim potomkom
    swap(a[i], a[menjam]);
    // i nastavljamo postupak popravke od tog potomka
    i = menjam;
  }
}

// od niza pravi hip
void napravi_hip(vector<int>& a) {
  // za sve pozicije od 1 do a.size()-1
  for (int i = 1; i < a.size(); i++)
    // na pozicijama [0, i) je vec napravljen hip
    // element na poziciji i jedini ne mora da zadovoljava uslov
    // pa ga pomeramo navise tako da dobijemo hip na pozicijama [0, i]
    pomeri_gore(a, i);
}

// od niza u kojem se nalazi hip pravi sortirani niz
void sortiraj_hip(vector<int>& a) {
```

```
  // za sve pozicije od a.size()-1 unazad do 1
  for (int k = a.size() - 1; k > 0; k--) {
    // najveci element u hipu razmenjujemo sa elementom na poziciji n
    swap(a[0], a[k]);
    // element na vrhu ne mora biti na svom mestu u hipu koji ima k
    // elemenata pa ga pomeramo na dole tako da tih prvih k elemenata
    // cine ispravan hip
    pomeri_dole(a, 0, k);
  }
}

void heap_sort(vector<int>& a) {
  // Od niza pravi hip - binarno drvo u cijem se svakom cvoru nalazi
  // element koji je veci ili jednak od svojih potomaka. Drvo je u niz
  // smesteno po nivoima. Na primer:
  //        13
  //    8        10          13 8 10 4 2 5
  // 4     2   5
  napravi_hip(a);
  // od niza u kojem se nalazi hip pravi sortirani niz
  sortiraj_hip(a);

  // moze i uz pomoc biblioteckih funkcija
  // make_heap(a.begin(), a.end());
  // sort_heap(a.begin(), a.end());
}
```

Оценимо сложеност овог алгоритма. Прво, веома је згодно то што је једна половина елемената низа већ на свом месту. Спуштање елемената низ хип има сложеност $O(\log n)$, па је овде поново у питању неки збир логаритамских сложености и на први поглед може се помислити да ће и овде сложеност бити $\Theta(n \log n)$ – она ће свакако бити $(n \log n)$, али ћемо показати да ће бити нижа тј. $\Theta(n)$. Наиме, није сваки наредни логаритам у збиру за један већи од претходног, како је то био случај приликом померања елемената навише. Ако бисмо разматрали хип у коме су сви нивои потпуно попуњени, постојала би једна позиција са које би се елемент спуштао целом висином хипа, две позиције са којих би се елемент спуштао висином која је за један мања, четири елемента са којих је висина за 2 мања и тако даље. Укупан број корака за хип висине i би био $i + 2(i-1) + 4(i-2) + \ldots 2^{i-1} \cdot 1$ тј.

$$\sum_{k=0}^{i-1} 2^k (i-k).$$

До овог резултата можемо доћи и разматрањем следеће рекурентне једначине. Број корака померања наниже одговара збиру висина свих чворова у дрвету. Нека је $H(i)$ збир свих висина потпуног бинарног дрвета висине i. Тада је $H(i) = 2H(i-1) + i$, јер се потпуно бинарно дрво састоји од два потпуна бинарна дрвета висине $i-1$ и корена висине i. Важи и $H(0) = 0$. Размотајмо ову рекурентну једначину.

$$\begin{aligned}
H(i) &= 2H(i-1) + i \\
&= 2(2H(i-2) + (i-1)) + i = 4H(i-2) + 2(i-1) + i \\
&= 4(2H(i-3) + (i-2)) + 2(i-1) + i = 8H(i-3) + 4(i-2) + 2(i-1) + i \\
&= \ldots \\
&= 2^i H(0) + \sum_{k=0}^{i-1} 2^k (i-k) = \sum_{k=0}^{i-1} 2^k (i-k) \\
&= i \sum_{k=0}^{i-1} 2^k - \sum_{k=0}^{i-1} k 2^k = i(2^i - 1) - ((i-1)-1)2^{(i-1)+1} - 2 = 2^{i+1} - i - 2
\end{aligned}$$

Последње важи јер знамо да је

$$\sum_{k=0}^{n} 2^k = 2^{n+1} - 1, \quad \sum_{k=0}^{n} k2^k = (n-1)2^{n+1} + 2.$$

Прво следи на основу формуле за збир геометријског реда $\sum_{k=0}^{n} x^k$, а друго њеним диференцирањем и множењем са x.

Постоји и лакши начин да се дође до истог резултата. Наиме, једначину $H(i) = 2H(i-1)+i$ можемо записати као $H(i) + i = 2(H(i-1) + (i-1)) + 2$ и онда увести смену $G(i) = H(i) + i$. Зато је $G(i) = 2G(i-1) + 2$ и $G(0) = 0$. Одмотавањем сада добијамо

$$\begin{aligned} G(i) &= 2G(i-1) + 2 \\ &= 2(2G(i-2) + 2) + 2 = 4G(i-2) + 2 \cdot 2 + 2 \\ &= 4(2G(i-3) + 2) + 2 \cdot 2 + 2 = 8G(i-3) + 4 \cdot 2 + 2 \cdot 2 + 2 \\ &= \ldots \\ &= 2^i G(0) + 2\sum_{k=0}^{i-1} 2^k = 2^{i+1} - 2 \end{aligned}$$

Зато је $H(i) = G(i) - i = 2^{i+1} - i - 2$.

Пошто је $G(i)+2 = 2(G(i-1)+2)$, сменом $F(i) = G(i)+2$ би се добило да је $F(i) = 2F(i-1)$ и $F(0) = 2$, па би скоро директно следило да је $F(i) = 2^{i+1}$ и $G(i) = F(i) - 2 = 2^{i+1} - 2$.

Од свих ових комплексних извођења битан нам је само крајњи резултат, а то је да је $H(i) = 2^{i+1} - i - 2$. Пошто је за потпуно дрво број елемената низа $n = 2^i - 1$, важи да је $H(i) = 2(n+1) - i - 2 = 2n - i$ и важи да је сложеност конструкције навише $\Theta(n)$. Дакле, формирање хипа навише има асиптотски бољу сложеност него формирање хипа наниже!

Једно интуитивно објашњење ове чињенице лежи у томе да се код конструкције хипа наниже чак половина елемената (сви листови) може пењати целом висином дрвета, све до корена. Са друге стране, код конструкције навиже, корен је једини чвор који се може спустити целом висином, док се највећи број елемената дрвета јако мало помера.

Завршна фаза сортирања се спроводи вађењем елемената хипа која следи након формирања хипа. Инваријанта ове фазе биће да се у низу на позицијама $[0, i]$ налазе елементи исправно формираног хипа, а да се у делу (i, n) налазе сортирани елементи који су сви већи или једнаки од елемената који се тренутно налазе у хипу. На почетку је $i = n - 1$, па је инваријанта тривијално задовољена (елементи у интервалу $[0, n-1]$ чине исправан хип, док је интервал $(n-1, n)$ празан). У сваком кораку се највећи елемент хипа (елемент на позицији 0) избацује из хипа и додаје на почетак сортираног дела низа (на позицију i). Елемент са позиције i који је овом разменом завршио на врху хипа се помера наниже, све док се задовољи услов хипа. Вредност i се смањује за 1, чиме се исправно одржава граница између хипа и сортираног дела низа (хип је за један елемент краћи, а сортирани део низа је за један елемент дужи). Када се петља заврши тада је $i = 0$, па из инваријанте следи да је низ исправно сортиран (сви елементи на позицијама $(0, n)$ су сортирани и већи или једнаки елементу на позицији 0). Пошто се врше само размене елемената мултискуп елемената низа се не мења.

Види другачија решења овог задатка.

4.6.8 Хеширање и хеш-табеле

Хеширање (енгл. hashing) или директна организација података је један од основних начина за имплементацију неуређених скупова и неуређених мапа. Те структуре података подразумевају чување одређених слогова који се могу претраживати на основу неког њиховог кључног атрибута тј. кључа (код скупова се заправо чувају само кључеви, док се код мапа чувају слогови који поред кључа садрже и вредност придружену том кључу). На пример, у мапи тј. речнку кључеви могу бити ЈМБГ особа, а слогови уз ЈМБГ могу чувати и имена, презимена и још неке додатне информације о особама.

Хеширање омогућава ефикасно додавање нових слогова у структуру података (додавање елемента у скуп тј. придруживање вредности кључу у мапи), ефикасну претрага на основу кључа (проверу да ли елемент постоји

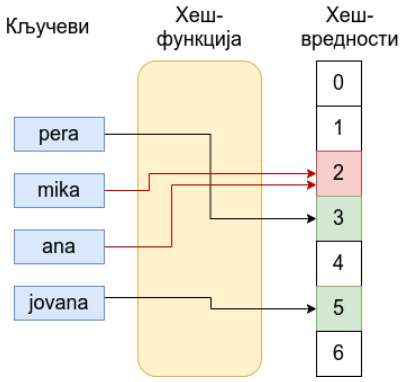
у скупу тј. проналажење вредности придружене кључу у мапи), брисање слогова са датим кључем и измену вредности слогова на основу кључа (измену вредности придружене кључу у мапи). Са друге стране, за разлику од, на пример, уређених дрвета, хеширање не омогућава испис слогова у уређеном (сортираном) редоследу, нити проналажење кључа најближег датом.

Хеширање подразумева коришћење тзв. **хеш-функције** која сваки кључ пресликава у неки цео број (кажемо да је то **хеш-вредност** или **хеш-кôд** кључа). Сви подаци се чувају у **хеш-табели** – низу који на свакој позицији чува један слог (а у неким варијантама и више слогова). Хеш-вредност кључа одређује позицију у табели на којој слог са тим кључем треба да се нађе. Позиције у табели се некада називају и *слотови* (енгл. slots) или *кофе* (енгл. buckets), јер се некада и више слогова чува на једној позицији. Дакле, ако је димензија табеле n, хеш-фунцкија слика вредности из неког произвољно великог скупа кључева у релативно мали скуп позиција у табели $[0, n)$.

Најједноставнији случај хеширања је случај када се може успоставити бијекција између скупа кључева и допустивих позиција у низу. На пример, ако сваком малом слову енглеске абецеде желимо да доделимо вредност у низу од 26 елемената, хеш-функција може бити `h(c) = c-'a'`, која елемент `a` слика у 0, `b` у 1 итд.

Ипак, много реалнија ситуација је да је број потенцијалних кључева доста већи од димензије табеле. Тада функција хеширања не може више бити инјективна тј. морају постојати **колизије** – ситуације у којима више кључева има исту хеш-вредност. Приликом хеширања потребно је да се реше два проблема:

- одабир хеш-функције којом се смањује вероватноћа настанка колизија
- разрешавање колизија када оне настану



Слика 4.3: Пресликавање кључева у хеш-вредности. Кључеви `mika` и `ana` су у колизији, јер се сликају у исту вредност 2

Наравно, на број колизија утиче и величина хеш-табеле, па се хеш-табеле најчешће с временом шире, када се табела у великој мери напуни и када број колизија постане превилики.

4.6.8.1 Избор хеш-функције

Хеш-функција треба да:

- се израчунава брзо,
- минимизује број колизија.

Наиме, очигледно је да је пожељно да се хеш-функција израчунава у сложености $O(1)$, пожељно са малим константним фактором и да је недопустиво користити функције које се споро израчунавају, јер се израчунавање хеш-вредности врши приликом сваког уметања, сваког брисања, сваке промене и сваке претраге кључа.

Да би се смањио број колизија, пожељно је да хеш-функција кључеве слика равномерно (не и насумично) на све позиције у табели (да би се смањила могућност колизија проузрокованих нагомилавањем кључева у неким областима табеле). Многе једноставне хеш-функције не задовају својсто равномерности. На пример, ако би се за смештање ЈМБГ користио низ од 1000 елемената и ако би се хеширање ЈМБГ вршило тако што би се хеш-кôд сваког ЈМБГ одређивао на основу три цифре које одређују годину рођења (на пример, таква хеш-функција би пресликала ЈМБГ 0305987810013 у позицију 987 у низу), Тада бисмо имали значајно нагомилавање на позицијама које почињу деветком тј. нулом и скоро потпуно неискоришћен остали простор у низу.

Ако се не зна унапред редослед обраде кључева, колизије је практично немогуће избећи. Наиме, до првих колизија долази веома брзо, чак иако се се резервише прилично велики меморијски простор за смештање елемената. Може се доказати да је, ако се елементи појављују у насумичном редоследу, ако се користи хеш-функција која равномерно распоређује елементе по табели и ако низ има n елемената, већ након убацивања око $\sqrt{2\ln(2)} \cdot \sqrt{n})$ тј. око $1,18 \cdot \sqrt{n}$ елемената, вероватноћа да убацивање новог елемента изазове колизију већа од 50%. Ово је у тесној вези са такозваним рођенданским парадоксом, који нам каже да је у скупу од само 23 особе вероватноћа да две особе славе рођендан истог дана већа него да све особе славе рођенда различитог дана (у скупу од 70 особа, та вероватноћа је већ преко 99,9%). На пример, ако резервишемо простор од милион (тј. 10^6 елемената), већ након убацивања тек нешто преко хиљаду и сто елемената (тј. $1,18 \cdot 10^3 = \sqrt{2\ln 2 \cdot 10^6}$), тј. већ када је попуњеност низа тек нешто изнад једног промила, имамо значајну вероватноћу да колизије крену да се јављају. Слободно можемо рећи да се колизије појављују и када је низ практично празан.

Дакле, колизије ће се сигурно стално јављати, међутим, ако је дужина низа n, а број различитих кључева m, код добрих хеш-функција вероватноћа да се на неком месту у низу појави више од m/n вредности треба да буде јако мала.

Хеширање се често изводи кроз два корака — први је да се хеш-функцијом кључу придружи произвољни природни број, а други корак је да се онда тај број преслика у неку позицију у низу, најчешће одређивањем остатка при дељењу са дужином низа n. Када је број елемената у низу одређени степен броја 2 своди на операцију издвајање одређеног броја крајњих битова у бинарном запису броја, што је веома ефикасно. Ипак, у том сценарију постоји неколико проблема.

Један потенцијални проблем у том сценарију представља то што у неким случајевима зависности између вредности кључева и дужине низа n могу створити неравномерну расподелу. На пример, ако је низ дужине 1024, и ако су из неког разлога све вредности кључева парни бројеви, половина позиција у низу ће бити неискоришћена (јер ће остатак парног броја при дељењу са 1024, одређен са последњих 10 битова броја, увек бити паран број). Тај проблем је знатно мањи ако је дужина низа прост број, али предност низова чија је дужина степен двојке је брже израчунавање хеш-функције (захваљујући простом издвајању крајњих битова).

Други проблем је то што вредност хеш-функције зависи само од крајњих битова кључа, па се информација коју кључ носи не користи у потпуности. Стога је у првој фази, пре издвајања последњих битова, потребно применити неку функцију чија ће вредност зависити од свих битова кључа. Чест начин је да се примени нека функција облика $h(x) = x \bmod p$, где је p неки прост број (такав да је $n << p << M$, где је n број елемената низа, а M укупан број различитих кључева). Још боље перформансе (али по цену споријег израчунавања) је коришћење функција $(ax + b) \bmod p$ — тиме се избегава проблем који настаје у случају када се функцијом $h(x) = x \bmod p$, хешира пуно кључева који имају исти остатак при дељењу са p.

У случајевима када је кључ сложен (низ вредности или ниска карактера) у првој фази се изврши неки облик агрегације више вредности у једну (та вредност је обично неки 32-битни или 64-битни цео број). То се обично врши применом неке операције попут сабирања или битовске ексклузивне дисјункције на појединачне елементе низа. Алтернативно, могуће је вредности низа схватити као коефицијенте полинома и затим израчунати вредност полинома (на пример, ниску карактера $c_0 \ldots c_{k-1}$ у којој учествује само 26 слова енглеског алфабета је могуће превести у број израчунавањем вредности $c_0 26^{k-1} + \ldots c_{k-1} 26^0$, занемарујући прекорачење).

За сваку хеш-функцију (чак и оних добрих тј. оних које кључеве равномерно распоређују по хеш-табели) може да се деси да постоји неки фамилија кључева која ће проузроковати велики број колизија и лоше перформансе. Решење је понекад *универзално хеширање* (енгл. universal hashing) које подразумева да се хеш-функција која ће се користити бира насумично из унапред задатог скупа добрих хеш-функција (ово је слично насумичном избору пивота код брзог сортирања). Тада је математичко очекивање броја колизија мало, какви год да се кључеви јаве на улазу.

На крају, нагласимо да је теоријски, ако је скуп свих кључева који ће се у програму јавити, могуће одредити *савршену хеш-функцију* која ће бити инјективна и која ће сваки кључ пресликати тачно на једно месту у низу (чија величина одговара броју кључева који се јављају). Тада се колизије не би јављале, међутим, показује се да је сама конструкција такве савршене хеш-функције често рачунаски доста захтевна, па се не исплати.

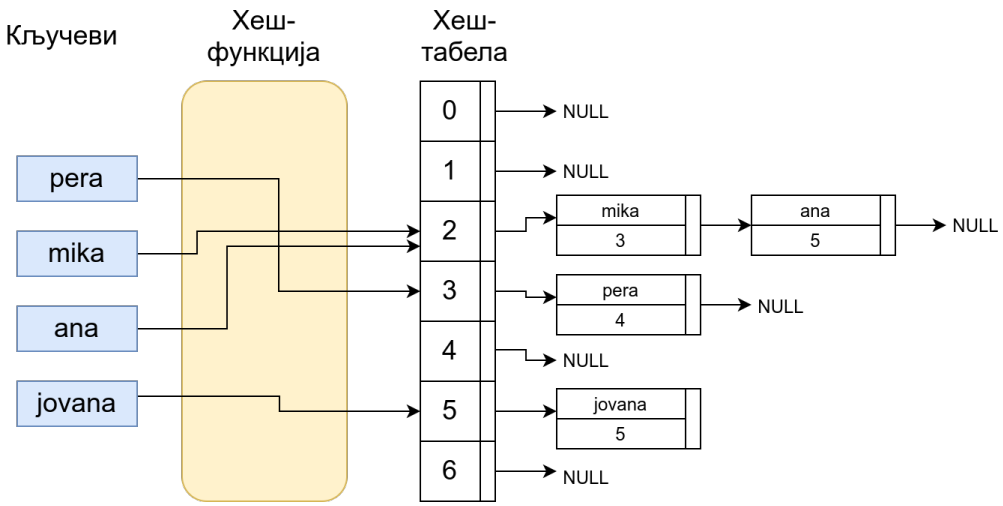
4.6.8.2 Разрешавање колизија

Постоји неколико начина да се разреше колизије. Два основна начина су:

- одвојено уланчавање
- отворено адресирање

Уланчавање

Најједноставнији начин је **одвојено уланчавање** (енгл. separate chaining), које се ређе назива и затворено адресирање или отворено хеширање. Оно подразумева да се на свакој позицији у низу формира посебна повезана листа у коју ће се смештати сви слогови чији се кључ хеш-функцијом пресликава на ту позицију. Приликом претраге слога са датим кључем врши се линеарна претрага листе свих елемената на позицији одређеној применом хеш-функције на тај кључ. Уметање новог слога се врши на почетак те листе (ако дупликати нису допуштени, претходно се линеарном претрагом проверава да ли елемент већ постоји у листи). Брисање елемента се врши из те листе (тако што се прво линеарном претрагом пронађе у листи).



Слика 4.4: Хеширање са уланчавањем

Пошто се у великом броју случајева на некој позицији налази само један елемент листе (или се не налази ни један), да би се смањио један ниво индирекције приликом приступа елементима, често се у низу уместо показивача чувају слогови (уз претпоставку да можемо да разликујемо попуњен и непопуњен слог).

Што листе постају дуже, то њихова линеарна претрага заузима све више и више времена. У најгорем случају (што често подразумева и да је избор хеш-функције направљен јако лоше) сви елементи могу да завршe у једној листи и сложеност свих операција за рад са хеш-табелом постаје линеарна. Ипак, у реалним имплементацијама се с времена на време врши повећање димензије низа (уз поновно распоређивање свих елемената, што подразумева поновно израчунавање свих хеш-вредности). Ова реалокација се врши у складу са геометријском стратегијом (на пример, димензија се повећава дупло приликом сваке реалокације), што гарантује да неће утицати на амортизовану сложеност операција. Реалокације се обично врше када фактор оптерећења (енгл. load factor) који се дефинише као укупан број елемената уписаних у табелу подељен димензијом табеле (бројем слотова тј. кофа) пређе одговарајућу границу (на пример, око 0,75). Други критеријум који може да се користи је дужина листа – када почну да се јављају дугачке листе, добро је повећати табелу. Наравно, уз лош одабир функције хеширања повећање димензије табеле не решава проблем и расипа се велика количина меморије (зато је јако важно да хеш-функција равномерно распоређује кључеве по табели).

Уместо повезаних листа, ланци се могу смештати и у низове који се динамички реалоцирају, па чак и у уређена балансирана бинарна дрвета. Ипак, показује се да ови компликованији приступи не значе обавезно и боље перформансе (јер се код кратких ланаца доста времена на одржавање дрвета – пре свега балансирање).

Отворено адресирање

Други често коришћен начин разрешавања колизија је **отворено адресирање** (енгл. open addressing), које се понекад назива и затворено хеширање. Када се користи отворено адресирање, за смештање података се користи само табела тј. само оригинални низ (за разлику од уланчавања где се подаци чувају у чворовима листе, ван оригиналног низа). У случају уписа података, када се деси колизија, место на које треба да се

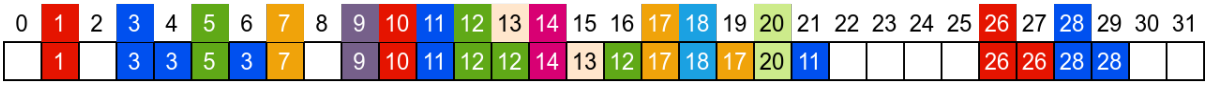
упише нови податак је већ заузето и податак ће тада бити уписан на неко друго место у низу. У зависности од тога како ћемо то друго место одабрати разликујемо:

- линеарно попуњавање (енгл. linear probing)
- квадратно попуњавање (енгл. quadratic probing)
- двоструко хеширање (енгл. double hashing)

Линеарно пробање подразумева да се у случају да је позиција $h(x)$ већ заузета слог који одговара кључу x смешта на наредну позицију у низу (као наредна позиција последњој рачуна се прва), ако је она слободна. Ако није, гледа се њој наредна позиција и тако даље. Дакле, ако је позиција p заузета, проверава се позиција $(p+1) \bmod n$. Уместо вредности 1, може се користити и нека друга вредност k (мада се тиме не мења значајно сложеност). Дакле, позиција на коју ће се елемент x уписати је прва слободна позиција у низу који започиње на позицији $h(x)$ и где се сваки наредна позиција у низу добија од претходне применом функције $(p+k) \bmod n$. Низ позиција је, дакле, $h(x)$, $(h(x)+k) \bmod n$, $(h(x)+2k) \bmod n$ итд. Елемент тог низа који је i-ти по реду (ако се броји од нуле) се налази на позицији $(h(x)+ik) \bmod n$. Када је вредност $k = 1$, тада се редом испитују узастопни елементи табеле, што има јако добре перформансе у односу на кеш-меморију савременог хардвера (много боље него било која друга техника разрешавања колизија) и зато се прилично често користи. Ако се користи вредност k различита од 1, пожељно је да је она узајамно проста са димензијом табеле n, да би низ $(h(x)+ik) \bmod n$ редом пролазио кроз све елементе табеле.

Поново се, дакле, формирају ланци елемената са истом хеш-вредношћу, али су они овај пут не смештају у одвојеним листама, већ у самој табели. Додатно, ова схема узрокује и појаву секундарних колизија. Наиме, елемент x који се смести на позицију $(h(x)+k) \bmod n$ може заузети место неком елементу који има баш ту хеш-вредност, па се и он мора сместити на неку каснију позицију у низу (то зовемо *секундарним колизијама*).

Основна мана линеарног попуњавања је стварање дугачких ланаца и *груписање* тј. *нагомилавање* (енгл. clustering) слогова у одређеним деловима табеле, што може знатно деградирати перформансе. Наиме, када мало дужи ланци крену да се формирају, повећава се вероватноћа настанка секундарних колизија на позицијама тих ланаца, што даље продужује ланце и ствара се зачарани круг. На слици је приказан ефекат груписања. За сваки елемент у низу приказана је његова хеш-вредност. Види се постојање неколико дугачких ланаца и многи елементи су прилично удаљени од своје оригиналне позиције. Претрага за неким кључем који има хеш-вредност 13 захтева обилазак свих елемената, све до прве празне позиције (то је позиција 22). Претрага другог унетог кључа који има хеш-вредност 11 захтева обилазак свих елемената од позиције 11 до позиције 21.



Слика 4.5: Ефекат груписања елемената

Претрага се врши по истом принципу – претражује се тај низ позиција све док се или не наиђе на слог са траженим кључем или не наиђе на празну позицију (када закључујемо да кључ не постоји у табели).

Брисање је проблематичнија операција, јер се брисањем елемента може избрисати неки елемент из средине ланца, што може узроковати да неки каснији елементи у ланцу не буду пронађени (на, пример, ако се три кључа која имају хеш-вредност $h(x)$ упишу редом на позиције $h(x)$, $h(x)+1$ и $h(x)+2$, и ако се затим обрше слог са позиције $h(x)+1$, кључ који је на позицији $h(x)+2$ неће моћи да се пронађе у табели). Зато је приликом брисања потребно пребацити неки каснији елемент из ланца на позицију обрисаног слога (а затим на исти начин попунити и те позиције са којих су елементи пребачени). Једноставнија алтернатива је да се позиције са којих је обрисан слог означе као обрисане (а не као празне тј. слободне) и да се уметање и претрага модификују тако да узму у обзир и постојање таквих ћелија. Мана овог приступа је то што могу да троше велику количину меморије, као и то што ланци могу постати јако дугачки услед великог броја обрисаних ћелија.

Смањивање нагомилавања елемената може се постићи ако се уместо линеарног попуњавања $(h(x)+ik) \bmod n$ користи квадратно пробање $(h(x)+ik_1+i^2k_2) \bmod m$ или двоструко хеширање, када се користе две независне хеш-функције h_1 и h_2 и када се наредна позиција одређује применом друге хеш-функције $(h_1(x)+ih_2(x)) \bmod m$. Двоструко хеширање у потпуности спречава

Код отвореног адресирања је веома важно да се фактор оптерећења табеле (количник броја унетих слогова и димензије табеле) стално држи испод вредности 1. Када фактор оптерећења нарасте, врши се реалокација

табеле уз поновно хеширање и уметање свих слогова. Да би се перформансе одржале, пожељно је да повећање димензије врши у складу са неком геометријском стратегијом (на пример, да се сваки пут табела повећа два пута).

Задатак: Мапа

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Задатак се лако може решити коришћењем библиотечке имплементације асоцијативног низа (било уређене, било неуређене).

```
#include <iostream>
#include <unordered_map>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  unordered_map<string, int> map;
  char c;
  while (cin >> c) {
    if (c == 'w') {
      string k; int v;
      cin >> k >> v >> ws;
      map[k] = v;
    } else if (c == 'r') {
      string k;
      cin >> k >> ws;
      auto it = map.find(k);
      if (it != map.end())
        cout << it->second << '\n';
      else
        cout << "-" << '\n';
    }
  }
  return 0;
}
```

Задатак можемо решити и ручном имплементацијом хеш-табеле уз разрешавање колизија коришћењем одвојеног уланчавања. У језику C++ можемо употребити библиотечку имплементацију функције хеширања (то је функција `hash` доступна кроз заглавље `functional`, која је дефинисана за разне типове кључева, укључујући и тип `string`). Једноставности ради, табелу можемо имплементирати коришћењем колекције типа `vector`.

```
// cvor liste
struct cvor {
  string kljuc;
  int vrednost;
  cvor* sledeci;
};

// kreira se novi cvor sa datim atributima
cvor* napravi_cvor(const string& kljuc, int vrednost, cvor* sledeci) {
  cvor* novi = new cvor();
  // potrebno je proveriti da li je alokacija uspela
  novi->kljuc = kljuc;
  novi->vrednost = vrednost;
  novi->sledeci = sledeci;
```

```
  return novi;
}

// hes-tabela sa odvojenim ulancavanjem
vector<cvor*> tabela(1, nullptr);
// hes-funkcija
hash<string> h;
// broj kljuceva u mapi
int broj_kljuceva;

// inicijalizacija tabele na datu velicinu
void inicijalizuj_tabelu(int velicina) {
  tabela.resize(velicina, nullptr);
}

// uvecavanje tabele, da bi se smanjio broj kolizija i skratile liste
void povecaj_tabelu() {
  // nova tabela
  vector<cvor*> nova_tabela(2*tabela.size(), nullptr);
  // prepisujemo sve kljuceve i vrednosti iz originalne tabele, ponovo
  // hesirajuci kljuceve
  for (cvor* c : tabela) {
    for (cvor* cc = c; cc != nullptr; cc = cc->sledeci) {
      int p = h(cc->kljuc) % nova_tabela.size();
      nova_tabela[p] = napravi_cvor(cc->kljuc, cc->vrednost, nova_tabela[p]);
    }
  }
  // razmenjujemo novu i staru tabelu (swap radi brze nego dodela)
  tabela.swap(nova_tabela);
}

void upisi(const string& kljuc, int vrednost) {
  // pozicija u tabeli na koju treba da bude smesten kljuc
  int p = h(kljuc) % tabela.size();

  // proveravamo da li u listi kljuc vec postoji
  for (cvor* c = tabela[p]; c != nullptr; c = c->sledeci)
    if (kljuc == c->kljuc) {
      c->vrednost = vrednost;
      return;
    }

  // dodajemo novi cvor na pocetak liste
  tabela[p] = napravi_cvor(kljuc, vrednost, tabela[p]);
  broj_kljuceva++;

  // faktor opterecenja
  double opterecenje = (double)broj_kljuceva / (double)tabela.size();
  if (opterecenje >= 0.75)
    povecaj_tabelu();
}

// pretraga vrednosti pridruzene datom kljucu
// funkcija vraca informaciju o tome da li je vrednost pronadjena,
// a sama vrednost, ako postoji, se vraca preko povratnog parametra
bool procitaj(const string& kljuc, int& vrednost) {
  // pozicija u tabeli na kojoj treba da se nalazi kljuc
  int p = h(kljuc) % tabela.size();
```

```
  // trazimo kljuc u listi
  for (cvor* c = tabela[p]; c != nullptr; c = c->sledeci)
    if (kljuc == c->kljuc) {
      vrednost = c->vrednost;
      return true;
    }
  return false;
}

// brisanje svih cvorova u tabeli
void obrisi_tabelu() {
  for (cvor* c : tabela) {
    cvor* cc = c;
    while (cc != nullptr) {
      cvor* sledeci = cc->sledeci;
      delete cc;
      cc = sledeci;
    }
  }
}
```

Задатак можемо решити и ручном имплементацијом хеш-табеле уз разрешавање колизија коришћењем отвореног адресирања уз линеарно попуњавање.

```
// cvor u tabeli
struct cvor {
  string kljuc;
  int vrednost;
  bool pun;
};

// hes-tabela sa otvorenim adresiranjem
vector<cvor> tabela;
// hes-funkcija
hash<string> h;
// broj kljuceva u mapi
int broj_kljuceva;

// inicijalizacija tabele na datu velicinu
void inicijalizuj(int velicina) {
  tabela.resize(velicina);
  for (cvor& c : tabela)
    c.pun = false;
}

// pronalazi poziciju datog kljuca u datoj tabeli
int pronadji_poziciju(const string& kljuc, const vector<cvor>& tabela) {
  int p0 = h(kljuc) % tabela.size();
  int p = p0;
  int i = 0;
  while (tabela[p].pun && tabela[p].kljuc != kljuc)
    p = (p0 + i++) % tabela.size();
  return p;
}

// uvecavanje tabele, da bi se smanjio broj kolizija i skratile liste
void povecaj_tabelu() {
  // nova tabela
  vector<cvor> nova_tabela(2*tabela.size());
```

```
  // prepisujemo sve kljuceve i vrednosti iz originalne tabele, ponovo
  // hesirajuci kljuceve
  for (cvor& c : tabela) {
    if (c.pun) {
      int p = pronadji_poziciju(c.kljuc, nova_tabela);
      nova_tabela[p] = c;
    }
  }
  // razmenjujemo novu i staru tabelu (swap radi brze nego dodela)
  tabela.swap(nova_tabela);
}

void upisi(const string& kljuc, int vrednost) {
  // pozicija u tabeli na koju treba da bude smesten kljuc
  int p = pronadji_poziciju(kljuc, tabela);
  if (!tabela[p].pun)
    broj_kljuceva++;
  tabela[p].kljuc = kljuc;
  tabela[p].vrednost = vrednost;
  tabela[p].pun = true;

  // faktor opterecenja
  double opterecenje = (double)broj_kljuceva / (double)tabela.size();
  if (opterecenje >= 0.75)
    povecaj_tabelu();
}

// pretraga vrednosti pridruzene datom kljucu
// funkcija vraca informaciju o tome da li je vrednost pronadjena,
// a sama vrednost, ako postoji, se vraca preko povratnog parametra
bool procitaj(const string& kljuc, int& vrednost) {
  // pozicija u tabeli na kojoj treba da se nalazi kljuc
  int p = pronadji_poziciju(kljuc, tabela);
  if (tabela[p].pun) {
    vrednost = tabela[p].vrednost;
    return true;
  } else
    return false;
}
```

Глава 5

Подели па владај

Основни механизам конструкције алгоритама је тзв. индуктивно-рекурзивна конструкција где се решење проблема своди на решавање потпроблема истог облика, али мање димензије. Веома често је димензија потпроблема за један мања од димензије оригиналног проблема. На пример, низ се разлаже на свој први елемент и низ елемената иза њега (суфикс) или се разлаже на елементе који претходе последњем елементу (префикс) и тај последњи елемент. На пример, у алгоритму сортирања селекцијом (selection sort) на прво место се поставља најмањи елемент низа, а затим се на исти начин обрађују елементи иза њега, док се у алгоритму сортирања уметањем (insertion sort) сортира префикс низа у који се на крају умеће последњи елемент. Ови алгоритми су описани у задатку Сортирање бројева. Осим решавања потпроблема, алгоритам обично садржи кораке потребне да се потпроблем припреми и кораке да се од резултата потпроблема добију резултати полазног проблема. На пример, у алгоритму сортирања селекцијом припремни корак је одређивање позиције минимума и његово довођење на почетак низа, док је у алгоритму сортирања уметањем након обраде потпроблема (сортирања префикса) додатни корак уметање последњег елемента. Ако се, као што је то случај у ова два примера, обрада потпроблема врши било као први или као последњи корак алгоритма (ако не постоји фаза припреме или обраде резултата потпроблема), тада имплементација може бити и итеративна.

- Ако се решава један потпроблем и ако се припрема потпроблема и обрада резултата потпроблема врше у времену $O(1)$, уз уобичајену претпоставку да се излаз из рекурзије такође врши у времену $O(1)$, време решавања задовољава једначину $T(n) = T(n-1) + O(1), T(0) = O(1)$, чије је решење $T(n) = O(n)$.
- Ако се решава један потпроблем и ако се припрема потпроблема и обрада резултата потпроблема врше у времену $O(n)$, уз уобичајену претпоставку да се излаз из рекурзије такође врши у времену $O(1)$, време решавања задовољава једначину $T(n) = T(n-1) + O(n), T(0) = O(1)$, чије је решење $T(n) = O(n^2)$.

Ефикаснија решења се често добијају техником која се назива техника *разлагања*, техника *декомпозиције* или техника *подели-па-владај* (енгл. divide-and-conquer). Њена основна идеја је то да је често ефикасније решавати неколико потпроблема чија је димензија два (или више) пута мања од димензије полазног потпроблема. Таквих проблема може бити два или више. Код изразито једноставних проблема решење је могуће добити и решавањем само једног таквог потпроблема, чиме се добијају изразито ефикасна решења.

Основни примере технике разлагања су сортирање обједињавањем (енгл. merge sort) и брзо сортирање (енгл. quick sort). И ови су алгоритми описани у задатку Сортирање бројева. Такође, алгоритми обраде бинарног дрвета спадају у ову групу (јер су поддрвета потпроблеми који су у случају балансираних дрвета два пута мањи од димензије полазног проблема). Бинарна претрага се такође може убројати у алгоритам овог типа, али, пошто се код ње један потпроблем потпуно занемарује (примењује се одсецање), та техника се понекад назива *смањи па владај* (енгл. decrease and conquer).

Ако су потпроблеми који се решавају једнаке димензије (два пута мање од димензије полазног проблема), добија се нека од следећих једначина (у зависности од тога колико времена је потребно за припрему потпроблема и обједињавање резултата добијених потпроблема). Решења једначина овог типа се могу често добити применом *мастер теореме*, а ми ћемо у наставку резимирати само неколико најкарактеристичнијих.

- Једначина $T(n) = 2T(n/2) + O(n), T(0) = O(1)$ има решење $O(n \log n)$.
- Једначина $T(n) = 2T(n/2) + O(1), T(0) = O(1)$ има решење $O(n)$.
- Једначина $T(n) = 2T(n/2) + O(\log n), T(0) = O(1)$ има решење $O(n)$.

- Једначина $T(n) = 2T(n/2) + O(n \log n)$, $T(0) = O(1)$ има решење $O(n \log^2(n))$.

Треба обратити пажњу на то да ако су потпроблеми стално неравномерне димензије, могуће је да се добије процес који се дегенерише у процес који се описује једначином $T(n) = T(n-1) + O(n)$, $T(0) = O(1)$ и чије је решење $O(n^2)$ (што је, на пример, сложеност најгорег случаја у алгоритму quick sort, ако пивот

У случају алгоритама типа *decrease and conquer*, решава се само један потпроблем и добијају се најчешће једначине следећег типа.

- Једначина $T(n) = T(n/2) + O(n)$, $T(0) = O(1)$ има решење $O(n)$.
- Једначина $T(n) = T(n/2) + O(1)$, $T(0) = O(1)$ има решење $O(\log n)$.

Задатак: Сортирање бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Брзо сортирање (QuickSort)

У сваком кораку алгоритма сортирања један елемент (обично називан *пивот*) се доводи на своје место (пожељно близу средине низа). Да би након тога, проблем могао бити сведен на сортирање два мања подниза, потребно је приликом довођења пивота на своје место груписати све елементе мање или једнаке од њега лево од њега, а све елементе веће од њега десно од њега (ако се низ сортира неопадајуће). То прегруписавање елемената низа, *корак партиционисања* кључни је корак алгоритма брзог сотирања.

Брзо сортирање се може имплементирати на следећи начин. Позив `qsort_(a, l, d)` sortira deo niza $a[l, d]$. Партиционисање се врши техником два показивача. Слична техника је приказана у задацима Двобојка и Тробојка.

Након партиционисање рекурзивно се сортирају лева и десна половина низа. Излаз из рекурзије представља случај када је низ (тј. његов део $a[l, d]$) празан или једночлан (такав низ је већ сортиран).

Анализа сложености. Сложеност најгорег случаја овог алгоритма може бити квадратна тј. $O(n^2)$, ако се стално дешава да пивот дели низ на две неравномерне целине. Ипак, може се доказати да је просечна сложеност овог алгоритма $O(n \log n)$ и у пракси он показује веома добре резултате (за разлику од сортирања обједињавањем не троши се време на померање елемената између помоћног и главног низа).

```
// soritra segment pozicija [l, d] u nizu a
void quick_sort(vector<int>& a, int l, int d) {
  // ako segment [l, d] jedan ili nula elementa on je vec sortiran
  if (l < d) {
    // za pivot uzimamo proizvoljan element segmenta
    swap(a[l], a[l + rand() % (d - l + 1)]);
    // particionisemo niz tako da se u njemu prvo javljaju elementi
    // manji ili jednaki pivotu, a zatim veci od pivota
    // tokom rada vazi [l, k] su manji ili jednaki pivotu
    // (k, i) su veci od pivota, [i, d] su jos neispitani
    int k = l;
    for (int i = l+1; i <= d; i++)
      if (a[i] <= a[l])
        swap(a[i], a[++k]);
    // razmenjujemo pivot sa poslednjim manjim ili jednakim elementom
    swap(a[l], a[k]);
    // rekurzivno sortiramo deo niza levo i desno od pivota
    quick_sort(a, l, k - 1);
    quick_sort(a, k + 1, d);
  }
}
```

```
// sortira niz a
void quick_sort(vector<int>& a) {
  // poziv pomocne funkcije koja u nizu a sortira segment pozicija [0, n-1]
  quick_sort(a, 0, a.size() - 1);
}
```

Види другачија решења овог задатка.

Задатак: Збир k најбољих

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. *Види текст задатка.*

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

QuickSelect

Алгоритам *брзе селекције* (QuickSelect) представља модификацију алгоритма брзог сортирања. Алгоритам брзог сортирања и његова имплементација су приказани у задатку Сортирање бројева. Брза селекција се користи да се низ подели тако да се на првих k места нађе k највећих (или најмањих) елемената низа, да се на местима иза њих налазе елементи мањи (или већи) од њих, при чему је редослед елемената у свакој од те две групе произвољан. Пошто је редослед елемената у групама произвољан, може се добити ефикаснији алгоритам од онога у ком би се сортирао цео низ (јер редослед елемената у свакој групи може бити произвољан).

Алгоритам брзе селекције се заснива на кораку партиционисања, идентичном као у алгоритму брзог сортирања, који у линеарној сложености елементе низа уређује тако да се прво у низу нађу елементи који су мањи од неког датог елемента (тзв. пивота), да се након њих налази тај елемент и да након тога следе елементи који су већи од пивота. Редослед елемената у свакој од ових група је потпуно произвољан. Ако се пивот јавља више пута, остала појављивања пивота могу бити било лево, било десно од пивота (често се узима да се лево од пивота налазе елементи мањи или једнаки од њега, а десно елементи строго већи од њега или обратно). Као код алгоритма брзог сортирања, за партиционисање се могу користити поступци засновани на техници два показивача. Детаљан опис поступака овог типа дат је у задацима Двобојка или Тробојка.

Пошто се у задатку тражи збир k највећих елемената у низу, једноставности ради, претпоставићемо да елементе низа сортирамо у обратном редоследу – желимо да се k највећих елемената низа нађе на његовом почетку.

Осовни алгоритам је рекурзиван и параметар рекурзије су границе l и d и број k, и задатак алгоритма је да део низа на позицијама $[l, d]$ реорганизује тако да на почетку буде k највећих елемената тог дела низа.

Излаз из рекурзије може бити када је k веће или једнако од дужине сегмента $[l, d]$, тј. када је $k \geq d - l + 1$ тада сви елементи низа спадају међу првих k. Ако је у старту $k \leq n$, где је n број елемената низа, тада k никада неће бити строго веће од дужине сегмента $[l, d]$ (али може бити једнако).

Након избора пивота и партиционисања, позната је позиција на којој се пивот налази. Нека је то нека позиција m (важи да је $l \leq m \leq d$).

- Ако је број елемената лево од пивота (вредност $m - l$) већа или једнака k онда је довољно наћи k највећих елемената левог дела низа. Наиме сви елементи у левом делу низа су већи или једанки од пивота и од елемената у десном делу, па се свих k највећих елемената дела низа са позиција $[l, d]$ налазе у левом делу низа, испред пивота. Дакле, потребно је извршити рекурзивни позив за интервал $[l, m-1]$ и број k тј. пронаћи k највећих елемената у делу низа на позицијама $[l, m-1]$.
- У супротном се закључно са пивотом налази $m-l+1$ од k највећих елемената низа и зато је потребно у десном делу одредити још $k-(m-l+1)$ највећих елемената из тог дела, тако да се рекурзивни позив врши за интервал $[m+1, d]$ и број $k-m+l-1$.

Приметимо да у функцији постоји само један рекурзивни позив и то репни, тако да се он може једноставно елиминисати.

Пример. Прикажимо рад овог алгоритма на примеру проналажења 5 највећих елемената низа $5, 9, 6, 3, 10, 13, 1, 7, 8, 14, 2$. Важи да је $k = 5$ и $n = 11$.

- На почетку је $[l, d] = [0, n-1] = [0, 10]$. После партиционисања добија се низ $9, 6, 10, 13, 7, 8, 14, \mathit{5}, 3, 1, 2$, где је пивот 5 завршио на позицији $m = 7$. Пошто је број елемената лево од пивота $m - l = 7$ већи од $k = 5$, рекурзивно проналазимо 5 највећих елемената у делу низа на позицијама у интервалу $[l, m-1] = [0, 6]$.
- Партиционишемо део низа одређен са $[l, d] = [0, 6]$. После партиционисања добија се низ $10, 13, 14, \mathit{9}, 6, 7, 8, \mathbf{5}, \mathbf{3}, \mathbf{1}, \mathbf{2}$, где је пивот 9 завршио на позицији $m = 3$. Овај пут је број елемената лево од пивота $m - l = 3$ строго мањи од $k = 5$. Зато је потребно рекурзивно понаћи $k - (m - l + 1) = 1$ највећи елемент у делу низа на позицијама $[m + 1, d] = [4, 6]$
- Партиционишемо део низа одређен са $[l, d] = [4, 6]$. После партиционисања добија се низ $\mathbf{10}, \mathbf{13}, \mathbf{14}, \mathbf{9}, 7, 8, \mathit{6}, \mathbf{5}, \mathbf{3}, \mathbf{1}, \mathbf{2}$, где је пивот 6 завршио на позицији $m = 6$. Број елемената лево од пивота $m - l = 2$ и он је већи од k, па рекурзивно тражимо $k = 1$ највећи елемент у делу низа на позицијама $[l, m-1] = [4, 5]$.
- Партиционишемо део низа одређен са $[l, d] = [4, 5]$. После партиционисања добија се низ $\mathbf{10}, \mathbf{13}, \mathbf{14}, \mathbf{9}, 8, 7, \mathbf{6}, \mathbf{5}, \mathbf{3}, \mathbf{1}, \mathbf{2}$, где је пивот 7 завршио на позицији $m = 5$. Број елемената лево од пивота $m - l = 1$ и он је једнак k, па рекурзивно тражимо $k = 1$ највећи елемент у делу низа на позицијама $[l, m-1] = [4, 4]$.
- Пошто је дужина $d - l + 1$ интервала $[l, d] = [4, 4]$ једнака $k = 1$, поступак је завршен. Коначно стање низа је $10, 13, 14, 9, 8, 7, 6, 5, 3, 1, 2$. Приметимо да низ није сортиран опадајуће, али смо сигурни да се 5 највећих елемената низа сада налази на његовом почетку.

Доказ коректности. Приметимо и да доказивање заустављања алгоритма није сасвим тривијално. Ако је k у старту веће од дужине низа, алгоритам ће се одмах зауставити. У супротном ће све време извршавања алгоритма важити инваријанта да је $0 \leq k \leq d-l+1$ (у тренутку када се постигне горња једнакост, алгоритам ће се зауставити), док ће се вредност $d - l + 1$ смањивати кроз рекурзивне позиве.

- Ако је $k \leq m - l$, извршиће се рекурзивни позив за $l' = l$, $d' = m - 1$ и $k' = k$. Пошто је $m \leq d$, важи и да је $m < d + 1$. Зато је $d' - l' + 1 = m - 1 - l + 1 = m - l < d - l + 1$.

  Важи и да је $k' \leq d' - l' + 1$, јер је $k' = k \leq m - l = d' - l' + 1$. Важи и да је $0 \leq k' = k$.
- Ако је $k > m - l$, извршиће се рекурзивни позив за $l' = m + 1$, $d' = d$ и $k' = k - (m - l + 1)$. Тада је $d' - l' + 1 = d - (m+1) + 1 = d - m$. Пошто је $l \leq m$, важи да је $d' - l' + 1 = d - m \leq d - l < d - l + 1$.

  Важи и да је $k' \leq d' - l' + 1$. Заменом добијамо да је тај услов еквивалентан $k - (m - l + 1) \leq d - (m + 1) + 1$, тј. $k + l - 1 \leq d$, но то важи, јер је $k \leq d - l + 1$.

  Пошто је $k > m - l$, важи и да је $k \geq m - l + 1$, па је $k' = k - (m - l + 1) \geq 0$.

Из ове анализе јасно је и да ако је $k \leq n$, тада услов заустављања алгоритма (излаз из рекурзије) може бити и када низ постане празан тј. када је $l - d + 1$ постане 0. Такође, услов заустављања би могао бити и то да је $k = 0$.

Анализа сложености. Под претпоставком да ће пивот делити низ на делове који су отприлике једнаке величине, сложеност овог алгоритма се описује једначином $T(n) = T(n/2) + O(n), T(0) = O(1)$, чије је решење $T(n) = O(n)$. Дакле, сложеност алгоритма брзе селекције је линеарна $O(n)$. Пошто се цео низ учитава и чува у меморији и просторна сложеност је $O(n)$.

```
// QuickSelect - odredjujemo najvecih k elemenata niza a tj. niz permutujemo
// tako da se najvecih k elemenata nadju na prvih k pozicija (u proizvoljnom
// redosledu)
void qsortK(vector<int>& a, int l, int d, int k) {
  if (k >= d - l + 1)
    return;

  // niz particionisemo tako da se pivot (element a[l]) dovede na
  // svoje mesto, da ispred njega budu svi elementi koji su veci ili
  // jednaki od njega, a da iza njega budu svi elementi veci od njega
  int m = l;
  for (int t = l+1; t <= d; t++)
    if (a[t] >= a[l])
      swap(a[++m], a[t]);
```

```
  swap(a[m], a[l]);

  if (k <= m - l)
    // svih k elemenata su levo od pivota - obradjujemo deo ispred pivota
    qsortK(a, l, m - 1, k);
  else
    // mozda su neki kod k najvecih iza pivota - obradjujemo deo iza pivota
    qsortK(a, m+1, d, k - (m - l + 1));
}

// QuickSelect - pomocna funkcija zbog lepseg interfejsa
void qsortK(vector<int>& a, int k) {
  qsortK(a, 0, a.size() - 1, k);
}
```

Рачунање збира током партиционисања

Уместо да елементе прво распоредимо тако да k највећих елемената буде на почетку, па тек затим да их сабирамо, функција може бити дефинисана тако да истовремено распоређује елементе и рачуна њихов збир. У овом случају згодно нам је да за излаз из рекурзије прогласимо услов $k = 0$ или услов да је низ празан тј. да је $d - l + 1 = 0$, јер у оба случаја функција треба да врати збир нула, што се веома једноставно програмира (ако би излаз из рекурзије био да k постане једнако дужини низа, приликом излаза из рекурзије, требало би израчунати збир свих елемената низа, што је мало компликованије).

Пошто је рекурзија репна, она се лако елиминише.

```
// QuickSelect - zbir k najvecih elemenata niza
int zbirKNajvecih(vector<int>& a, int k) {
  int l = 0, d = a.size() - 1;
  int zbir = 0;
  while (k != 0) {
    // niz particionisemo tako da se pivot (element a[l]) dovede na
    // svoje mesto, da ispred njega budu svi elementi koji su veci ili
    // jednaki od njega, a da iza njega budu svi elementi veci od njega
    int m = l;
    for (int t = l+1; t <= d; t++)
      if (a[t] >= a[l])
        swap(a[++m], a[t]);
    swap(a[m], a[l]);

    if (k <= m - l)
      // svih k elemenata su levo od pivota - obradjujemo deo ispred pivota
      d = m - 1;
    else {
      // sabiramo sve elemente zakljucno sa pivotom
      for (int i = l; i <= m; i++)
        zbir += a[i];
      // mozda su neki kod k najvecih iza pivota - obradjujemo deo iza pivota
      k -= m - l + 1;
      l = m + 1;
    }
  }
  return zbir;
}
```

Библиотечка функција

У језику C++ библиотечка функција `nth_element` врши поделу низа тако да се на позицији n нађе елемент који ту и припада у сортираном редоследу, да се испред те позиције нађу елементи који су сви мањи или једнаки од њега, а да се иза те позиције нађу елементи који су сви већи или једнаки од њега. Функцији се

прослеђује итератор на почетак дела низа (вектора) који се обрађује (обично добијен помоћу `begin`), итератор на неку позицију на средини низа и итератор који указује непосредно иза краја низа (обично добијен помоћу `end`). Ако средишњи итератор указује на n-ту позицију у низу након примене функције на тој позицији ће се наћи n-ти по величини елемент, док ће сви елементи лево од њега бити мањи или једнаки од свих елемената десно од њега. Рецимо и да постоји функција `partial_sort` која је слична претходној али уједно елементе испред дате позиције уређује (соритра) по величини, међутим, у овом случају то нам није потребно и тиме би се само непотребно губило време.

```
// niz particionisemo tako da je  k-ti element na svom mestu i da su
// svi elementi ispred njega manji ili jednaki od svih elemenata iza
nth_element(a.begin(), next(a.begin(), k), a.end(), greater<int>());

// odredjujemo i ispisujemo zbir prvih k elemenata transformisanog niza
cout << accumulate(a.begin(), next(a.begin(), k), 0) << endl;
```

Задатак: Сортирање бројева

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Сортирање обједињавањем (MergeSort)

Алгоритам сортирања обједињавањем дели низ на два дела чије се дужине разликују највише за 1 (уколико је дужина низа паран број, онда су ова два дела једнаких дужина), рекурзивно сортира сваки од њих и затим обједињује сортиране половине. За обједињавање је неопходно користити додатни, помоћни низ, а на крају се обједињени низ копира у полазни низ. Излаз из рекурзије је случај једночланог низа (случај празног низа не може да наступи осим ако је полазни низ празан).

Кључна операција у овом алгоритму је операција обједињавања сортираних низова, техником два показивача. Њена имплементација описана је у задатку Обједињавање. На пример, обједињавањем сортираних низова `a` и `b` добија се сортирани низ `c`. Два већ сортирана низа могу се објединити у трећи сортирани низ само једним проласком кроз низове (тј. у линеарном времену $O(m+n)$ где су m и n димензије полазних низова).

```
a:  1 3 4 7 9 11                  b:  2 5 8 9 10 12 14
         c: 1 2 3 4 5 7 8 9 9 10 11 12 14
```

Функција сортирања обједињавањем сортира део низа $a[l, d]$, уз коришћење низа tmp као помоћног. Променљива n чува број елемената који се сортирају у оквиру овог рекурзивног позива, а променљива s чува средишњи индекс у низу између l и d. Рекурзивно се сортира $n_1 = \left\lfloor \frac{n}{2} \right\rfloor$ елемената између позиција l и $s-1$ и $n_2 = n - \left\lfloor \frac{n}{2} \right\rfloor$ елемената између позиција s і d. Након тога, сортирани поднизови обједињују се у помоћни низ. Пошто се више не обједињују цели низови, већ делови једног низа, функцију обједињавања морамо мало прилагодити.

Помоћни низ може се пре почетка сортирања алоцирати и користити кроз рекурзивне позиве.

Анализа сложености. Добијена функција сортирања има гарантовану сложеност најгорег случаја $O(n \log n)$, што значи да је много бржа од функција заснованих на сортирању селекцијом или сортирању уметањем чија је сложеност $O(n^2)$.

```
// ucesljava deo niza a iz intervala pozicija [i, m] i deo niza b iz
// intervala pozicija [j, n] koji su vec sortirani tako da se dobije
// sortiran rezultat koji se smesta u niz c, krenuvsi od pozicije k
void merge(vector<int>& a, int i, int m,
           vector<int>& b, int j, int n,
           vector<int>& c, int k) {
  while (i <= m && j <= n)
    c[k++] = a[i] <= b[j] ? a[i++] : b[j++];
  while (i <= m)
    c[k++] = a[i++];
```

```
  while (j <= n)
    c[k++] = b[j++];
}

// sortira deo niza a iz intervala pozicija [l, d] koristeci
// niz tmp kao pomocni
void merge_sort(vector<int>& a, int l, int d, vector<int>& tmp) {
  // ako je segment [l, d] jednoclan ili prazan, niz je vec sortiran
  if (l < d) {
    // sredina segmenta [l, d]
    int s = l + (d - l) / 2;
    // sortiramo segment [l, s]
    merge_sort(a, l, s, tmp);
    // sortiramo segment [s+1, d]
    merge_sort(a, s+1, d, tmp);

    // ucesljavamo segmente [l, s] i [s+1, d] smestajuci rezultat u
    // niz tmp
    merge(a, l, s, a, s+1, d, tmp, l);
    // vracamo rezultat iz niza tmp nazad u niz a
    for (int i = l; i <= d; i++)
      a[i] = tmp[i];

    // moze i pomocu biblioteckih funkcija
    /*
    merge(next(a.begin(), l), next(a.begin(), s+1),
          next(a.begin(), s+1), next(a.begin(), d+1),
          next(tmp.begin(), l));
    copy(next(tmp.begin(), l), next(tmp.begin(), d+1), next(a.begin(), l));
    */
  }
}

// sortira niz a
void merge_sort(vector<int>& a) {
  // alociramo pomocni niz
  vector<int> tmp(a.size());
  // pozivamo funkciju sortiranja
  merge_sort(a, 0, a.size() - 1, tmp);
}
```

Задатак: Број инверзија

Напиши програм који одређује колико у низу има инверзија (позиција $0 \leq i < j < n$, таквих да је $a_i > a_j$.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 10^5$) и затим n целих бројева, сваки у посебном реду.

Излаз: На стандардни излаз исписати само тражени број инверзија.

Пример

Улаз	*Излаз*
5 3 1 4 2 5	3

Решење

Груба сила

Грубом силом се задатак решава тако што се помоћу угнежђених петљи испитају сви парови позиција $0 \leq i < j < n$ и преброје сви случајеви када је $a_i > a_j$ (бројимо елементе филтриране серије). Сложеност овог алгоритма одговара броју парова, а то је $O(n^2)$.

```
long long broj_inverzija(const vector<int>& a) {
  int n = a.size();
  long long broj = 0;
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++)
      if (a[j] < a[i])
        broj++;
  return broj;
}
```

Подели па владај - модификација алгоритма MergeSort

Размотримо како бисмо проблем решили декомпозицијом. Празан и једночлан низ немају инверзија. Ако је низ подељен на две половине, укупан број инверзија једнак је збиру броја инверзија међу елементима прве половине, броја инверзија међу елементима друге половине и броја парова елемената где први елемент припада првој, други елемент припада другој половини и први је већи од другог. Прва два броја можемо одредити рекурзивно и остаје само питање како ефикасно одредити трећи број. Да бисмо добили укупну сложеност $O(n \log n)$ тај проблем је потребно решити у сложености $O(n)$ тако да испитивање свих парова елемената из прве и друге половине не долази у обзир. Задатак би се могао лакше решити ако би прва и друга половина биле сортиране (кључни увид је да сортирање елемената тих половина не мења трећи број). Тада можемо применити технику два показивача и веома слично као у случају обједињавања два сортирана низа одредити жељени трећи број. Уместо да сортирамо половине засебно, можемо алгоритам сортирања интегрисати са бројањем инверзија и проширити инваријанту наше функције (ојачати индуктивну хипотезу) тако да функција враћа број инверзија и уједно и сортира низ. На основу инваријанте, рекурзивни позиви ће сортирати леву и десну половину, а да бисмо је одржали, током одређивања трећег броја вршићемо обједињавање сортираних низова (исто као у алгоритму MergeSort).

Пример. Покажимо на једном примеру како можемо да пребројимо инверзије током обједињавања. Нека је дат низ $1, 3, 5, 4, 7, 6, 2, 8$. Све инверзије у њему су $(3, 2)$, $(5, 4)$, $(5, 2)$, $(4, 2)$, $(7, 6)$, $(7, 2)$ и $(6, 2)$ и има их 7.

Поделом се добијају половине $1, 3, 5, 4$ и $6, 7, 2, 8$. Рекурзивни позиви сортирају половине низа и у првој половини проналазе инверзију $(5, 4)$, а у другој половини инверзије $(7, 6)$, $(7, 2)$ и $(6, 2)$. Недостају још инверзије где је први елемент из прве, а други из друге половине низа (то су инверзије $(3, 2)$, $(4, 2)$ и $(5, 2)$). Сортиране половине су $1, 3, 4, 5$ и $2, 6, 7, 8$. Започнимо обједињавање ових низова.

- На почетку се пореде елементи 1 и 2. Пошто је елемент из леве половине мањи, он се пребацује у резултујући низ. Пошто је друга половина сортирана, знамо да је елемент 1 мањи од свих елемената друге половине, па он учествује у 0 инверзија.
- Након тога се пореде елементи 3 и 2. Овај пут је елемент десне половине 2 мањи од елемента из леве половине 3, па зато њега пребацујемо у резултат. Елемент 2 је мањи од елемента 3, а пошто је лева половина сортирана, мањи је и од свих елемената иза њега. Зато знамо да он учествује у 3 инверзије.
- Након тога се пореде елементи 3 и 6. Елемент 3 је мањи од елемента 6, па се он пребацује у резултат. Пошто се елемент 6 налази на позицији 1 у десној половини и пошто је десна половина сортирана, можемо закључити да елемент 3 учествује у једној инверзији (то је она са елементом 2).
- Након тога се пореде елементи 4 и 6. Елемент 4 је мањи од елемента 6, па се он пребацује у резултат. Пошто се елемент 6 налази на позицији 1 у десној половини и пошто је десна половина сортирана, можемо закључити да елемент 4 учествује у једној инверзији (то је она са елементом 2).
- Након тога се пореде елементи 5 и 6. Елемент 5 је мањи од елемента 6, па се он пребацује у резултат. Пошто се елемент 6 налази на позицији 1 у десној половини и пошто је десна половина сортирана, можемо закључити да елемент 5 учествује у једној инверзији (то је она са елементом 2).
- Пошто су сви елементи из прве половине преписани у резултујући низ, преписују се елементи из друге половине. Преписује се елемент 6 и пошто је он већи од свих елемената из прве половине, он не

учествује ни у једној инверзији. Исто важи и за елементе 7 и 8.

Дакле, приликом обједињавања за сваки елемент можемо број инверзија у којима учествује (наравно, говоримо само о инверзијама између две половине низа).

- Када се у резултујући низ преписује елемент из леве половине низа он је строго већи од свих елемената десне половине који су већ преписани (њихов број се лако одреди на основу вредности десног показивача), а мањи или једнак од осталих елемената десне половине, па је број инверзија у којима он учествује једнак вредности десног показивача (од које треба одузети позицију почетка десне половине низа, ако њени елементи нису смештени од позиције 0). Ако десна половина почиње на позицији $s+1$, тада се број инверзија може одредити као $p_d - s - 1$.
- Када се у резултујући низ преписује елемент из десне половине низа, он је строго мањи од текућег елемента у левој половини и свих елемената иза њега. Дакле, број инверзија у којима учествује може се израчунати тако што се од укупног броја елемената леве половине одузме позиција левог показивача (наравно, опет је потребно у обзир узети и позицију на којој почиње лева половина низа). Ако се лева половина завршава на позицији s онда се број инверзија може израчунати као $s + 1 - p_l$.

Укупан број инверзија се може израчунати било тако што се сабере број појединачних инверзија за сваки елемент из леве половине (тада бројач увећавамо у тренутку када се пребацује елемент из леве половине) или тако што се сабере број појединачних инверзија за сваки елемент из десне половине (тада бројач увећавамо у тренутку када се пребацује елемент из десне половине). Можда је мало једноставније сабрати инверзије свих елемената у десној половини, јер тада у фази преписивања елемената преостале половине (када се једна половина исцрпи) нема потребе за ажурирањем броја инверзија. Заиста, ако су преостали само елементи леве половине, за сваки елемент десне половине је већ израчунат и сабран број инверзија, а ако су препостали само елементи десне половине, пошто су исцрпљени сви елементи леве половине, ти преостали елементи десне половине не учествују ни у једној инверзији. Ипак, пошто рекурзивна функција поред израчунавања броја инверзија треба да сортира низ, преостале елементе морамо преписати у резултат (чак иако број инверзија знамо и раније).

```
long long broj_inverzija(vector<int>& a, int l, int d, vector<int>& b) {
  if (l >= d)
    return 0;
  int s = l + (d - l) / 2;
  long long broj = 0;
  broj += broj_inverzija(a, l, s, b);
  broj += broj_inverzija(a, s+1, d, b);
  int pl = l, pd = s+1, pb = 0;
  while (pl <= s && pd <= d) {
    if (a[pl] <= a[pd])
      b[pb++] = a[pl++];
    else {
      broj += s - pl + 1;
      b[pb++] = a[pd++];
    }
  }
  while (pl <= s)
    b[pb++] = a[pl++];
  while (pd <= d)
    b[pb++] = a[pd++];

  copy(begin(b), next(begin(b), d - l + 1), next(begin(a), l));

  return broj;
}

long long broj_inverzija(const vector<int>& a) {
  vector<int> tmp1(a.size()), tmp2(a.size());
  copy(begin(a), end(a), begin(tmp1));
  return broj_inverzija(tmp1, 0, a.size()-1, tmp2);
}
```

Задатак: Број сегмената у низу целих бројева чији је збир најмање K

Напиши програм који у низу целих бројева (не обавезно позитивних) одређује број сегмената (поднизова узастопних елемената) чији је збир мањи или једнак од датог броја K.

Улаз: Са стандардног улаза се уноси дужина низа n ($1 \leq n \leq 10^5$), а затим у наредном реду n елемената низа (целих бројева између -1000 и 1000, раздвојених размацима). Након тога се уноси број K ($0 \leq K \leq 10^6$).

Излаз: На стандардни излаз исписати тражени број сегмената.

Пример

Улаз	*Излаз*

`6`
`1 2 -3 4 -5 6`
`3`
	`8`

Објашњење

Сегменти чији је збир елемената бар 3 су $[1, 2]$, $[1, 2, -3, 4]$, $[1, 2, -3, 4, -5, 6]$, $[2, -3, 4]$, $[2, -3, 4, -5, 6]$, $[4]$, $[4, -5, 6]$ и $[6]$.

Решење

Подели па владај

Први корак у ефикасном решењу је да се примени техника префиксних збирова и да се задатак преформулише тако да се проналажење сегмента $a_i, \ldots, a_{j-1}$, чији је збир најмање K сведе на проналажење два префиксна збира z_j и z_i таква да је $i < j$ и да је $z_j - z_i \geq K$ (наиме, ако је z_m збир првих m чланова низа a, тада је збир $a_i + \ldots + a_{j-1}$ једнак $z_j - z_i$). Коришћење збирова префикса за брзо израчунавање збирова сегмената низа приказано је у задатку Збирови сегмената.

Напомена. Могућност да низ садржи и негативне елементе доводи до тога да низ префиксних збирова није монотон (па се не може претраживати бинарном претрагом, као, на пример, у задатку Слаткиши за сав новац).

Након примене технике префиксних збирова, можемо приметити велику сличност овог задатка са задатком Број инверзија. Број парова који су у инверзији тј. број парова a_i и a_j, таквих да је $i < j$ и $a_i > a_j$ једнак је броју парова таквих да је $i < j$ и $a_j - a_i < 0$ тј. $a_j - a_i \leq -1$. Дакле бројање инверзија одговара проналажењу броја парова којима је разлика највише -1 (у нашем задатку тражили смо да је разлика најмање K, а алгоритам се јако лако модификује тако да проналази парове код којих је разлика највише K). Стога се за добијање ефикасног решења могу користити исте технике које се употребљавају у задатку Број инверзија.

Задатак је могуће решити техником подели па владај.

У сваком рекурзивном позиву разматраћемо део низа префиксних збирова на позицијама $[l, d]$.

Ако је тај део низа празан или једночлан, у њему не постоји пар различитих елемената чија би разлика могла да буде бар K.

У супротном сегмент $[l, d]$ делимо на две половине и рекурзивно проналазимо број парова у левој половини низа $[l, s]$ и у десној половини низа $[s+1, d]$. На те парове, потребно је још додати "мешовите" парове такве да се z_i налази у левој, а z_j налази у десној половини низа и да је $z_j - z_i \geq K$. Испитивање свих парова збирова где је први елемент из леве, а други из десне половине, довело би до неефикасног алгоритма. Кључни увид је тај да се број "мешовитих" парова не мења ако се сортирају сви збирови у првој и сви збирови у другој половини низа префиксних збирова, а да нам сортирање омогућава да број "мешовитих" парова израчунамо прилично ефикасно. Сортирање може да се уради у склопу рекурзивних позива (чиме ојачавамо индуктивну хипотезу, јер наша функција осим што треба да израчуна број парова сада треба и да сортира низ префиксних збирова). Када су половине сортиране, тада ефикасно можемо пронаћи тражени број парова (на пример, техником два показивача).

Покрећемо алгоритам обједињавања два сортирана низа у трећи, помоћни. Тај алгоритам је описан у задатку Обједињавање. Приликом пребацивања сваког збира z_j из десног дела низа у помоћни низ, израчунаћемо колико постоји збирова z_i из левог дела низа таквих да је $z_j - z_i \geq K$. Пошто је леви низ сортиран, одржаваћемо показивач (индекс) r такав да за свако $l \leq i < r$ важи да је $z_j - z_i \geq K$, а да за свако $r \leq i \leq s$

важи да је $z_j - z_i < K$. Такву позицију r можемо пронаћи кренувши од l и увећавајући r све док је $r \leq s$ и $z_j - z_r \geq K$. Када је r пронађено, знамо да се z_j може упарити са свим елементима на позицијама $[l, r-1]$ (и ни са једним другим) тј. да z_j учествује тачно у $(r-1) - l + 1 = r - l$ парова таквих да је $z_j - z_i \geq K$. Зато бројач парова увећавамо за $r - l$. Приликом преласка на наредни збир z_{j+1}, пошто је и десна половина низа сортирана, позиција r може само да се повећа. Наиме, ако је за свако $l \leq i < r$ важило да је $z_j - z_i \geq K$, тада ће исто важити и за $z_{j+1} - z_i$ (јер се преласком са z_j на $z_{j+1} \geq z_j$ та разлика не може смањити (она може остати иста или се увећати). Зато показивач r не треба враћати назад на позицију l већ га само треба померити на десно до жељене позиције (прве позиције у којој је разлика премала).

Пример. Прикажимо рад овог алгоритма на пример низа $1, 2, -3, 4, -5, 6$ и вредности $K = 3$. Низ префиксних збирова је $0, 1, 3, 0, 4, -1, 5$. Пре рекурзивних позива делимо низ на две половине $0, 1, 3, 0$ и $4, -1, 5$. Први рекурзивни позив израчунава да се у првој половини налази само један пар збирова који има разлику већу или једнаку од $K = 3$ (то је пар $(0, 3)$, који одговара сегменту $[1, 2]$) и сортира тај део низа, док други рекурзивни позив израчунава да се у другој половини такође налази само један тражени пар (то је пар $(-1, 5)$, који одговара сегменту $[6]$) и сортира тај део низа. Након тога, прелазимо на обједињавање сортираних делова $0, 0, 1, 3$ и $-1, 4, 5$.

- На почетку је $i = 0$ и $j = 4$. Поредимо $z_0 = 0$ и $z_4 = -1$. Пошто је елемент -1 мањи, њега пребацујемо у помоћни низ. Пошто је већ $-1 - 0 < 3$, вредност r остаје 0 што значи да не постоји ни један пар у коме учествује збир $z_4 = -1$ (као десни збир z_j).
- Сада је $i = 0$ и $j = 5$. Поредимо елементе $z_0 = 0$ и $z_5 = 4$. Пошто је елемент 0 мањи, њега пребацујемо у помоћни низ.
- Сада је $i = 1$ и $j = 5$. Поредимо елементе $z_1 = 0$ и $z_5 = 4$. Пошто је елемент 0 мањи, њега пребацујемо у помоћни низ.
- Сада је $i = 2$ и $j = 5$. Поредимо елементе $z_2 = 1$ и $z_5 = 4$. Пошто је елемент 1 мањи, њега пребацујемо у помоћни низ.
- Сада је $i = 3$ и $j = 5$. Поредимо елементе $z_3 = 3$ и $z_5 = 4$. Пошто је елемент 3 мањи, њега пребацујемо у помоћни низ.
- Сада је $i = 4$ и $j = 5$. Пребацујемо елемент $z_5 = 4$ у помоћни низ. Пошто је $4 - 0 > 3$ и $4 - 1 > 3$, док је $4 - 3 < 3$, r увећавамо све до вредности $r = 3$ и бројач увећавамо за три пара у којима учествује вредност $z_j = 4$ (то су парови $(0, 4)$, $(0, 4)$ и $(1, 4)$, који одговарају сегментима $[1, 2, -3, 4]$, $[4]$ и $[2, -3, 4]$).
- Сада је $i = 4$ и $j = 6$. Пребацујемо елемент $z_6 = 5$ у помоћни низ. Пошто је $5 - 2 < 3$, вредност r не померамо и бројач увећавамо за три пара у којима учествује вредност $z_j = 5$ (то су парови $(0, 5)$, $(0, 5)$ и $(1, 5)$, који одговарају сегментима $[1, 2, -3, 4, -5, 6]$, $[4, -5, 6]$ и $[2, -3, 4, -5, 6]$).

Дакле, укупно смо нашли 6 "мешовитих" парова, што са по једним паром пронађеним у сваком рекурзивном позиву даје укупно 8 парова.

Анализа сложености. Израчунавање збирова префикса се врши у сложености $O(n)$. Након тога се позива рекурзивна функција чије време извршавања задовољава једначину $T(n) = 2T(n/2) + O(n)$, $T(1) = O(1)$, чије је решење $T(n) = O(n \log n)$.

```
// izracunava broj parova (i, j), takvih da je l <= i < j <= d i da je
// ps[j] - ps[i] <= K, pri tom sortirajuci deo niza ps na pozicijama [l, d]
// koristeci tmp kao pomocni niz
long long broj_segmenata(vector<int>& ps, int l, int d, vector<int>& tmp, int K) {
  // ako je deo niza [l, d] prazan ili jednoclan u njemu nema parova
  if (l >= d)
    return 0;
  // delimo niz na dve polovine
  int s = l + (d - l) / 2;
  // odredjujemo broj parova u delu niza [l, s] i [s+1, d]
  // sortirajuci te delove niza
  long long broj_levo = broj_segmenata(ps, l, s, tmp, K);
  long long broj_desno = broj_segmenata(ps, s+1, d, tmp, K);

  // odredujemo broj parova takvih da je i unutar [l, s] a j unutar [s+1, d]
```

```
  long long broj_mesovito = 0;
  // modifikujemo algoritam objedinjavanja dva niza
  int i = l, j = s+1, k = 0;
  // prva pozicija r u delu [l, s] takva da je ps[j] - ps[r] < K
  int r = l;
  while (i <= s || j <= d) {
    if (j > d || (i <= s && ps[i] <= ps[j]))
      tmp[k++] = ps[i++];
    else {
      // pomeramo r tako da je za svaku poziciju p unutar [l, r-1] vazi da
      // je ps[j] - ps[p] >= K
      while(r <= s && ps[j] - ps[r] >= K)
        r++;
      // dakle, ps[j] se moze kombinovati sa tacno r-l elemenata iz dela [l, s]
      broj_mesovito += r-l;
      tmp[k++] = ps[j++];
    }
  }

  // vracamo vrednosti iz pomocnog niza u polazni niz
  copy(begin(tmp), next(begin(tmp), d-l+1), next(begin(ps), l));

  // vracamo ukupan broj parova
  return broj_levo + broj_desno + broj_mesovito;
}

// izracunava broj segmenata niza a ciji je zbir elemenata >= K
long long broj_segmenata(const vector<int>& a, int K) {
  // zbirovi prefiksa niza a
  vector<int> ps(a.size() + 1);
  ps[0] = 0;
  partial_sum(begin(a), end(a), next(begin(ps)));
  // pomocni niz
  vector<int> tmp(ps.size());
  return broj_segmenata(ps, 0, ps.size() - 1, tmp, K);
}
```

Задатак: Силуета града

Са брода се виде зграде на обали велеграда. Дуж обале је постављена координатна оса и за сваку зграду се зна позиција левог краја, висина и позиција десног краја. Написати програм који израчунава силуету града.

Силуета је део-по-део константна функција и одређена је интервалима константности $(-\infty, x_0)$, $[x_0, x_1)$, $[x_1, x_2)$, ..., $[x_{n-1}, +\infty)$, одређеним тачкама поделе $x_0 < x_1 < \ldots < x_{n-1}$ и вредностима 0, h_0, ..., h_{n-2} и h функције на сваком од интервала.

$$\begin{array}{ccccccccccc} & 0 & & h_0 & & h_1 & & \ldots & & h_{n-2} & & 0 & \\ -\infty & & x_0 & & x_1 & & x_2 & \ldots & x_{n-2} & & x_{n-1} & & +\infty \end{array}$$

Подразумевамо да су крајње тачке $-\infty$ и $+\infty$ и да су вредности на тим интервалима једнаке нули. Дакле, део-по-део константна функција се може представити помоћу n тачака $x_0, \ldots x_{n-1}$ и $n-1$ вредности $h_0, \ldots, h_{n-2}$. Једноставности ради ми ћемо овакве функције представљати помоћу n уређених парова (x_0, h_0), (x_1, h_1), ..., (x_{n-2}, h_{n-2}) и $(x_{n-1}, 0)$. Дакле, наш алгоритам прима низ уређених тројки који описује појединачне зграде, а враћа низ уређених парова који описује силуету.

Улаз: Са стандардног улаза се учитава број зграда n, а затим, у n наредних линија по три цела броја раздвојена са по једним размаком: леви крај зграде, десни крај зграде и висина зграде.

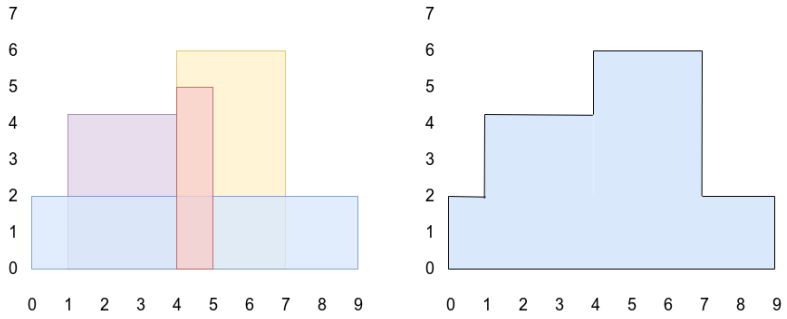
Излаз: На стандардни излаз исписати силуету описану преко низа промена висина.

Пример

Улаз	*Излаз*
4	0 2
1 4 4	1 4
4 5 5	4 6
4 7 6	7 2
0 9 2	9 0

Објашњење

Са леве стране су приказане зграде, а са десне силуета.



Слика 5.1: Зграде и силуета

Решење

Основна индуктивна конструкција подразумева да умемо да нађемо силуету све осим последње зграде и да на крају умемо да ту зграду уклопимо у силуету осталих зграда. Дакле, обилазимо зграду по зграду и једну по једну убацујемо у већ одређену силуету претходних зграда.

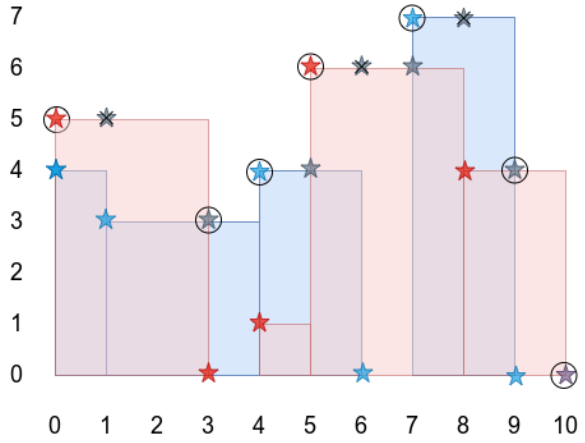
Анализа сложености. Да бисмо зграду уклопили у постојећу силуету потребно нам је линеарно време. Наиме, иако се део силуете у који се зграда интегрише може пронаћи бинарном претрагом (то је део силуете између левог и десног краја зграде), ако се елементи чувају у низу, уметање нових елемената у низ и евентуално брисање постојећих елемената захтева линеарно време. Ако промене чували у повезаној листи уместо у низу, тада бисмо уметање нових промена и брисање промена из силуете могли вршити у константном времену, међутим, не бисмо могли користити бинарну претрагу. У сваком случају, потребно је да прођемо и пронализирамо све елементе постојеће силуете у делу где се умеће нова зграда (између њеног левог и десног краја), а њих може да буде пуно ако је зграда широка и њихова обрада захтева време $O(n)$, где је n текући број елемената силуете. То значи да ће решење бити сложености $T(n) = T(n-1) + O(n), T(1) = O(1)$, што даје сложеност $O(n^2)$, где је n број зграда.

Подели па владај

Проблем можемо ефикасно решити техником подели-па-владај, веома слично алгоритму сортирања обједињавањем (енгл. merge sort). Кључна опаска је то да две силуете можемо објединити за исто време за које можемо објединити једну зграду у силуету (уз претпоставку да ће нова силуета бити смештена у посебном низу). Пошто су резултујуће силуете сортиране можемо их обједињавати веома слично обједињавању два сортирана низа бројева. Тај је алгоритам описан у задатку Обједињавање.

Током рада алгоритма одржаваћемо два индекса (показивача) који одређују текуће елементе у левој тј. у десној силуети. Одржаваћемо и две променљиве и у којима се чува тренутна висина леве тј. десне силуете. Ако је неки од показивача стигао до краја, само ћемо преписивати преостале елементе из силуете која још није потпуно обрађена. У супротном ћемо поредити x координате текућих елемената обе силуете. Ако је x координата текућег елемента у левој силуети мања, ажурираћемо текућу висину леве силуете и увећати леви показивач, ако је x координата текућег елемента у десној силуети мања, ажурираћемо висину десне силуете и увећати десни показивач, а ако су x координате текућих елемената обе силуете једнаке, ажурираћемо висине обе зграде и увећаваћемо оба показивача. Након тога ћемо у резултујућу силуету додавати x координату управо одабраног елемента и придружићемо јој већу од текућих висина две силуете. При том ћемо водити рачуна да у резултат не додајемо нови елемент ако је његова висина једнака висини претходног елемента у резултујућој силуети.

Пример. Прикажимо како се врши обједињавање две силуете на једном примеру.



Слика 5.2: Пример две силуете које се обједињавају

Обједињавамо силуете: $L : (0,5),(3,0),(4,1),(5,6),(8,4),(10,0)$ и $D : (0,4),(1,3),(4,4),(6,0),(7,7),(9,0)$.

	x	Hl	Hd	H	dodaje se	silueta
		0	0			
(0,5), (0,4)	0	5	4	5	(0,5)	(0,5)
(1,3)	1	5	3	5	(1,5)	(0,5)
(3,0)	3	0	3	3	(3,3)	(0,5),(3,3)
(4,1), (4,4)	4	1	4	4	(4,4)	(0,5),(3,3),(4,4)
(5,6)	5	6	4	6	(5,6)	(0,5),(3,3),(4,4),(5,6)
(6,0)	6	6	0	6	(6,6)	(0,5),(3,3),(4,4),(5,6)
(7,7)	7	6	7	7	(7,7)	(0,5),(3,3),(4,4),(5,6),(7,7)
(8,4)	8	4	7	7	(8,7)	(0,5),(3,3),(4,4),(5,6),(7,7)
(9,0)	9	4	0	4	(9,4)	(0,5),(3,3),(4,4),(5,6),(7,7),(9,4)
(10,0)	10	0	0	0	(10,0)	(0,5),(3,3),(4,4),(5,6),(7,7),(9,4),(10,0)

Анализа сложености. Пошто се обједињавање две силуете може урадити у линеарном времену у односу на укупан број зграда у обе силуете, укупна сложеност се израчунава из једначине $T(n) = 2T(n/2) + O(n), T(1) = O(1)$, чије је решење $O(n \log n)$.

Напоменимо да би се имплементација могла убрзати ако би се за обједињавање користио један помоћни вектор (у тренутној имплементацији се одређено време губи на динамичку алокацију елемената резултујућег вектора).

```
// zgrada je odredjena pocetkom a, krajem b i visinom h
struct zgrada {
  int a, b, h;
  zgrada(int a = 0, int b = 0, int h = 0)
    : a(a), b(b), h(h) {
  }
};

// silueta je odredjena nizom promena
// svaka promena je odredjena koordinatom x i visinom h
struct promena {
  int x, h;
  promena(int x = 0, int h = 0)
    : x(x), h(h) {
  }
};
```

```
// integrisemo promenu (x, h) u postojecu siluetu
void dodajPromenu(vector<promena>& silueta, int x, int h) {
  // ako je siuleta prazna ili ako je prethodna visina razlicita od
  // tekuce, dodajemo novu promenu u niz
  if (silueta.empty() || h != silueta.back().h)
      // dodajemo novu promenu u niz
      silueta.emplace_back(x, h);
}

// odredjuje se silueta zgrada na pozicijama [l, d]
vector<promena> silueta(const vector<zgrada>& zgrade, int l, int d) {
  vector<promena> rezultat;

  // silueta koja odgovara jednoj zgradi
  if (l == d) {
    rezultat.emplace_back(zgrade[l].a, zgrade[l].h);
    rezultat.emplace_back(zgrade[l].b, 0);
    return rezultat;
  }

  // određujemo posebno siluete za prvu i drugu polovinu zgrada
  int s = l + (d - l) / 2;
  vector<promena> rezultat_l = silueta(zgrade, l, s);
  vector<promena> rezultat_d = silueta(zgrade, s+1, d);

  // objedinjujemo dve siluete

  // tekući indeksi i visine u levoj i desnoj silueti
  int ll = 0, dd = 0;
  int Hl = 0, Hd = 0;
  // dok god postoji neka neobrađena promena
  while (ll < rezultat_l.size() || dd < rezultat_d.size()) {
    // određujemo novu tačku promene
    int x;
    // ako smo završili sa levom siluetom samo prebacujemo zgrade iz desne
    if (ll == rezultat_l.size()) {
      x = rezultat_d[dd].x; Hd = rezultat_d[dd].h;
      dd++;
    // ako smo završili sa desnom siluetom samo prebacujemo zgrade iz desne
    } else if (dd == rezultat_d.size()) {
      x = rezultat_l[ll].x; Hl = rezultat_l[ll].h;
      ll++;
    } else {
      // određujemo raniju od tekućih promena leve i desne siluete
      int xl = rezultat_l[ll].x;
      int xd = rezultat_d[dd].x;
      if (xl < xd) {
        x = xl; Hl = rezultat_l[ll].h;
        ll++;
      } else if (xl > xd) {
        x = xd; Hd = rezultat_d[dd].h;
        dd++;
      } else {
        x = xl; Hl = rezultat_l[ll].h; Hd = rezultat_d[dd].h;
        ll++; dd++;
      }
    }
```

```
    // veća od dve tekuće visine
    int h = max(Hl, Hd);

    // integrišemo promenu (x, h) u tekuću rezultujuću siluetu
    dodajPromenu(rezultat, x, h);
  }

  return rezultat;
}

// odredjuje se silueta niza zgrada
vector<promena> silueta(const vector<zgrada>& zgrade) {
   return silueta(zgrade, 0, zgrade.size() - 1);
}
```

Задатак: Максимални збир сегмента

Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.

Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.

Решење

Разлагање на потпроблеме

Један начин да решимо овај проблем је заснован на техници разлагања. Декомпозиција нам сугерише да је пожељно да низ поделимо на два подниза једнаке дужине чија решења можемо да конструишемо на основу индуктивне хипотезе (најчешће рекурзивним позивима). Базу и овај пут чини случај празног низа, који садржи само празан сегмент чији је збир нула. Фиксирајмо средишњи елемент низа. Све сегменте низа можемо да групишемо у три групе: сегменте који су у потпуности лево од средишњег елемента, сегменте који су у потпуности десно од средишњег елемента и сегменте који садрже средишњи елемент. Највеће збирове сегмената у првој и у другој групи знамо на основу индуктивне хипотезе. Највећи збир сегмента у трећој групи можемо лако одредити анализом свих сегмената: крећемо од једночланог сегмента који садржи само средишњи елемент и инкрементално се ширимо налево додајући један по један елемент и рачунајући текући максимум, а затим крећемо од максималног сегмента проширеног налево и инкрементално га проширујемо једним по једним елементом надесно, тражећи нови максимум.

Анализа сложености. Ако са n означимо дужину низа $d - l + 1$ и ако време извршавања обележимо са $T(n)$, тада важи да је $T(0) = O(1)$ и да је $T(n) = 2T(n/2) + O(n)$. Наиме, врше се два рекурзивна позива за дупло мање низове, а највећи збир сегмената који обухватају средишњи елемент израчунавамо у времену $O(n)$ (што је прилично очигледно, јер имамо две петље које се укупно извршавају n пута, а чија су тела константне сложености). На основу мастер теореме лако се закључује да је $T(n) = O(n \log n)$. Дакле, овај алгоритам је мање ефикасан од алгоритама сложености $O(n)$, али је и даље прилично употребљив (и сигурно је много бољи од алгоритама грубе силе који су квадратне или кубне сложености).

```
int maksZbirSegmenta(const vector<int>& a, int l, int d) {
  if (l > d)
    return 0;
  int s = l + (d - l) / 2;
  int maks_zbir_levo = maksZbirSegmenta(a, l, s-1);
  int maks_zbir_desno = maksZbirSegmenta(a, s+1, d);
  int zbir_sredina = a[s];
  int maks_zbir_sredina = zbir_sredina;
  for (int i = s-1; i >= l; i--) {
    zbir_sredina += a[i];
    if (zbir_sredina > maks_zbir_sredina)
      maks_zbir_sredina = zbir_sredina;
  }
  zbir_sredina = maks_zbir_sredina;
  for (int i = s+1; i <= d; i++) {
```

```
    zbir_sredina += a[i];
    if (zbir_sredina > maks_zbir_sredina)
      maks_zbir_sredina = zbir_sredina;
  }
  return max({maks_zbir_levo, maks_zbir_desno, maks_zbir_sredina});
}

int maksZbirSegmenta(const vector<int>& a) {
  return maksZbirSegmenta(a, 0, a.size() - 1);
}
```

Ојачање индуктивне хипотезе

На идеји декомпозиције можемо изградити и ефикаснији алгоритам. Кључни увид је да се највећи збир сегмента око средњег елемента може добити као збир највећег суфикса низа лево од тог елемента и највећег префикса низа десно од тог елемента. Можемо ојачати индуктивну хипотезу и уместо да префикс и суфикс рачунамо у петљи, у линеарном времену, можемо претпоставити да за обе половине низа префикс и суфикс добијамо као резултат рекурзивног позива. То нам је довољно да одредимо максимални збир функције, али морамо „вратити дуг'' и наша функција сада поред максималног збира сегмента мора израчунати и максимални збир префикса и максимални збир суфикса целог низа. Максимални збир префикса целог низа је већи број од максималног збира префикса левог дела и од збира целог левог дела и максималног збира префикса десног дела. Слично, максимални збир суфикса целог низа је већи од максималног збира суфикса десног дела и од збира максималног збира суфикса левог дела и целог десног дела. Зато је неопходно додатно ојачати индуктивну хипотезу и током рекурзије рачунати и збир целог низа.

Претпоставимо да је P збир максималног префикса низа, да је S збир максималног суфикса низа, да је Z збир целог низа и да је M максимални збир сегмента низа. Означимо индексом l те статистике у левој половини низа и индексом d те статистке у десној половини низа. Тада важи:

$$\begin{aligned} Z &= Z_l + Z_d \\ P &= \max(P_l, Z_l + P_d) \\ S &= \max(S_d, S_l + Z_d) \\ M &= \max(M_l, M_d, S_l + P_d) \end{aligned}$$

Анализа сложености. Једначина која описује ову рекурзију је $T(n) = 2T(n/2) + O(1)$, па је сложеност овог решења линеарна тј. $O(n)$.

```
void maksZbirSegmenta(const vector<int>& a, int l, int d,
                      int& zbir, int& maks_zbir,
                      int& maks_prefiks, int& maks_sufiks) {
  if (l == d) {
    zbir = maks_zbir = maks_prefiks = maks_sufiks = a[l];
    return;
  }
  int s = l + (d - l) / 2;
  int zbir_levo, maks_zbir_levo, maks_sufiks_levo, maks_prefiks_levo;
  maksZbirSegmenta(a, l, s,
                   zbir_levo, maks_zbir_levo,
                   maks_prefiks_levo, maks_sufiks_levo);
  int zbir_desno, maks_zbir_desno, maks_sufiks_desno, maks_prefiks_desno;
  maksZbirSegmenta(a, s+1, d,
                   zbir_desno, maks_zbir_desno,
                   maks_prefiks_desno, maks_sufiks_desno);
  zbir = zbir_levo + zbir_desno;
  maks_prefiks = max(maks_prefiks_levo, zbir_levo + maks_prefiks_desno);
  maks_sufiks = max(maks_sufiks_desno, maks_sufiks_levo + zbir_desno);
  maks_zbir = max({maks_zbir_levo, maks_zbir_desno,
                   maks_sufiks_levo + maks_prefiks_desno});
```

```
}

int maksZbirSegmenta(const vector<int>& a) {
  int zbir, maks_zbir, maks_prefiks, maks_sufiks;
  maksZbirSegmenta(a, 0, a.size() - 1,
                   zbir, maks_zbir, maks_prefiks, maks_sufiks);
  return maks_zbir;
}
```

Задатак: Најближи пар тачака

У датом скупу тачака у равни одредити колико је растојање између две тачке које су међусобно најближе.

Улаз: Са стандардног улаза се уноси број тачака n ($1 \leq n \leq 50000$), а затим у наредних n редова координате тачака (два цела броја између -10^9 и 10^9, раздвојена размаком).

Излаз: На стандардни излаз исписати тражено растојање, заокружено на пет децимала.

Пример

Улаз	*Излаз*
5 0 0 0 2 2 0 2 2 1 1	1.41421

Решење

Груба сила

Решење грубом силом подразумева испитивање свих парова тачака и сложеност му је $O(n^2)$.

```
struct Tacka {
  int x, y;
};

double rastojanje(const Tacka& t1, const Tacka& t2) {
  double dx = t1.x - t2.x;
  double dy = t1.y - t2.y;
  return sqrt(dx*dx + dy*dy);
}

double najblizeTacke(vector<Tacka>& tacke) {
  int n = tacke.size();
  double d = numeric_limits<double>::max();
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++)
      d = min(d, rastojanje(tacke[i], tacke[j]));
  return d;
}
```

Декомпозиција

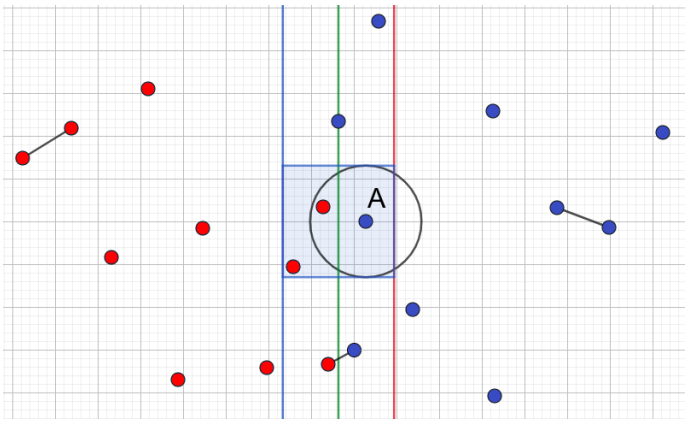
Један начин да се до решења дође ефикасније је да се примени декомпозиција.

Базни случај представља ситуација у којој имамо мање од четири тачке, јер њих не можемо поделити у две половине у којима постоји бар по један пар тачака (а ако у скупу немамо бар 2 тачке, најмање растојање није јасно дефинисано). У том случају решење налазимо поређењем растојања свих парова тачака (пошто је тачака мало, овај корак је сложености $O(1)$).

Скуп тачака можемо једном вертикалном линијом поделити на две отприлике истобројне половине. Ако тачке сортирамо по координати x, вертикална линија може одговарати координати средишње тачке. Рекурзивно одређујемо најмање растојање у првој половини (те тачке се налазе лево од вертикалне линије или евентуално на њој) и у другој половини (те тачке се налазе десно од вертикалне линије или евентуално на њој). Најближи пар је такав да су (1) обе тачке у левој половини, (2) обе тачке у десној половини или (3) једна тачка је у левој, а друга у десној половини. За прва два случаја већ знамо решења (на основу резултата рекурзивних позива) и остаје да се размотри само трећи.

Нека је d_l минимално растојање тачака у левој половини, d_r минимално растојање тачака у десној половини, а d мање од та два растојања. Ако вертикална линија има x-координату x, тада је могуће одбацити све тачке које су лево од $x - d$ и десно од $x + d$, јер је њихово растојање до најближе тачке из супротне половине сигурно веће од d. Потребно је испитати све преостале тачке, тј. све тачке из појаса $[x - d, x + d]$, проверити да ли међу њима постоји неки пар тачака чије је растојање строго мање од d и вредност d ажурирати на вредност најмањег растојања таквог пара тачака. Ако су тачке насумично распоређене, реално је очекивати да ће већина тачака бити ван тог појаса. Међутим, проблем је то што у најгорем случају у појасу може бити пуно тачака (могуће је чак и да се свих n тачака нађе у том појасу) и ако испитујемо све парове, долазимо у најгорем случају до око $n^2/4$ поређења (ако је пола тачака лево, а пола десно од линије поделе). Ипак, проверу је могуће организовати тако да се провери само мали број парова тачака.

Једноставности ради ћемо претпоставити да на исти начин разматрамо све тачке унутар појаса $[x-d, x+d]$, без обзира са које стране вертикалне линије се налазе (унапред знамо да је провера тачака које су са исте стране вертикалне линије поделе непотребна, али не може нарушити коректност, док год смо сигурни да се пореде и сви потребни парови тачака са различите стране те линије). Сваку тачку A из појаса је довољно упоредити само са оним тачкама које леже унутар круга са центром у тачки A и полупречником d, јер су све тачке ван тог круга сигурно од тачке A удаљене више од d, што омогућава значајна одсецања. Међутим, припадност кругу није једноставно проверити и зато уместо њега можемо разматрати квадрат странице дужине $2d$, чији се центар налази на правој $y = x$, а на чијој се хоризонталној средњој линији налази тачка A и тачку A ћемо поредити само са тачкама унутар тог квадрата (знамо да су тачке ван тог квадрата сигурно на растојању већем од d). Тиме ће одсецање бити за нијансу мање него у случају круга (јер су неке тачке унутар квадрата на растојању већем од d), али ће детектовање тачака које припадају том квадрату бити веома једноставно – то ће бити све оне тачке из појаса $[x - d, x + d]$, којима је координата y у интервалу $[y_A - d, y_A + d]$.



Слика 5.3: Најближи пар тачака у левом појасу, десном појасу и између појасева. Круг и квадрат који садрже тачке које се пореде са тачком A.

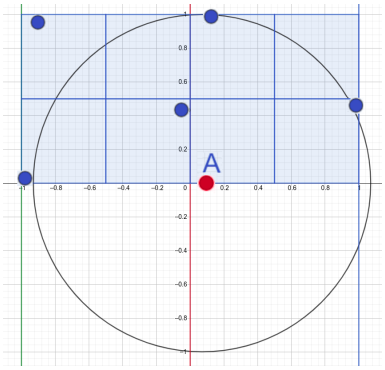
Додатно смањење броја поређења можемо добити ако приметимо да сваки пар обрађујемо два пута (једном док обрађујемо тачке у околини прве, а једном док обрађујемо тачке у околини друге тачке). Можемо једноставно закључити да је довољно сваку тачку A поредити само са оним тачкама које се су изнад ње (или су евентуално на истој висини као она), тј. не у целом квадрату, него само у његовој горњој половини. Дакле, сваку тачку A је потребно упоредити само са тачкама чије x координате леже унутар интервала $[x - d, x + d]$ и чије y координате леже унутар интервала $[y_A, y_A + d]$. Први услов можемо обезбедити тако што пре поређења све тачке из појаса ширине d око вертикалне линије поделе издвојимо у посебан низ (за то нам је потребно $O(n)$ додатне меморије и времена). Други услов ефикасније можемо обезбедити ако све тачке тог помоћног низа сортирамо по координати y (за то нам је потребно време $O(n \log n)$ и затим тачке обрађујемо у неопадајућем редоследу y координата. За сваку тачку A обрађујемо само тачке које се налазе након ње у

сортираном низу и обрађујемо једну по једну тачку све док не наиђемо на тачку чија је координата y већа или једнака од вредности $y_A + d$ (она од тачке A не може бити на мањем растојању од d, а исто важи и за све тачке у низу иза ње).

```
// funkcija pronalazi najblizi par tacaka u delu nizu [l, r]
double najblizeTacke(vector<Tacka>& tacke, int l, int r, vector<Tacka>& pojas) {
  // za manje od 4 tacke, grubom silom odredjujemo najblizi par
  if (r - l + 1 < 4) {
    double d = numeric_limits<double>::max();
    for (int i = l; i < r; i++)
      for (int j = i+1; j <= r; j++)
        d = min(d, rastojanje(tacke[i], tacke[j]));
    return d;
  }

  // delimo niz tacaka sa pozicija [l, r] u dve polovine
  int s = l + (r - l) / 2;

  // rekurzivno pronalazimo najblizi par u levoj i desnoj polovini niza
  double d1 = najblizeTacke(tacke, l, s, pojas);
  double d2 = najblizeTacke(tacke, s+1, r, pojas);

  // najmanje rastojanje svih parova tacaka
  double d = min(d1, d2);

  // pronalazimo tacke u pojasu sirine 2d oko sredisnje linije
  double dl = tacke[s].x - d, dr = tacke[s].x + d;
  int k = 0;
  for (int i = l; i <= r; i++)
    if (dl <= tacke[i].x && tacke[i].x <= dr)
      pojas[k++] = tacke[i];

  // sortiramo tacke u pojasu po y koordinati
  sort(begin(pojas), next(begin(pojas), k),
       [](const Tacka& t1, const Tacka& t2) {
         return t1.y < t2.y;
       });

  // analiziramo sve tacke u pojasu
  for (int i = 0; i < k; i++)
    // svaku tacku poredimo samo sa onim tackama koje su u
    // pravougaoniku iznad nje
    for (int j = i+1; j < k && pojas[j].y - pojas[i].y < d; j++)
      d = min(d, rastojanje(pojas[i], pojas[j]));

  // vracamo najkrace rastojanje
  return d;
}

double najblizeTacke(vector<Tacka>& tacke) {
  // sortiramo tacke po x koordinati
  sort(begin(tacke), end(tacke),
       [](const Tacka& t1, const Tacka& t2) {
         return t1.x < t2.x;
       });
  // pomocni niz (alociramo ga samo jednom, van rekurzije)
  vector<Tacka> pojas(tacke.size());
  return najblizeTacke(tacke, 0, tacke.size() - 1, pojas);
}
```

Анализа сложености. Одредимо сложеност претходног алгоритма. Алгоритам се састоји од два рекурзивна позива за двоструко мању димензију низа тачака и фазе добијања крајњег резултата на основу резултата рекурзивних позива и додатне анализе тачака у појасу $[x - d, x + d]$. Већ смо констатовали да издвајање тачака централног појаса захтева $O(n)$ меморије и времена и да сортирање тих тачака по координати y захтева додатних $O(n \log n)$ корака. Остаје још да се процени сложеност угнежђених петљи у којима се пореде тачке унутар појаса. Иако делује да је сложеност квадратна, елементарним геометријским резоновањем доказаћемо да је сложеност тог корака линеарна тј. $O(n)$ и да се у сваком кораку спољашње петље унутрашња петља може извршити само веома мали број пута (доказаћемо да је тај број извршавања ограничен одозго са 7, мада је у пракси он често и доста мањи од тога и за насумично генерисане тачке та петља се најчешће извршава 0, 1 или евентуално 2 пута).

За сваку тачку A можемо конструисати 8 квадрата димензије $d/2$, као што је приказано на слици (квадрати су уписани у појас $[x - d, x + d]$, у два реда од по четири квадрата и тачка A лежи на доњој ивици доњих квадрата).



Слика 5.4: Најближи пар тачака

Највеће растојање између две тачке унутар неког квадрата се постиже када они леже у његовим наспрамним теменима, а пошто је дужина дијагонале квадрата странице $\frac{d}{2}$ једнака $\frac{d\sqrt{2}}{2} \approx 0,70711 \cdot d$, растојање између сваке две тачке унутар истог квадрата је строго мање од d. Пошто сви квадрати леже било потпуно са леве стране вертикалне линије поделе, било са њене десне стране унутар сваког од квадрата се може наћи највише једна тачка нашег скупа (у супротном би се било са леве, било са десне стране централне линије поделе налазио пар тачака са растојањем строго мањим од d, што је контрадикторно са дефиницијом величине d). То значи да се изнад тачке A може налазити највише 7 тачака које припадају осталим квадратима (сама тачка A већ припада једном од квадрата) и да се све остале тачке које су изнад A налазе и изнад наших квадрата, што значи да им је растојање од A сигурно веће од d (јер им је вертикално растојање веће од d) и њих није потребно разматрати.

Тачке које су са исте стране линије поделе као и тачка A можемо просто прескочити у телу унутрашње петље и тако уштедити на рачунању њиховог растојања од тачке A, али експерименти показују да та уштеда није осетна. Друга могућност за имплементацију је да не чувамо све тачке из појаса у истом скупу, већ да их поделимо у два појаса и да затим да обрадимо прво све тачке из левог појаса гледајући растојања у односу на наредне највише 4 тачке из десног појаса, а затим да обрадимо све тачке из десног појаса гледајући растојања у односу на највише 4 тачке из левог појаса (јер у супротном појасу постоји 4 квадрата димезије $d/2$, за које смо доказали да не могу да садрже две тачке истовремено). Имплементација на тај начин је мало компликованија, а експерименти не указују на значајне добитке.

Дакле, након рекурзивних позива, за добијање коначног резултата је потребно извршити додатних $O(n \log n)$ корака и декомпозиција задовољава рекурентну једначину $T(n) = 2T(n/2) + O(n \log n)$. Решење ове једначине, на основу мастер теореме, је $O(n(\log n)^2)$.

Сортирање обједињавањем

Сложеност се може поправити ако се сортирање по координати y врши истовремено са проналажењем најближег пара тачака, тј. ако се ојача индуктивна хипотеза и ако се претпостави да ће рекурзивни позив вратити растојање између најближе две тачке и уједно сортирати дате тачке по координати y. У кораку обједињавања два сортирана низа обједињујемо у један. То можемо урадити уобичајеним алгоритмом обједињавања,

заснованом на техници два показивача, који обједињавање врши у линеарној сложености. У језику C++ тај алгоритам је доступан и помоћу библиотечке функције `merge`. На тај начин добијамо алгоритам који задовољава једначину $T(n) = 2T(n/2) + O(n)$ и сложености је $O(n \log n)$. Нагласимо да ова оптимизација није револуционарна, али може мало побољшати ефикасност.

На нивоу имплементације, мало побољшање бисмо могли добити и тако што бисмо избегли алокације помоћног вектора унутар рекурзивних позива и код изменити тако да се у сваком рекурзивном позиву користи исти, унапред алоциран помоћни вектор. Још једна могућа оптимизација о којој би се могло размислити је смањивање броја операција кореновања.

```
bool porediX(const Tacka& t1, const Tacka& t2) {
  return t1.x < t2.x;
}

bool porediY(const Tacka& t1, const Tacka& t2) {
  return t1.y < t2.y;
}

// funkcija pronalazi najblizi par tacaka u delu nizu [l, r] i dodatno
// sortira tacke unutar tog dela niza po y koordinati
double najblizeTacke(vector<Tacka>& tacke, int l, int r, vector<Tacka>& pojas) {
  // za manje od 4 tacke, grubom silom odredjujemo najblizi par
  if (r - l + 1 < 4) {
    // odredjujemo najblizi par
    double d = numeric_limits<double>::max();
    for (int i = l; i < r; i++)
      for (int j = i+1; j <= r; j++)
        d = min(d, rastojanje(tacke[i], tacke[j]));
    // sortiramo tacke
    sort(next(begin(tacke), l), next(begin(tacke), r+1), porediY);
    return d;
  }

  // delimo niz tacaka sa pozicija [l, r] u dve polovine
  int s = l + (r - l) / 2;
  int x = tacke[s].x;
  // rekurzivno pronalazimo najblizi par u levoj i desnoj polovini niza
  // sortirajuci te polovine po y koordinati
  double d1 = najblizeTacke(tacke, l, s, pojas);
  double d2 = najblizeTacke(tacke, s+1, r, pojas);

  // najmanje rastojanje svih parova tacaka
  double d = min(d1, d2);

  // objedinjavamo dva sortirane polovine (koristimo niz pojas kao pomocni)
  merge(next(begin(tacke), l), next(begin(tacke), s+1),
        next(begin(tacke), s+1), next(begin(tacke), r+1),
        begin(pojas), porediY);
  copy(begin(pojas), next(begin(pojas), r - l + 1), next(begin(tacke), l));

  // pronalazimo tacke u pojasu sirine 2d oko sredisnje linije
  int k = 0;
  double dl = x - d, dr = x + d;
  for (int i = l; i <= r; i++)
    if (dl <= tacke[i].x && tacke[i].x <= dr)
      pojas[k++] = tacke[i];

  // analiziramo sve tacke u pojasu
  for (int i = 0; i < k; i++)
```

```
    // svaku tacku poredimo samo sa onim tackama koje su u
    // pravougaoniku iznad nje
    for (int j = i+1; j < k && pojas[j].y - pojas[i].y < d; j++)
      d = min(d, rastojanje(pojas[i], pojas[j]));

  return d;
}

double najblizeTacke(vector<Tacka>& tacke) {
  sort(begin(tacke), end(tacke), porediX);
  vector<Tacka> pojas(tacke.size());
  return najblizeTacke(tacke, 0, tacke.size() - 1, pojas);
}
```

Задатак: Множење полинома

Напиши програм који ефикасно одређује производ два полинома.

Улаз: Са стандардног улаза се носе два полинома. За сваки полином је у једној линији дат степен n (цео број између 1 и 50000), а затим у наредној линији коефицијенти (реални бројеви заокружени на једну децималу, раздвојени размацима). Коефицијенти се задају редом, кренувши од слободног члана тј. коефицијента уз x^0, па закључно са коефицијентом уз x^n.

Излаз: На стандардни излаз исписати полином производ, у истом формату у ком су задати и чиниоци, осим што сви коефицијенти треба да буду заокружени на две децимале.

Пример

Улаз	*Излаз*
2 1.0 2.0 3.0 1 1.0 2.0	3 1.00 4.00 7.00 6.00

Решење

Множење можемо извршити класичним алгоритмом множења, који смо приказали у задатку Аритметика над полиномима. Сложеност тог алгоритма је $O(n_1 \cdot n_2)$, где су n_1 и n_2 степени полинома који се множе.

```
// funkcija mnozi dva polinoma p1*p2
vector<double> proizvod(const vector<double>& p1,
                        const vector<double>& p2) {
  int n1 = p1.size(), n2 = p2.size();
  vector<double> proizvod(n1+n2-1, 0);
  for (int i = 0; i < n1; i++)
    for (int j = 0; j < n2; j++)
      proizvod[i+j] += p1[i] * p2[j];
  return proizvod;
}

vector<double> ucitajPolinom() {
  int n;
  cin >> n;
  vector<double> p(n+1, 0.0);
  for (int i = 0; i <= n; i++)
    cin >> p[i];
  return p;
}

int main() {
  vector<double> p1 = ucitajPolinom();
  vector<double> p2 = ucitajPolinom();
```

```
  vector<double> r = proizvod(p1, p2);
  cout << r.size() - 1 << endl;
  for (size_t i = 0; i < r.size(); i++)
    cout << fixed << showpoint << setprecision(2) << r[i] << " ";
  cout << endl;
  return 0;
}
```

Иако се неко време сматрало да је доња граница сложености множења полинома $O(n_1 \cdot n_2)$, Анатолиј Карацуба је 1960. показао да је декомпозицијом могуће добити ефикаснији алгоритам. Претпоставимо да је потребно помножити полиноме $a + bx$ и $c + dx$. Директан приступ подразумева израчунавање $ac + (ad + bc)x + bd$, што подразумева 4 множења. Карацубина кључна опаска је да се исто може остварити само са три множења (на рачун мало већег броја сабирања тј. одузимања, што није критично, јер са сабирање и одузимање обично врши брже него множење, а што важи и за полиноме, јер је сабирање и одузимање полинома операција линеарне сложености). Наиме, важи да је $ad + bc = (a + b)(c + d) - (ac + bd)$. Потребно је, дакле, само израчунати производе ac, bd и $(a + b)(c + d)$, а онда прва два производа употребити по два пута (они су потребни и директно и за израчунавање производа $ad + bc$).

Имајући овај Карацубин "трик" у виду, лако можемо направити алгоритам заснован на декомпозицији. Да би имплементација била једноставнија пре множења полиноме допуњујемо нулама тако да оба имају 2^n коефицијената.

На пример, два полинома степена 3 (са по 4 коефицијената) множимо тако што их представимо помоћу полинома 1 (са по 2 коефицијента).

$$(a_3x^3+a_2x^2+a_1x+a_0)\cdot(b_3x^3+b_2x^2+b_1x+a_0) = ((a_3x+a_2)x^2+(a_1x+a_0))\cdot((b_3x+b_2)x^2+(b_1x+b_0))$$

Сада су "коефицијенти" $A = a_3x + a_2$ и $B = a_1x + a_0$, $C = b_3x + b_2$ и $D = b_1x + b_0$ и врши се множење полинома $Ax^2 + B$ и $Cx^2 + D$. На основу Карацубиног поступка рекурзивно ћемо израчунавати производе $(A + B)(C + D)$, AC и BD, а то су сада потпроблеми двоструко мање димензије од полазне. Аналогно се поступа и када је димензија било који већи степен двојке.

Анализа сложености. Пошто множење задовољава једначину $T(n) = 3T(n/2) + O(n)$. Заиста, врше се три множења двоструко мањих полинома, док се сва потребна сабирања (и у фази припреме рекурзивних позива и у фази обједињавања њихових резултата) врше у сложености $O(n)$. Сложеност алгоритма је зато $O(n^{\log_2 3})$.

Имплементацију најједноставније можемо направити тако да се у сваком рекурзивном позиву сви међурезултати, као и крајњи резултат смештају у посебним векторима. Међутим, таква имплементација је неефикасна и тестови показују да не доприноси побољшању ефикасности наивне процедуре. Кључни проблем је то што се током рекурзије граде вектори у којима се чувају привремени резултати и те алокације и деалокације троше јако пуно времена.

```
// funkcija mnozi dva polinoma p1*p2 sa po 2^k koeficijenata
vector<double> karacuba(const vector<double>& p1,
                        const vector<double>& p2) {
  // broj koeficijenata polinoma
  int n = p1.size();

  // polinome stepena 0 direktno množimo
  if (n == 1)
    return vector<double>{p1[0] * p2[0], 0.0};

  // delimo p1 na dve polovine: a i b
  vector<double> a(n / 2), b(n / 2);
  copy_n(begin(p1), n/2, begin(a));
  copy_n(next(begin(p1), n/2), n/2, begin(b));

  // delimo p2 na dve polovine: c i d
  vector<double> c(n / 2), d(n / 2);
  copy_n(begin(p2), n/2, begin(c));
  copy_n(next(begin(p2), n/2), n/2, begin(d));
```

```
  // Važi:
  //   (ax+b)*(cx+d) = a*c*x^2 + ((a+b)*(c+d) - a*c - b*d)*x + b*d

  // rekurzivno računamo a*c i b*d
  vector<double> ac = karacuba(a, c);
  vector<double> bd = karacuba(b, d);

  // izračunavamo a+b (rezultat smeštamo u vektor a)
  for (int i = 0; i < n/2; i++)
    a[i] += b[i];
  // izračunavamo c+d (rezultat smeštamo u vektor c)
  for (int i = 0; i < n/2; i++)
    c[i] += d[i];

  // izračunavamo (a+b)*(c+d)
  vector<double> adbc = karacuba(a, c);
  // izračunavamo (a+b)*(c+d) - a*c - b*d
  for (int i = 0; i < n; i++)
    adbc[i] -= ac[i] + bd[i];

  // sklapamo proizvod iz delova
  vector<double> proizvod(2*n, 0.0);
  for (int i = 0; i < n; i++) {
    proizvod[n + i] += bd[i];
    proizvod[n/2 + i] += adbc[i];
    proizvod[i] += ac[i];
  }

  // vraćamo rezultat
  return proizvod;
}

// najmanji broj oblika 2^k koji je veci ili jednak od n
int stepenDvojke(int n) {
  int s = 1;
  while (s < n)
    s <<= 1;
  return s;
}

// funkcija mnozi dva polinoma, prosirujuci ih eventualno nulama
vector<double> mnozi_polinome(vector<double>& p1,
                              vector<double>& p2) {
  int s1 = stepenDvojke(p1.size());
  int s2 = stepenDvojke(p2.size());
  int s = max(s1, s2);
  p1.resize(s, 0.0); p2.resize(s, 0.0);
  return karacuba(p1, p2);
}
```

Пажљивија анализа показује да је могуће сву помоћну меморију алоцирати само једном и онда током рекурзије користити стално исти помоћни меморијски простор. Величина потребне помоћне меморије је $4n$ (два пута по n да се сместе полиноми $a+b$ и $c+d$ и још $2n$ да се смести њихов производ). Додатна оптимизација је да се примети да је за мале степене полинома класичан алгоритам бржи него алгоритам заснован на декомпозицији (ово је чест случај код алгоритама заснованих на декомпозицији). Експерименталном анализом се утврђује да се више исплати применити класичан алгоритам кад год је $n \leq 4$.

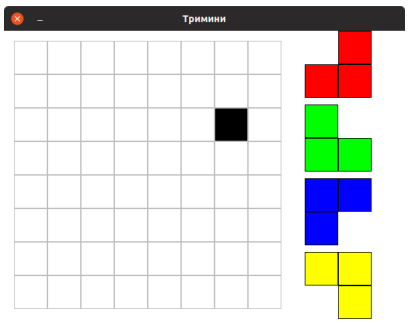
```
// množimo polinome čiji su koeficijenti smešteni u vektorima
```

```
// p1[start1, start1+n) i p2[start2, start2+n)
// i rezultat smeštamo u vektor
// proizvod[start_proizvod, start_proizvod + 2n),
// koristeći pomocni memorijski prostor u vektoru
// pom[start_pom, start_pom + 4n)
void karacuba(int n,
              const vector<double>& p1, int start1,
              const vector<double>& p2, int start2,
              vector<double>& proizvod, int start_proizvod,
              vector<double>& pom, int start_pom) {

  // izlaz iz rekurzije
  if (n <= 4) {
    // klasični algoritam množenja
    for (int i = 0; i < 2*n; i++)
      proizvod[start_proizvod + i] = 0;
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        proizvod[start_proizvod + i+j] +=
           p1[start1 + i] * p2[start2 + j];
    return;
  }

  // Važi: (a+bx)*(c+dx) =
  //           a*c + ((a+b)*(c+d) - a*c - b*d)*x + b*d*x^2

  // Izračunavamo rekurzivno a*c i smeštamo ga u levu polovinu
  // proizvoda
  karacuba(n / 2, p1, start1, p2, start2,
           proizvod, start_proizvod, pom, start_pom);
  // Izračunavamo rekurzivno b*d i smeštamo ga u desnu polovinu
  // proizvoda
  karacuba(n / 2, p1, start1 + n/2, p2, start2 + n/2,
           proizvod, start_proizvod + n, pom, start_pom);

  // Izračunavamo a+b i smeštamo ga u pomoćni vektor (na početak)
  for (int i = 0; i < n/2; i++)
    pom[start_pom + i] =
        p1[start1 + i] + p1[start1 + n/2 + i];
  // Izračunavamo c+d i smeštamo ga u pomoćni vektor (iza (a+b))
  for (int i = 0; i < n/2; i++)
    pom[start_pom + n / 2 + i] =
        p2[start2 + i] + p2[start2 + n/2 + i];

  // Rekurzivno izračunavamo (a+b)*(c+d) i smeštamo ga
  // u pomoćni vektor, iza (a+b) i (c+d)
  karacuba(n / 2, pom, start_pom, pom, start_pom + n / 2,
           pom, start_pom + n, pom, start_pom + 2*n);

  // Izračunavamo (a+b)*(c+d) - (ac + bd)
  for (int i = 0; i < n; i++)
    pom[start_pom + n + i] -=
      proizvod[start_proizvod + i] + proizvod[start_proizvod + n + i];

  // Dodajemo ad+bc na sredinu proizvoda
  for (int i = 0; i < n; i++)
    proizvod[start_proizvod + n/2 + i] += pom[start_pom + n + i];
}
```

```
// funkcija množi dva polinoma p1*p2 sa po 2^k koeficijenata
vector<double> karacuba(const vector<double>& p1,
                        const vector<double>& p2) {
  int n = p1.size();
  // koeficijenti proizvoda
  vector<double> proizvod(2 * n);
  // pomoćni memorijski prostor potreban za realizaciju algoritma
  vector<double> pom(4 * n);
  // vršimo množenje
  karacuba(n, p1, 0, p2, 0, proizvod, 0, pom, 0);
  // vraćamo proizvod
  return proizvod;
}
```

Задатак: Тримини

Нека је дата табла димензије 8×8 на којој недостаје једно поље. Задатак је попунити преостала 63 поља триминима (облицима који се добију када се из квадрата димензије 2×2 избаци једно поље).



Слика 5.5: Тримини

Улаз: Са стандардног улаза се учитавају два цела броја између 0 и 7, раздвојена размаком која представљају координате (врсту и колону) поља које је избачено са табле.

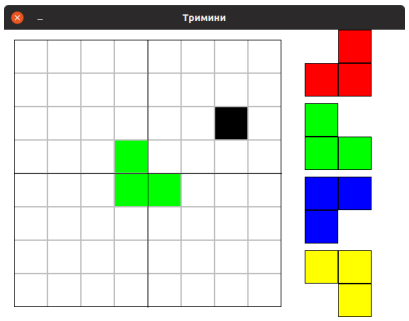
Излаз: На стандардни излаз исписати матрицу карактера која представља таблу прекривену триминима. На месту недостајућег поља треба да буде исписан размак, а тримини се представљају различитим карактерима (на пример, малим словима енглеске абецеде). Решење није јединствено.

Пример

Улаз	*Излаз*
3 5	cceemmoo
	cbbemllo
	dbffnnlp
	ddfan pp
	hhjaartt
	hgjjrrqt
	iggksqqu
	iikkssuu

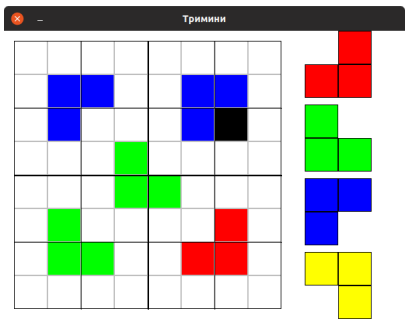
Решење

Задатак се може решити наредним рекурзивним поступком. Табла димензије 8×8 се може разложити на 4 табле димензије 8×8. Постављањем једног тримина можемо постићи да у свакој од те 4 табле недостаје по један тримино.



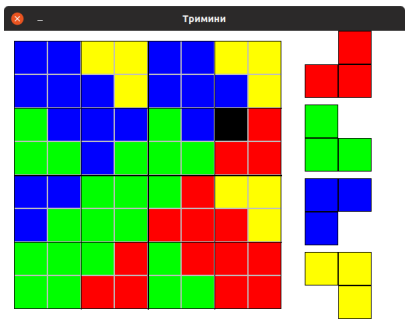
Слика 5.6: Тримини

Након тога, сваки од 4 потпроблема решавамо рекурзивно. Свака табла димензије 4×4 се може разложити на 4 табле димензије 2×2. Постављањем једног тримина можемо постићи да у свакој од те 4 табле недостаје по један тримино.



Слика 5.7: Тримини

На крају, сваку таблу димензије 2×2 којој недостаје једно поље можемо једноставно попунити постављањем одговарајућег тримина.



Слика 5.8: Тримини

Рекурзивној функцији је уз тренутно стање табле и број наредног тримина који се поставља (та променљива се прослеђује по референци, јер се током рекурзије мења и представља улазно-излазни параметар функције) морају проследити аргументи који описују правоугаоник који се тренутно попуњава и положај недостајућег (већ попуњеног) поља. Правоугаоник се описује координатама (бројем врсте и колоне) његовог горњег левог темена и димензијом.

```
// popunjava se kvadrat dimenzije dim cije je gornje levo teme na
// poziciji (v0, k0) i kome je polje na poziciji (v, k) vec popunjeno
// broj je redni broj narednog trimina koji se moze postaviti koji
// se tokom popunjavanja uvecava
void trimini(int tabla[8][8], int dim, int v0, int k0,
             int v, int k, int& broj)
```

```
{
  // ako je dimenzija kvadrata 1x1 u kome je jedno polje popunjeno,
  // tada je ceo kvadrat vec popunjen
  if (dim > 1) {
    // kordinate 4 polja oko sredista kvadrata
    int vsred1 = v0 + dim/2 - 1;
    int vsred2 = v0 + dim/2;
    int ksred1 = k0 + dim/2 - 1;
    int ksred2 = k0 + dim/2;

    // odredjujemo kvadrant kome pripada postojeca rupa
    bool rupaGoreLevo = v <= vsred1 && k <= ksred1;
    bool rupaDoleLevo = v >= vsred2 && k <= ksred1;
    bool rupaGoreDesno = v <= vsred1 && k >= ksred2;
    bool rupaDoleDesno = v >= vsred2 && k >= ksred2;

    // postavljamo trimino u sredisna polja kvadrata
    if (!rupaGoreLevo)
      tabla[vsred1][ksred1] = broj;
    if (!rupaDoleLevo)
      tabla[vsred2][ksred1] = broj;
    if (!rupaGoreDesno)
      tabla[vsred1][ksred2] = broj;
    if (!rupaDoleDesno)
      tabla[vsred2][ksred2] = broj;
    broj++;

    // rekurzivno popunjavamo 4 manja kvadrata

    // uvek ispitujemo da li je u tom kvadrantu stara rupa ili
    // je nastala postavljanjem novog trimina

    // kvadrat gore levo - gornje levo teme mu je na (v0, k0)
    // a nova rupa mu je u njegovom donjem desnom temenu na (vsred1, ksred1)
    if (rupaGoreLevo)
      trimini(tabla, dim/2, v0, k0, v, k, broj);
    else
      trimini(tabla, dim/2, v0, k0, vsred1, ksred1, broj);

    // kvadrat dole levo - gornje levo teme mu je na (vsred2, k0)
    // a nova rupa mu je u njegovom gornjem desnom temenu na (vsred2, ksred1)
    if (rupaDoleLevo)
      trimini(tabla, dim/2, vsred2, k0, v, k, broj);
    else
      trimini(tabla, dim/2, vsred2, k0, vsred2, ksred1, broj);

    // kvadrat gore desno - gornje levo teme mu je na (v0, ksred2)
    // a nova rupa mu je u njegovom donjem levom temenu na (vsred1, ksred2)
    if (rupaGoreDesno)
      trimini(tabla, dim/2, v0, ksred2, v, k, broj);
    else
      trimini(tabla, dim/2, v0, ksred2, vsred1, ksred2, broj);

    // kvadrat dole desno - gornje levo teme mu je na (vsred2, ksred2)
    // i nova rupa mu je u tom gornjem levom temenu na (vsred2, ksred2)
    if (rupaDoleDesno)
      trimini(tabla, dim/2, vsred2, ksred2, v, k, broj);
    else
```

```
      trimini(tabla, dim/2, vsred2, ksred2, vsred2, ksred2, broj);
  }
}

// triminima popunjavamo tablu dimenzije dim kome nedostaje polje (v, k)
void trimini(int dim, int v, int k, int tabla[8][8])  {
  // popunjavamo pocetno polje
  int broj = 0;
  tabla[v][k] = broj++;
  // rekurzivnom funkcijom popunjavamo ostatak table
  trimini(tabla, dim, 0, 0, v, k, broj);
}
```

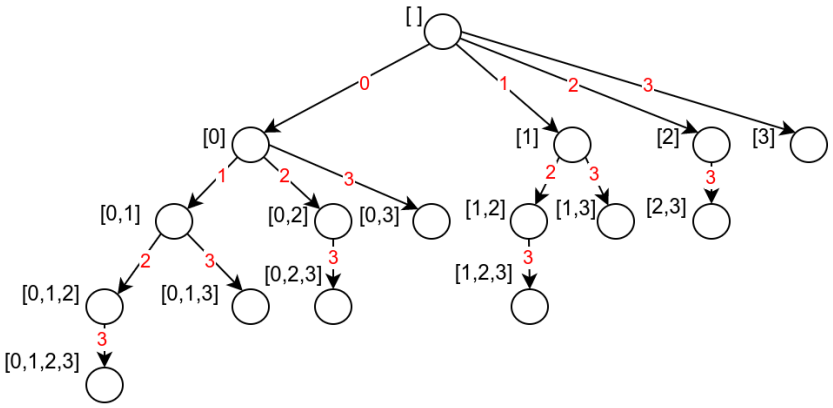
Глава 6

Генерисање комбинаторних објеката

Проблеми се често могу решити исцрпном претрагом (грубом силом), што подразумева да се испитају сви могући кандидати за решења. Предуслов за то је да умемо све те кандидате да набројимо. Иако у реалним применама простор потенцијалних решења може имати различиту структуру, показује се да је у великом броју случаја то простор одређених класичних *комбинаторних објеката*: свих подскупова неког коначног скупа, свих варијација (са или без понављања), свих комбинација (са или без понављања), свих пермутација, свих партиција и слично. У овом поглављу ћемо проучити механизме њиховог систематичног генерисања. Нагласимо да по правилу оваквих објеката има експоненцијално много у односу на величину улаза, тако да су сви алгоритми практично неупотребљиви осим за веома мале димензије улаза.

Објекти се обично представљају n-торкама бројева, при чему се исти објекти могу торкама моделовати на различите начине. На пример, сваки подскуп скупа $\{a_0, \ldots, a_{n-1}\}$ се може представити коначним низом индекса елемената који му припадају. Да би сваки подскуп био јединствено представљен, потребно је да тај низ буде канонизован (на пример, уређен строго растући). На пример, торка $(0, 2, 3)$ једнозначно одређује подскуп $\{a_0, a_2, a_3\}$. Други начин да се подскупови представе су n-торке логичких вредности или вредности 0-1. На пример, ако је $n = 6$, и ако претпоставимо да се битови слева надесно, тада торка 1011 означава скуп $\{a_0, a_2, a_3\}$.

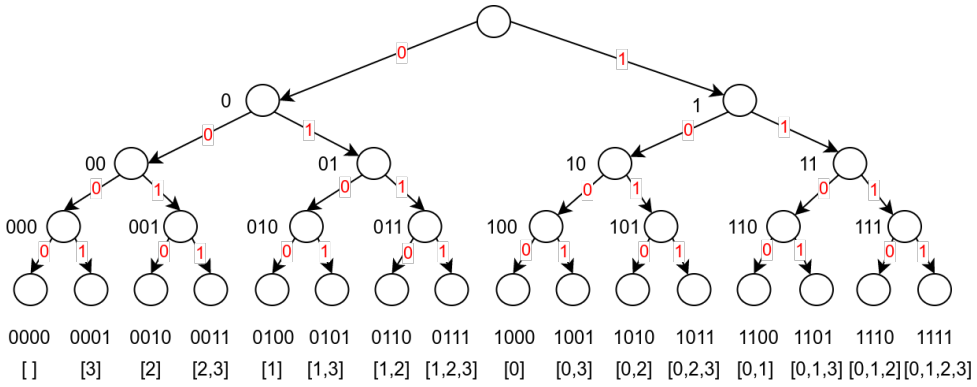
Сви објекти се обично могу представити дрветом и то дрво одговара процесу њиховог генерисања тј. обиласка (оно се не прави експлицитно, у меморији, али нам помаже да разумемо и организујемо поступак претраге). Обилазак дрвета се најједноставније изводи у дубину (често рекурзивно). За прву наведену репрезентацију подскупова дрво је дато на слици 6.1. Сваки чвор дрвета одговара једном подскупу, при чему се одговарајућа торка очитава на гранама пута који води од корена до тог чвора.



Слика 6.1: Сви подскупови четворочланог скупа - сваки чвор дрвета одговара једном подскупу

За другу наведену репрезентацију подскупова дрво је дато на слици 6.2. На почетку се бира да ли ће елемент a_0 бити укључен у подскуп, на наредном нивоу да ли ће бити укључен елемент a_1, затим елемент a_2 и тако даље. Само листови дрвета у којима је за сваки елемент донета одлука да ли припада или не припада подскупу,

одговарају подскуповима, при чему се одговарајућа торка логичких вредности очитава на гранама пута који води од корена до тог чвора.



Слика 6.2: Сви подскупови четворочланог скупа - сваки лист дрвета одговара једном подскупу

Приметимо да оба дрвета садрже 2^n чворова којима се представљају подскупови (у првом случају су то сви чворови дрвета, а у другом само листови).

Приликом генерисања објеката често је пожељно ређати их одређеним редом. С обзиром на то то да се сви комбинаторни објекти представљају одређеним торкама (коначним низовима), природан поредак међу њима је *лексикографски поредак* (који се користи за утврђивање редоследа речи у речнику). Подсетимо се, торка $a_0 \dots a_{m-1}$ лексикографски претходи торци $b_0 \dots b_{n-1}$ акко постоји неки индекс i такав да за свако $0 \leq j < i$ важи $a_j = b_j$ и важи или да је $a_i < b_i$ или да је $i = m < n$. На пример важи да је $11 < 112 < 221$ (овде је $i = 2$, а затим $i = 0$).

На пример, ако подскупове скупа $\{1, 2, 3\}$ представимо на први начин, торкама у којима су елементи уређени растуће, лексикографски поредак би био $\emptyset$, $\{1\}$, $\{1, 2\}$, $\{1, 2, 3\}$, $\{1, 3\}$, $\{2\}$, $\{2, 3\}$, $\{3\}$. Ако бисмо их представљали на други начин, торкама у којима се нулама и јединицама одређује да ли је неки елемент укључен у подскуп, лексикографски редослед би био: 000 ($\emptyset$), 001 ($\{3\}$), 010($\{2\}$), 011($\{2, 3\}$), 100($\{1\}$), 101($\{1, 3\}$), 110($\{1, 2\}$) и 111($\{1, 2, 3\}$).

У наставку ће бити приказано како је могуће набројати све објекте који имају неку задату комбинаторну структуру. У већини задатака могуће је разматрати две врсте решења. Једна група решења је заснована на рекурзивном поступку набрајања објеката, док је друга група решења заснована на проналажењу наредног комбинаторног објекта у односу на неки задати редослед (најчешће лексикорафски).

Задатак: Следећи подскуп

Напиши програм који одређује подскуп скупа бројева $\{1, \dots, n\}$ који у лексикографском редоследу следи непосредно иза датог подскупа. Подскупови су задати у облику строго растуће сортираних низова.

Улаз: Прва линија садржи број n ($1 \leq n \leq 100$), а наредна линија садржи подскуп чији су елементи задати сортирано растуће, раздвојени по једним размаком.

Излаз: На стандардни излаз у једној линији исписати елементе траженог подскупа тј. - ако је учитани подскуп лексикографски највећи.

Пример

Улаз	*Излаз*
`5` `1 2 3 4 5`	`1 2 3 5`

Решење

Напишимо, на пример, лексикографски уређен списак свих подскупова скупа бројева од 1 до 4.

```
-, 1, 12, 123, 1234, 124, 13, 134, 14, 2, 23, 234, 24, 3, 34, 4
```

Можемо приметити да постоје два начина да се дође до наредног подскупа. Анализирајмо ове скупове у истом редоследу, груписане и на основу броја елемената.

```
-  1  12  123  1234
          124
       13  134
       14
   2  23  234
       24
   3  34
   4
```

Један начин је *проширивање* када се наредни подскуп добија додавањем неког елемента у претходни. То су кораци у претходној табели код којих се прелази из једне у наредну колону. Да би добијени подскуп следио непосредно иза претходног у лексикографском редоследу, додати елемент подскупу мора бити најмањи могући. Пошто је сваки подскуп сортиран, елемент мора бити за један већи од последњег елемента подскупа који се проширује (изузетак је празан скуп, који се проширује елементом 1). Једини случај када проширивање није могуће је када је последњи елемент подскупа највећи могући (у нашем примеру то је 4).

Други начин је *скраћивање* када се наредни елемент добија уклањањем неких елемената из подскупа и изменом преосталих елемената. То су кораци у претходној табели код којих се прелази са краја једне у наредну врсту. У овом случају скраћивање функционише тако што се из подскупа избаци завршни највећи елемент, а затим се највећи од преосталих елемената увећа за 1 (он не може бити највећи, јер су елементи унутар сваког подскупа строго растући). Ако након избацивања највећег елемента остане празан скуп, наредна комбинација не постоји.

Подскупове можемо представити динамичким низом који нам омогућва да елементе додајемо и уклањамо са десног краја. У језику C++ можемо употребити вектор (тј. колекцију `vector`).

```
// na osnovu datog podskupa skupa {1, ..., n} određuje leksikografski
// naredni podskup i vraća da li takav podskup postoji
bool sledeciPodskup(vector<int>& podskup, int n) {
  // specijalni slučaj proširivanja praznog skupa
  if (podskup.empty()) {
    podskup.push_back(1);
    // podskup je uspešno pronađen
    return true;
  }

  // proširivanje
  if (podskup.back() < n) {
    // u podskup dodajemo element koji je za 1 veći od
    // trenutno najvećeg elementa
    podskup.push_back(podskup.back() + 1);
    // podskup je uspešno pronađen
    return true;
  }

  // skraćivanje
  // uklanjamo poslednji najveći element
  podskup.pop_back();
  // ako nema preostalih elemenata ne postoji naredni podskup
  if (podskup.empty())
    return false;
  // najveći od preostalih elemenata uvećavamo za 1
  podskup.back()++;
  // podskup je uspešno pronađen
  return true;
}
```

Подскупове можемо чувати и у оквиру низа који је унапред алоциран тако да може да смести елементе највећег подскупа (оног који има тачно n елемената). У том случају је неопходно да одржавамо и промењиву у којој бележимо број елемената подскупа. Пошто се она мења у функцији која одређује наредни подскуп, потребно је пренети је по референци.

```
// na osnovu datog podskupa skupa {1, ..., n} određuje leksikografski
// naredni podskup i vraća da li takav podskup postoji. Tekući podskup
// je smešten u nizu dužine k
bool sledeciPodskup(int podskup[], int& k, int n) {
  // specijalni slučaj proširivanja praznog skupa
  if (k == 0) {
    podskup[k++] = 1;
    return true;
  }

  // proširivanje
  if (podskup[k-1] < n) {
    // u podskup dodajemo element koji je za 1 veći od
    // trenutno najvećeg elementa
    podskup[k] = podskup[k-1] + 1;
    k++;
    return true;
  }

  // skraćivanje
  // izbacujemo najveći element iz podskupa
  k--;
  // ako nema preostalih elemenata, naredni podskup ne postoji
  if (k == 0)
    return false;
  // najveći od preostalih elemenata uvećavamo za 1
  podskup[k-1]++;
  return true;
}
```

Задатак: Сви подскупови лексикографски

Напиши програм који исписује све подскупове скупа $\{0, \ldots, n-1\}$ у лексикографском редоследу.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 15$).

Излаз: На стандардни излаз исписати тражене подскупове, сваки у посебном реду. Сваки се подскуп представља растуће сортираним низом својих елемената.

Пример

Улаз	*Излаз*
3	0
	0 1
	0 1 2
	0 2
	1
	1 2
	2

Решење

Функција за одређивање следећег подскупа у лексикографском редоследу

Задатак се једноставно може решити коришћењем функције за одређивање следећег подскупа у лексикографском редоследу. Тај поступак је описан у задатку Следећи подскуп.

Празан подскуп се проширује елементом 0, а непразан елементом који је за један већи од његовог највећег елемената (ако је његов највећи елемент строго мањи $n-1$). Ако подскуп садржи елемент $n-1$, тада се тај елемент уклања, а највећи елемент из преосталог скупа се увећава за 1. Ако такав елемент не постоји, не постоји ни следећи подскуп у лексикографском редоследу (скуп $\{n-1\}$ је заиста лексикографски највећи).

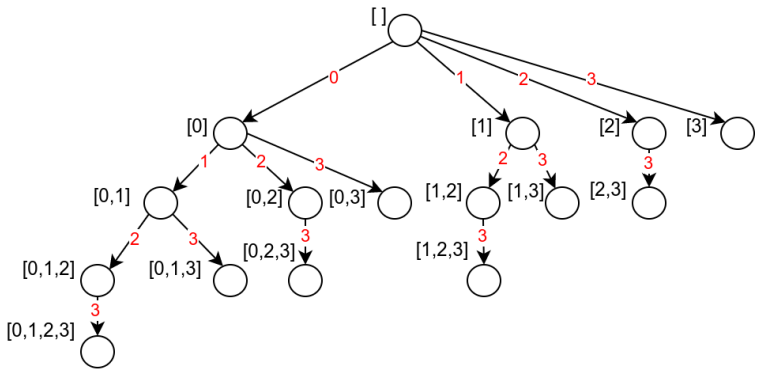
```
// ispis svih podskupova skupa {0, ..., n-1} u leksikografskom redosledu
void obradiSvePodskupove(int n) {
  // krecemo od praznog podskupa
  vector<int> podskup(n);
  // broj elemenata podskupa
  int k = 0;
  // obradjujemo (ispisujemo) tekuci podskup i prelazimo na sledeci,
  // dok god je to moguce
  do {
    obradi(podskup, k);
  } while (sledeciPodskup(podskup, k, n));
}
```

Рекурзивно набрајање свих подскупова у лексикографском поретку

Подскупове скупа $\{0, \ldots, n-1\}$ у лексикографском поретку можемо генерисати рекурзивном функцијом која прима текући подскуп, обрађује га (у нашем случају само исписује), и затим покушава да га на све могуће начине прошири. Сваки подскуп ће бити представљен строго растућим сортираним низом елемената скупа $\{0, \ldots, n-1\}$. Да би низ остао сортиран и након проширивања, низ се проширује редом свим елементима који су строго већи од његовог последњег елемента и мањи од n. Изузетак је празан подскуп (представљен празним низом), који нема последњи елемент и који се проширује редом свим елементима од 0 до $n-1$.

Да бисмо избегли потребу за коришћењем низова који се динамички шире додавањем елемената, унапред ћемо алоцирати низ од n елемената, а подскуп ће бити представљен само елементима у неком његовом префиксу (при том ћемо број елемената подскупа чувати кроз посебну променљиву).

Рекурзивни позиви и подскупови који се њима генеришу су представљени на наредној слици.



Слика 6.3: Рекурзивно генерисање свих подскупова у лексикографсом редоследу

```
// rekurzivna procedura koja ispisuje sva moguca prosirenja datog
// podskupa koji sadrzi k elemenata elementima skupa {0, ..., n-1}
void obradiSvePodskupove(int n, vector<int>& podskup, int k) {
  // obradjujemo (tj. ispisujemo sam podskup)
  obradi(podskup, k);
  // prosirujemo ga svim mogucim elementima skupa {0, ..., n-1}, tako
  // da ostane strogo rastuce sortiran i dobijena prosirenja
  // rekurzivno obradjujemo
  int pocetak = k == 0 ? 0 : podskup[k-1] + 1;
  for (int i = pocetak; i < n; i++) {
    podskup[k] = i;
    obradiSvePodskupove(n, podskup, k+1);
  }
}

// procedura koja obradjuje (ispisuje) sve podskupove skupa {0, ..., n-1}
void obradiSvePodskupove(int n) {
```

```
  vector<int> podskup(n);
  obradiSvePodskupove(n, podskup, 0);
}
```

Задатак: Сви подскупови

Напиши програм који исписује све подскупове датог скупа.

Улаз: Са стандардног улаза се учитава број n (важи $3 \leq n \leq 10$), а затим n природних бројева, растуће сортираних, раздвојених по једним размаком.

Излаз: На стандардни излаз исписати све подскупове учитаног скупа бројева, сваки у посебном реду, са елементима раздвојеним једним размаком. Прво се ређају подскупови у којима први елемент није укључен, а затим они у којима јесте. У свакој од те две групе, прво се исписују подскупови у којима други елемент није укључен, а затим они где јесте и тако даље.

Пример

Улаз	*Излаз*
3	3
1 2 3	2
	2 3
	1
	1 3
	1 2
	1 2 3

Решење

Рекурзивни поступак генерисања свих варијација дужине n скупа $\{0, 1\}$

Генерисање свих подскупова одговара генерисању свих варијација дужине n од нула и јединица (сваки елемент је или укључен или искључен). Поредак описан у поставци задатка указује на то да подскупови треба да буду уређени лексикографски, у односу на њихову репрезентацију у облику варијација. Решење је зато слично решењима задатка Све варијације.

Опишимо индуктивно-рекурзивну конструкцију функције која генерише све подскупове скупа S.

- Ако је скуп S празан, онда је једини његов подскуп празан.
- Ако скуп S није празан, онда се може разложити на неки елемент x и скуп $S' = S \setminus x$ добијен када се тај елемент избаци из полазног скупа. Пошто је скуп S' мањи од скупа S, његови се подскупови могу одредити рекурзивно. Сви подскупови полазног скупа S су онда они који су одређени за мањи скуп S', као и сви они који се од њих добијају додавањем издвојеног елемента x.

Претходну конструкцију није економично програмски реализовати, јер се претпоставља да резултат рада функције представља скуп свих подскупова скупа. Уместо такве функције дефинисаћемо процедуру која неће истовремено чувати и враћати све подскупове већ само један по један набројати и обрадити (у нашем случају само исписати).

До решења се може доћи тако што се у рекурзивној функцији прослеђује неки подсуп P скупа $\{a_0, \ldots, a_{i-1}\}$ и који она на све могуће начине проширује елементима скупа $\{a_i, \ldots, a_{n-1}\}$.

- Ако је скуп $\{a_i, \ldots, a_{n-1}\}$ празан (ако је $i = n$) тада је подскуп P комплетно формиран и обрађује се (тј. исписује).
- У супротном разматрамо елемент a_i и две могућности: да тај елемент буде изостављен из подсупа и могућност да тај елемент буде додат у подскуп P. У оба случаја настављамо рекурзивно проширивање скупа P (прво непроширеног, а затим и проширеног елементом a_i) елементима скупа $\{a_{i+1}, \ldots, a_{n-1}\}$.

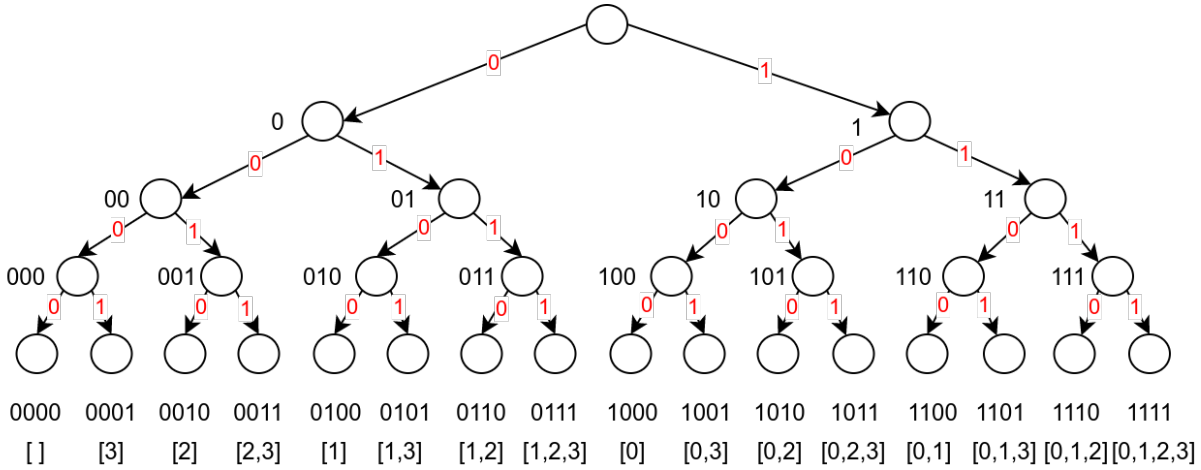
Иницијално је $i = 0$, а подскуп је празан (празан скуп је, заиста, једини подскуп празног супа $\{a_0, \ldots, a_{i-1}\}$ и он се на све могуће начине проширује елементима скупа $\{a_i, \ldots, a_{n-1}\} = \{a_0, \ldots, a_{n-1}\}$).

Иако многи савремени језици пружају тип за репрезентовање скупова, имплементација је једноставнија и ефикаснија ако се елементи скупа чувају у низу. Да бисмо избегли потребу за продужавањем и скраћивањем низа и коришћењем динамичких низова, листа или вектра, низ можемо алоцирати на максималну могућу

дужину (број елемената полазног скупа) и паралелно са низом можемо одржавати број елемената подскупа који је тренутно смештен у низ (он је скоро увек строго мањи од дужине низа).

Дакле, дефинишемо рекурзивну функцију која на сваком наредном нивоу рекурзије обрађује наредни елемент полазног скупа (представљеног низом), све док се не исцрпе сви елементи. У првом случају тај елемент не додаје у резултујући подскуп (такође представљен низом, који прослеђујемо као додатни параметар) и прелази на наредни ниво рекурзије, а у другом га додаје на крај тренутног резултујућег подскупа и прелази на наредни ниво рекурзије. Када се цео полазни низ исцрпи (када је дубина рекурзије једнака дужини полазног низа), тада се тренутно акумулирани подскуп обрађује тј. исписује.

Рад рекурзивне функције приказан је и на слици.



Слика 6.4: Рекурзивно набрајање свих варијација тј. подскупова

```
// procedura određuje i obrađuje sve moguće skupove koji se dobijaju
// tako što se na elemente prosleđenog podskupa p dužine j, dodaju
// podskupovi prosleđenog skupa smeštenog u nizu a, od pozicije i nadalje
void obradi_sve_podskupove(const vector<int>& a, int i, vector<int>& p, int j) {
  // skup preostalih elemenata u nizu a koji se mogu ubaciti u podskup je prazan
  if (i == a.size())
    // ispisujemo formirani podskup
    obradi(p, j);
  else {
    // element na poziciji i ne uključujemo u podskup
    obradi_sve_podskupove(a, i + 1, p, j);
    // element na poziciji i uključujemo u podskup
    p[j] = a[i];
    obradi_sve_podskupove(a, i + 1, p, j + 1);
  }
}

void obradi_sve_podskupove(const vector<int>& a) {
  // podskup je na početku prazan, i u njega potencijalno dodajemo sve
  // elemente skupa a od pozicije 0 nadalje
  vector<int> p(a.size());
  obradi_sve_podskupove(a, 0, p, 0);
}
```

У имплементацији можемо користити и библиотечке колекције за репрезентовање низа елемената. Међутим, треба бити веома обазрив јер је могуће да се током рекурзије граде нови низови и копирају елементи, што знатно утиче на ефикасност. Наредна имплементација је због тога веома лоша.

```
// procedura određuje i obrađuje sve moguće skupove koji se dobijaju
// tako što se na elemente prosleđenog podskupa dodaju podskupovi
// prosleđenog skupa
```

```
void obradiSvePodskupove(const vector<int>& skup, const vector<int>& podskup) {
  // skup je prazan
  if (skup.size() == 0)
    // na prosleđeni podskup možemo dodati samo prazan skup
    obradi(podskup);
  else {
    // izdvajamo i uklanjamo proizvoljan element skupa
    int x = skup.back();
    vector<int> smanjenSkup = skup;
    smanjenSkup.pop_back();
    // u podskup dodajemo sve podskupove skupa bez izdvojenog
    // elementa
    vector<int> podskupBez = podskup;
    obradiSvePodskupove(smanjenSkup, podskupBez);
    // u podskup uključujemo izdvojeni element i zatim sve
    // podskupove skupa bez izdvojenog elementa
    vector<int> podskupSa = podskup;
    podskupSa.push_back(x);
    obradiSvePodskupove(smanjenSkup, podskupSa);
  }
}
```

Могуће је направити решење које користи библиотечке колекција података, а уједно не врши њихово копирање и довољно је ефикасно. У имплементацији се користе два низа (један за скуп, други за подскуп) који се током рада алгоритма мењају. Низови ће се мењати додавањем и укљањањем елемената са краја. Зато је пре почетка функције неохподно обрнути редослед елемената у низу (да би се на крају прво нашли почетни елементи низа).

У имплементацијама које мењају низове често је важно осигурати да се на крају тела рекурзивне функције стање низова врати на исто стање какво је било на уласку у функцију, јер се тиме гарантује да рекурзивни позив неће модификовати низове који су му предати као параметри (што користимо када се ослањамо на то да ће и након првог и након другог рекурзивног позива скуп и подскуп бити исти какви су били и пре тог рекурзивног позива).

```
// procedura određuje i obrađuje sve moguće skupove koji se dobijaju
// tako što se na elemente prosleđenog podskupa dodaju podskupovi
// prosleđenog skupa
void obradiSvePodskupove(vector<int>& skup, vector<int>& podskup) {
  // skup je prazan
  if (skup.size() == 0)
    // na prosleđeni podskup možemo dodati samo prazan skup
    obradi(podskup);
  else {
    // izdvajamo i uklanjamo proizvoljan element skupa
    int x = skup.back();
    skup.pop_back();
    // u podskup dodajemo sve podskupove skupa bez izdvojenog
    // elementa
    obradiSvePodskupove(skup, podskup);
    // u podskup uključujemo izdvojeni element i zatim sve
    // podskupove skupa bez izdvojenog elementa
    podskup.push_back(x);
    obradiSvePodskupove(skup, podskup);
    // vraćamo skup i podskup u početno stanje
    podskup.pop_back();
    skup.push_back(x);
  }
}

void obradiSvePodskupove(vector<int>& skup) {
```

```
  // pošto elementi iz skupa u podskup prebacuju sa desnog kraja, da
  // bismo dobili traženi redosled podskupa, potrebno je da obrnemo
  // redosled elemenata skupa
  vector<int> skupObratno = skup;
  reverse(begin(skupObratno), end(skupObratno));
  // krećemo od praznog podskupa
  vector<int> podskup;
  // efikasnosti radi rezervišemo potrebnu memoriju za najveći podskup
  podskup.reserve(skup.size());
  // prazan skup proširujemo svim podskupovima datog skupa
  obradiSvePodskupove(skupObratno, podskup);
}
```

Функција за одређивање наредне варијације у лексикографском редоследу

Једно решење је да се у посебном низу логичких вредности набрајају све варијације скупа тачно-нетачно. Свака таква варијација одговара једном подскупу, тако што се у подскуп укључују елементи са оних позиција на којима је је вредност тачно. Варијације набрајамо коришћењем функције за одређивање следеће варијације, описане у задатку Следећа варијација.

```
void obradiSvePodskupove(const vector<int>& a) {
  vector<bool> v(a.size(), false);
  do {
    obradi(v, a);
  } while (sledecaVarijacija(v));
}
```

Задатак: Следећа варијација

Напиши програм који одређује наредну варијацију дужине k скупа $\{1, \ldots, n\}$ у лексикографском поретку.

Улаз: Прва линија стандардног улаза садржи број k ($1 \leq k \leq 100$), а друга број n ($1 \leq n \leq 100$). У трећој линији се налази варијација описана бројевима раздвојеним по једним размаком.

Излаз: На стандардни излаз исписати следећу варијацију у лексикографском поретку, ако она постоји, или `-`, ако је учитана варијација лексикографски максимална.

Пример

Улаз	*Излаз*
5 4 1 1 2 3 4	1 1 2 4 1

Решење

Следећа варијација у лексикографском поретку се може генерисати тако што се увећа последњи број у варијацији који се може увећати, и што се након увећавања сви бројеви иза увећаног броја поставе на 1. Позиција на којој се број увећава назива се *преломна тачка* (енгл. turning point). На пример, ако набрајамо варијације скупа $\{1, 2, 3\}$ дужине 5 наредна варијација за варијацију 21332 је 21333 (преломна тачка је позиција 4, која је последња позиција у низу), док је њој наредна варијација 22111 (преломна тачка је позиција 1 на којој се налазио елемент 1). Низ 33333 нема преломну тачку, па самим тим ни лексикографски следећу варијацију.

Један начин имплементације је да преломну тачку нађемо линеарном претрагом од краја низа, ако преломна тачка постоји да увећамо елемент и да од следеће позиције до краја низ попунимо јединицама. Међутим, те две фазе можемо објединити. Варијацију обилазимо од краја постављајући на 1 сваки елемент у варијацији који је једнак броју n. Ако се зауставимо пре него што смо стигли до краја низа, значи да смо пронашли елемент који се може увећати и увећавамо га. У супротном је варијација имала све елементе једнаке n и била је максимална у лексикографском редоследу.

```
bool sledecaVarijacija(int n, vector<int>& varijacija) {
   // od kraja varijacije tražimo prvi element koji se moze povecati
   int i;
```

```
    int k = varijacija.size();
    for (i = k-1; i >= 0 && varijacija[i] == n; i--)
        varijacija[i] = 1;
    // svi elementi su jednaki n - ne postoji naredna varijacija
    if (i < 0) return false;
    // uvecavamo element koji je moguće uvecati
    varijacija[i]++;
    return true;
}
```

Задатак: Све варијације

Напиши програм који одређује све варијације са понављањем дужине k скупа $\{1, \ldots, n\}$.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 5$) и у наредној линији број k ($1 \leq k \leq 5$).

Излаз: На стандардни излаз исписати све варијације, сортиране лексикографски.

Пример

Улаз	*Излаз*
2	1 1 1
3	1 1 2
	1 2 1
	1 2 2
	2 1 1
	2 1 2
	2 2 1
	2 2 2

Решење

Рекурзивно генерисање варијација

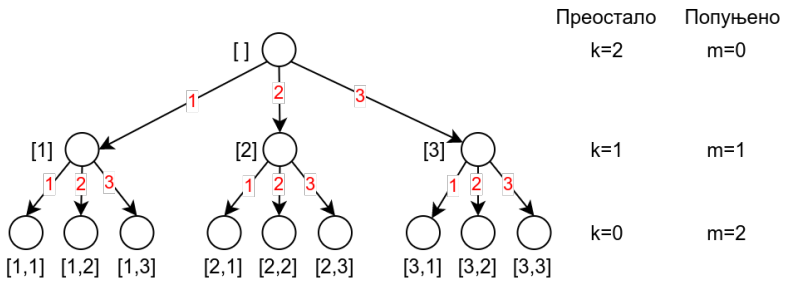
Варијације се могу набројати индуктивно рекурзивном конструкцијом.

- Једина варијација дужине нула је празна.
- Све варијације дужине k се могу добити тако што се на прво место упише било који од бројева од 1 до n, а затим се преостала места допуне свим варијацијама дужине $k - 1$.

Рецимо и да је могуће на последње место постављати један по један број од 1 до n, а затим рекурзивно попуњавати префикс, но тиме би редослед варијација био другачији од траженог.

Уместо да дефинишемо функцију која враћа колекцију варијација, дефинисаћемо рекурзивну процедуру која прима низ коме су попуњени сви елементи осим последњих k, и који ће на све могуће начине допуњавати варијацијама дужине k (која ће се смањивати кроз рекурзивне позиве). Дакле, након попуњеног дела низа постављамо једну по једну вредност од 1 до n и затим рекурзивно позивамо процедуру да попуни остатак низа (тиме што смањујемо дужину k и тиме прелазимо на наредну позицију).

Да бисмо избегли потребу за динамичким проширивањем низова унапред ћемо алоцирати низ дужине k, а онда ћемо му у рекурзији попуњавати једну по једну позицију (текућа позиција се може одредити као разлика између дужине низа и текуће вредности броја k).



Слика 6.5: Рекурзивни поступак генерисања варијација дужине 2 од елемената скупа $\{1, 2, 3\}$

```
// Sve varijacije duzine k elemenata skupa {1, ..., n}
// Data varijacija sa ponavljanjem duzine varijacije.size() - k se
// dopunjuje svim mogucim varijacijama sa ponavljanjem duzine k skupa
// {1, ..., n} i sve tako dobijene varijacije se obrađuju funkcijom
// obradi
void obradiSveVarijacije(int k, int n,
                         vector<int>& varijacija) {
  // k je 0, pa je jedina varijacija duzine nula prazna i njenim
  // dodavanjem na polazni niz on se ne menja
  if (k == 0)
    obradi(varijacija);
  else
    // na tekucu poziciju postavljamo sve moguce vrednosti od 1 do n i
    // dobijeni niz onda rekurzivno proširujemo
    for (int nn = 1; nn <= n; nn++) {
       varijacija[varijacija.size() - k] = nn;
       obradiSveVarijacije(k-1, n, varijacija);
    }
}


// sve varijacije duzine k skupa {1, ..., n}
void obradiSveVarijacije(int k, int n) {
  vector<int> varijacija(k);
  obradiSveVarijacije(k, n, varijacija);
}
```

Наравно, уместо дужине k преосталог дела низа који тек треба да се попуни, могуће је прослеђивати дужину m већ попуњеног дела низа (то је уједно позиција на коју се уписује наредни елемент). Излаз из рекурзије је када број попуњених елемената m достигне дужину низа (тада је цео низ којим је варијација попуњен).

```
// Sve varijacije duzine k elemenata skupa {1, ..., n}
// Data varijacija sa ponavljanjem duzine m se dopunjuje svim mogucim
// varijacijama sa ponavljanjem duzine n-m skupa {1, ..., n} i sve
// tako dobijene varijacije se obrađuju funkcijom obradi
void obradiSveVarijacije(int n,
                         vector<int>& varijacija, int m) {
  // m je n, pa se trenutna varijacija ne moze vise dopuniti
  if (m == varijacija.size())
    obradi(varijacija);
  else
    // na tekucu poziciju postavljamo sve moguce vrednosti od 1 do n i
    // dobijeni niz onda rekurzivno proširujemo
    for (int nn = 1; nn <= n; nn++) {
       varijacija[m] = nn;
       obradiSveVarijacije(n, varijacija, m+1);
    }
```

```
}


// sve varijacije duzine k skupa {1, ..., n}
void obradiSveVarijacije(int k, int n) {
  vector<int> varijacija(k);
  obradiSveVarijacije(n, varijacija, 0);
}
```

Проналажење лексикографски следеће варијације

Друга могућност је да се крене од лексикографски најмање варијације (то је варијација $\underbrace{11\ldots11}_{k}$) и да се коришћењем функције описане у задатку Следећа варијација одређује наредна варијација дате варијације у односу на лексикографски редослед, све док таква постоји.

```
void obradiSveVarijacije(int k, int n) {
   // krećemo od varijacije 11...11 - ona je leksikografski najmanja
   vector<int> varijacija(k, 1);
   // obradjujemo redom varijacije dok god postoji leksikografski
   // sledeca
   do {
     obradi(varijacija);
   } while(sledecaVarijacija(n, varijacija));
}
```

Задатак: Следећи бинарни низ без суседних јединица

Посматрајмо лексикографски поређане бинарне низове дужине n који садрже нуле и јединице, али не садрже две суседне јединице. На пример, такви низови дужине 3 су 000, 001, 010, 100 и 101. Напиши програм који за дати низ одређује наредни низ у лексикографском поретку.

Улаз: Са стандардног улаза се учитава бинарни низ без узастопних јединица дужине n ($1 \leq n \leq 50$). Сви елементи су записани један иза другог, без размака.

Излаз: Једина линија стандардног излаза треба да садржи елементе наредног низа у лексикографском поретку (исписане један иза другог, без размака) или текст `ne postoji` ако је учитани низ лексикографски највећи.

Пример 1

Улаз	*Излаз*
`10101000100001010`	`10101000100010000`

Пример 2

Улаз	*Излаз*
`10101010`	`ne postoji`

Решење

Алгоритам којим се одређује следећи низ у лексикографском редоследу представљаће модификацију алгоритма којим се одређује следећа варијација у лексикографском редоследу. Тај алгоритам је описан у задатку Следећа варијација.

Подсетимо се, крећемо од краја низа, проналазимо прву позицију на којој се налази елемент чија вредност није тренутно максимална (ако она постоји), увећавамо је, и након тога све елементе иза ње постављамо на минималну вредност (у нашем контексту максимална вредност је 1, а минимална 0). Ово можемо остварити и у једном проласку тако што крећући се уназад све максималне вредности одмах постављамо на нулу.

Модификујмо сада овај алгоритам тако да ради за низове који немају две узастопне јединице. Ако се након примене претходног алгоритма догодило да је увећана (постављена на јединицу) цифра испред које не стоји јединица, то је решење које смо тражили. Међутим, ако је увећана цифре испред које је јединица, то није решење које смо тражи, и морамо да наставимо да једнице претварамо у нуле.

Пример. На пример, следећи низ у односу на низ 01001 је 01010. Међутим, ако затражимо наредни елемент, добићемо 01011, а то је елемент који није допуштен. Настављањем даље добијамо 01100, што је такође елемент који није допуштен, након тога ређају се елементи од 01101, до 01111, који су сви недопуштени да бисмо на крају добили 10000, што је заправо наредни елемент у односу на 01010.

Дакле, опет се крећемо са краја низа и уписујемо нуле све док се на тренутној или на претходној позицији у низу налази јединица. На крају, на позицији на којој смо се зауставили и нисмо уписали нулу (ако таква постоји) уписујемо јединицу (то је позиција на којој пише нула и испред ње или нема ништа или пише нула). Ако таква позиција не постоји, онда је тренутни низ лексикографски највећи.

Једноставности ради, низ представљамо у облику ниске карактера. Наравно, могућа је и репрезентација у неком облику низа целих бројева.

```
bool sledeciNiz(string& s){
  int n = s.length();
  int i = n - 1;
  while ((i >= 0 && s[i] == '1') ||
         (i > 0 && s[i - 1] == '1'))
    s[i--] = '0';
  if (i < 0)
    return false;
  s[i] = '1';
  return true;
}
```

Задатак: Сви бинарни низови без суседних јединица

Напиши програм који исписује све низове бинарних бројева дате дужине у којима се не јављају две узастопне јединице. Бројеве исписати у лексикографском редоследу.

Улаз: Са стандардног улаза се уноси број n.

Излаз: На стандардни излаз исписати тражене бројеве, сваки у посебном реду.

Пример

Улаз	*Излаз*
3	000
	001
	010
	100
	101

Решење

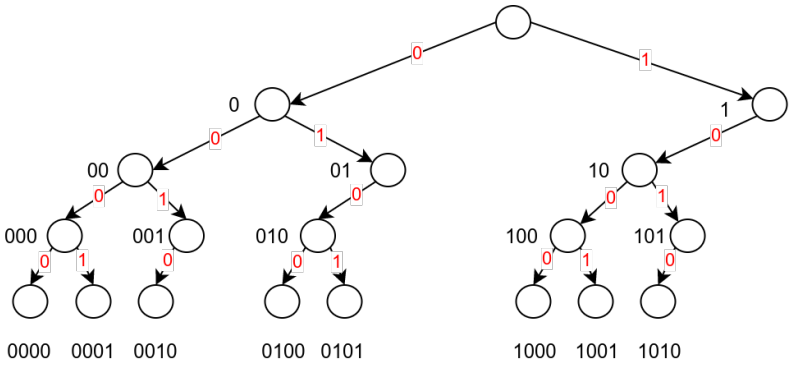
Задатак представља модификацију задатка Све варијације.

Следећа варијација у лексикографском поретку

Једна могућност је да се употреби функција која генерише наредну у лексикографском поретку бинарну ниску без суседних једница. Та функција је описана у задатку Следећи бинарни низ без суседних јединица). Креће се од ниске која садржи n нула и исписује се и рачуна наредна варијација све док таква постоји.

Рекурзивно генерисање варијација

Један начин је да дефинишемо рекурзивну функцију која генерише тражене бројеве. Она добија попуњен префикс низа дужине i и покушава на све начине да га прошири (низ карактера је у старту алоциран на дужину n и кроз рекурзију се мало по мало попуњава). Ако је $i = n$, тада је цео низ попуњен и исписује се. У супротном, на позицију i увек можемо дописати нулу и рекурзивно наставити са продужавањем тако добијене ниске. Са друге стране, јединицу можемо уписати само ако претходни карактер није јединица (у супротном бисмо добили две узастопне јединице). То се дешава или када нема претходне цифре (када је $i = 0$) или када је претходна цифра (на позицији $i - 1$) различита од јединице.



Слика 6.6: Рекурзивно генерисање свих бинарних низова без суседних јединица - када је последња цифра у текућем низу јединица, низ се не може проширити јединицом

```
// prefiks duzine i se prosiruje na sve moguce nacine
void obradiSveBinarneBez11(string& binarni, int i) {
  // ceo niz je popunjen
  if (i == binarni.size())
    obradi(binarni);
  else {
    // prefiks uvek mozemo prosiriti nulom
    binarni[i] = '0';
    obradiSveBinarneBez11(binarni, i+1);
    // prefiks mozemo prosiriti jedinicom, samo ako se ne zavrsava jedinicom
    if (i == 0 || binarni[i-1] != '1') {
      binarni[i] = '1';
      obradiSveBinarneBez11(binarni, i+1);
    }
  }
}

// generise i ispisuje sve binarne nizove koji ne sadrze dve susedne jedinice
void obradiSveBinarneBez11(int n) {
  string binarni(n, '0');
  obradiSveBinarneBez11(binarni, 0);
}
```

Задатак: Варијације без понављања

Варијација класе k без понављања елемената скупа S је сваки уређена k - торка од k различитих елемената скупа S. Написати програм који за дато n и k приказује све варијације без понављања класе k скупа бројева од 1 до n, у лексикографском поретку.

Улаз: Прва линија стандардног улаза садржи природан број n ($n \leq 8$), у другој линији налази се природан број k ($0 < k \leq n$).

Излаз: На стандрадном излазу приказати у лексикографском поретку све варијације класе k бројева од 1 до n (сваку у посебном реду).

Пример

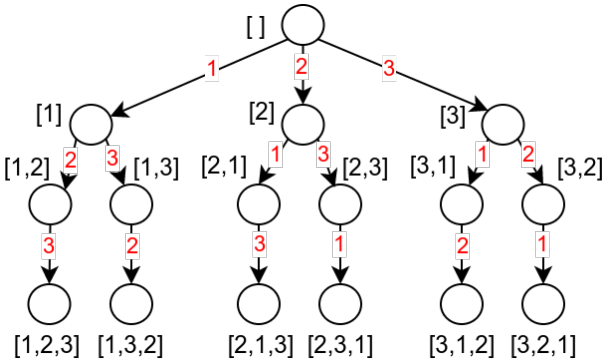
Улаз	*Излаз*
3	1 2
2	1 3
	2 1
	2 3
	3 1
	3 2

Решење

Рекурзивно генерисање варијација

Можемо дефинисати и рекурзивну функцију која набраја све варијације без понављања, која је слична функцији која набраја варијације са понављањем. Та функција је приказана у задатку Све варијације.

Функција прима до сада попуњен префикс варијације и покушава да постави елемент на позицију i. Претпоставићемо да је у почетку низ алоциран на дужину k, и да попуњавање креће од позиције $i = 0$. Ако је $i = k$, тада је варијација цела попуњена и обрађујемо је (у нашем случају је само исписујемо). У супротном на место i редом постављамо један по један елемент скупа $\{1, \ldots, n\}$ који није раније употребљен у варијацији и рекурзивно прелазимо на попуњавање варијације од позиције $i + 1$.



Слика 6.7: Рекурзивно генерисање варијација без понављања за $k =$ и $n = 3$ — приметимо да су резултујуће варијације заправо пермутације низа $1, 2, 3$

Да бисмо ефикасно могли да одредимо да ли је тренутни кандидат већ употребљен, користимо помоћни низ логичких вредности (на место i уписујемо вредност тачно ако и само ако се елемент i јавља у текућој варијацији тј. у њеном до тада попуњеном делу).

```
// popunjava se varijacija a od pozicije i nadalje elementima skupa
// {1, ..., n} pri cemu se u nizu upotrebljen beleze upotrebljeni
// elementi u delu varijacije pre pozicije i
void varijacijeBezPonavljanja(vector<int>& a, int n, vector<bool>& upotrebljen, int i) {
  // varijacija je cela popunjena, pa je ispisujemo
  if (i == a.size())
    obradi(a);
  else {
    // na poziciju i stavljamo redom svaki neupotrebljen element
    for (int x = 1; x <= n; x++)
      if(!upotrebljen[x]) {
        a[i] = x;
        upotrebljen[x] = true;
        varijacijeBezPonavljanja(a, n, upotrebljen, i+1);
        upotrebljen[x] = false;
      }
  }
}

// ispisuje sve varijacije bez ponavljanja k elemenata skupa {1, ..., n}
void varijacijeBezPonavljanja(int n, int k) {
  vector<int> a(k);
  vector<bool> upotrebljen(n+1, false);
  varijacijeBezPonavljanja(a, n, upotrebljen, 0);
}
```

Следећа варијација у лексикографском поретку

Један начин да се задатак реши је да се дефинише функција која за дату варијацију проналази следећу варијацију у лексикографском поретку.

Алгоритам представља модификацију алгоритма описаном у задатку Следећа варијација.

Полазимо од краја варијације тражећи позицију на којој се налази неки елемент који се може увећати. Да би увећање елемента a_i било могуће, мора постојати неки елемент који је строго већи од a_i (а мањи или једнак n), који се не јавља пре позиције i (јер дупликати нису дозвољени). Ако таква позиција не постоји, тада је варијација лексикографски највећа. У супротном увећавамо елемент a_i на позицији i (на најмању могућу вредност из скупа $\{1, \ldots, n\}$ која се не јавља пре позиције i), а елементе иза позиције i попуњавамо редом што мањим елементима скупа $\{1, \ldots, n\}$, који се нису појављивали у дотадашњем делу низа.

Да бисмо ефикасније одређивали елементе који су већ употребљени у првом делу варијације, посебно ћемо одржавати скуп тих елемената (најбоље у облику низа логичких вредности тако да вредност на месту i говори да ли је елемент i већ употребљен). Низ ћемо иницијализовати коришћењем целе полазне варијације, а онда ћемо приликом њеног обиласка здесна налево искључивати једну по једну вредност (јер нас занимају само појављивања вредности пре позиције i, а не на њој и после ње).

Пример. На пример, за $n = 5$ и $k = 5$, следећа варијација у односу на $1, 4, 3, 5, 2$ је $1, 4, 5, 2, 3$. Наиме, елемент 2 не може да се увећа (јер су већи елементи 3, 4 и 5 сви већ употребљени испред њега), елемент 5 не може да се увећа (јер од њега нема већих елемената), док елемент 3 може да се увећа на 5 (не може на 4, јер је елемент 4 већ употребљен испред њега). Након увећања иза се слажу редом елементи 2 и 3 (јер је 1 већ употребљен).

У главном програму се креће од лексикографски најмање варијације $1, \ldots, k$, свака варијација се обрађује (исписује) и тражи се наредна.

Анализа сложености. И уз коришћења помоћног низа логичких вредности, сложеност тражења наредне варијације у лексикографском редоследу је квадратна $O(kn)$, јер се за сваку од k позиција обилази $O(n)$ вредности (анализирају се све вредности од $a_i + 1$ до n). Сложеност се може поправити коришћењем ефикаснијих структура података.

```
// pronalazi i vraca prvi element u intervalu (ai, n] koji nije upotrebljen
// tj. -1 ako takav element ne postoji
int veciNeupotrebljen(int ai, int n, const vector<bool>& upotrebljen) {
  for (int x = ai+1; x <= n; x++)
    if (!upotrebljen[x])
      return x;
  return -1;
}

bool sledecaVarijacija(vector<int>& a, vector<bool>& upotrebljen, int n) {
  // broj elemenata varijacije
  int k = a.size();

  // analiziramo jednu po jednu poziciju od kraja i pokusavamo da
  // pronadjemo poziciju i takvu da joj se vrednost moze uvecati
  // tj. takvu da postoji element ai_novo > a[i] koji se ne javlja
  // nigde ispred pozicije i
  int i, ai_novo;
  for (i = k-1; i >= 0; i--) {
    upotrebljen[a[i]] = false;
    ai_novo = veciNeupotrebljen(a[i], n, upotrebljen);
    if (ai_novo != -1)
      break;
  }
  // ako se ni jedan element ne moze uvecati, trenutna varijacija je
  // leksikografski najvecu varijaciju
  if (i < 0) return false;
```

```
  // menjamo element a[i] vrednoscu ai_novo
  upotrebljen[ai_novo] = true;
  a[i++] = ai_novo;

  // elemente do kraja varijacije popunjavamo sto manjim,
  // neupotrebljenim elementima
  for (int x = 1; i < k; x++)
    if (!upotrebljen[x]) {
      a[i++] = x;
      upotrebljen[x] = true;
    }

  // uspeli smo da konstruisemo leksikografski narednu varijaciju
  return true;
}

// ispisuje sve varijacije bez ponavljanja k elemenata skupa {1, ..., n}
void varijacijeBezPonavljanja(int n, int k) {
  // elementi varijacije
  vector<int> a(k);
  // za svaki element od 1 do n belezimo da li je upotrebljen u
  // tekucoj varijaciji
  vector<bool> upotrebljen(n+1, false);

  // krecemo od leksikografski najmanje varijacije 1, 2, ..., k
  for (int i = 0; i < k; i++) {
    a[i] = i+1;
    upotrebljen[i+1] = true;
  }

  // ispisujemo tekucu varijaciju i prelazimo na sledecu, sve dok ne
  // dodjemo do leksikografski najvece (koja nema sledecu)
  do {
    obradi(a);
  } while (sledecaVarijacija(a, upotrebljen, n));
}
```

Задатак: Следећа комбинација

Комбинације дужине k од n елемената подразумевају да се врши одабир k елемената скупа $\{1, \ldots, n\}$, слично као што се, на пример, у игри лото бира 7 од 39 куглица. Напиши програм који за дату комбинацију одређује наредну у лексикографском поретку.

Улаз: Са стандардног улаза се уноси број n ($2 \leq n \leq 100$) а затим у наредном реду једна комбинација дужине $1 \leq k \leq n$. Елементи су задати сортирани растуће, одвојени по једним размаком.

Излаз: На стандардни излаз исписати комбинацију која је наредна у лексикографском редоследу у односу на дату, тј. - ако таква комбинација не постоји.

Пример 1		**Пример 2**		**Пример 3**	
Улаз	*Излаз*	*Улаз*	*Излаз*	*Улаз*	*Излаз*
5 1 3 4	1 3 5	5 1 3 5	1 4 5	5 3 4 5	-

Решење

Опишимо поступак којим од дате комбинације можемо добити следећу комбинацију у лексикографском редоследу.

Напишимо, илустрације ради, све комбинације дужине 3 из скупа од 5 елемената.

Поново тражимо *преломну тачку* тј. елемент који се може увећати. Пошто су комбинације дужине k и организоване су строго растуће, максимална вредност на последњој позицији је n, на претпоследњој $n-1$ итд. Дакле, последњи елемент се може увећати ако није једнак n, претпоследњи ако није једнак $n-1$ итд. Преломна тачка је позиција првог елемента који је мањи од свог максимума. Ако позиције бројимо од 0, максимум на позицији $k-1$ је n, на позицији $k-2$ је $n-1$ итд. тако да је максимум на позицији i једнак $n-k+1+i$. Ако преломна тачка не постоји (ако су све вредности на својим максимумима), наредна комбинација у лексикографском редоследу не постоји. У супротном увећавамо елемент на преломној позицији и да бисмо након тога добили лексикографски што мању комбинацију, све елементе иза њега постављамо на најмање могуће вредности. Пошто комбинација мора бити сортирана строго растуће, након увећања преломне вредности све елементе иза ње постављамо на вредност која је за један већа од вредности њој претходне вредности у низу.

Пример. На пример, ако је $n = 6$ и $k = 4$, тада је наредна комбинација комбинацији 1256, комбинација 1345 - преломна вредност је 2 и она се може увећати на 3, након чега слажемо редом елементе за по један веће.

Прикажимо све преломне тачке приликом генерисања комбинација дужине $n = 3$ из скупа од $k = 5$ елемената.

$12\mathbf{3} \to 12\mathbf{4} \to 1\mathbf{2}5 \to 13\mathbf{4} \to 1\mathbf{3}5 \to \mathbf{1}45 \to 23\mathbf{4} \to 2\mathbf{3}5 \to \mathbf{2}45 \to 345$

Задатак: Све комбинације

Комбинације дужине k од n елемената подразумевају да се врши одабир k елемената скупа $\{1, \ldots, n\}$, слично као што се, на пример, у игри лото бира 7 од 39 куглица. Напиши програм који за дате вредности k и n набраја и исписује све комбинације, поређане по лексикографском редоследу.

Улаз: Прва линија стандардног улаза садржи број k ($1 \leq k \leq n$), а наредна број n ($2 \leq n \leq 20$).

Излаз: На стандардни излаз исписати све комбинације. Свака комбинација треба да буде представљена низом бројева сортираним строго растуће, а све комбинације треба да буду поређане у лексикографском редоследу.

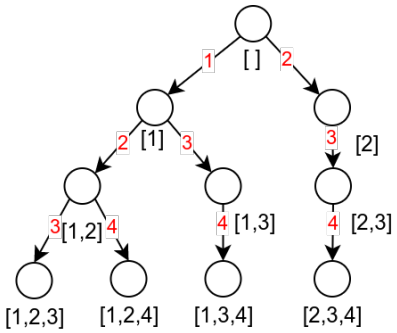
Пример

Улаз	*Излаз*
3	1 2 3
5	1 2 4
	1 2 5
	1 3 4
	1 3 5
	1 4 5
	2 3 4
	2 3 5
	2 4 5
	3 4 5

Решење

Рекурзивни позиви по позицијама

Задатак рекурзивне функције биће да допуни низ дужине k од позиције i па до краја. Када је $i = k$, низ је попуњен и потребно је обрадити (у нашем случају исписати) добијену комбинацију. У супротном бирамо елемент који ћемо поставити на позицију i. Пошто су комбинације уређене строго растуће, он мора бити већи од претходног (ако претходни не постоји, онда може бити 1) и мањи или једнак n. Заправо, ово горње ограничење мора да се смањи. Пошто су елементи строго растући, а од позиције i па до краја низа треба поставити $k-i$ елемената, на позицији i може бити $n+i-k+1$ и тада ће на позицији $k-1$ бити вредност n. У петљи стављамо један по један од тих елемената на позицију i и рекурзивно настављамо генерисање од наредне позиције. Дакле, у општем случају, дрво рекурзивних позива неће бити бинарно (функција може да направи много више од два рекурзивна позива).



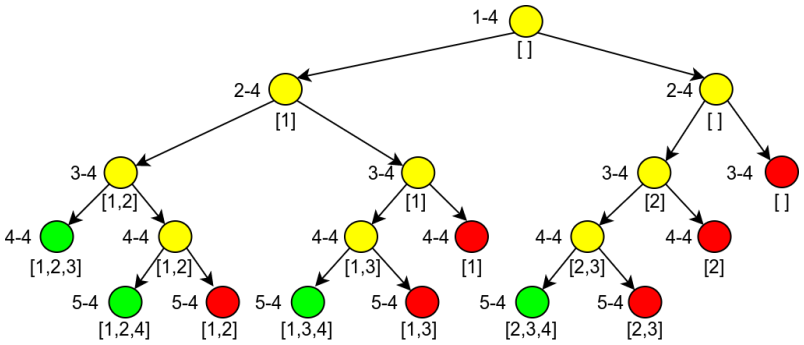
Слика 6.8: Рекурзивно генерисање комбинација дужине 3 из скупа $\{1, 2, 3, 4\}$

```
// niz kombinacije dužine k na pozicijama [0, i) sadrži uređen
// niz elemenata iz skupa [1, n-i+1). Procedura na sve moguće
// načine dopunjava elementima iz skupa [1, n) tako da niz bude
// uređen rastući
void obradiSveKombinacije(vector<int>& kombinacija, int i, int n) {
  // tražena dužina kombinacije
  int k = kombinacija.size();

  // ako je popunjen ceo niz samo ispisujemo kombinaciju
  if (i == k) {
    obradi(kombinacija);
    return;
  }
  // određujemo raspon elemenata na poziciji i
  int pocetak = i == 0 ? 1 : kombinacija[i-1]+1;
  int kraj = n + i - k + 1;
  // jedan po jedan element upisujemo na poziciju i, pa
  // nastavljamo generisanje rekurzivno
  for (int x = pocetak; x <= kraj; x++) {
    kombinacija[i] = x;
    obradiSveKombinacije(kombinacija, i+1, n);
  }
}

// nabraja i obradjuje sve kombinacije dužine k skupa
// {1, 2, ..., n}
void obradiSveKombinacije(int k, int n) {
  vector<int> kombinacija(k);
  obradiSveKombinacije(kombinacija, 0, n);
}
```

Рекурзивни позиви по вредностима

Постоји начин да избегнемо рекурзивне позиве у петљи. Током рекурзије можемо да чувамо информацију о томе који је распон елемената којим се проширује низ. Знамо да су то елементи скупа $\{1, \ldots, n\}$, међутим, пошто су комбинације сортиране растуће скуп кандидата је ужи. У претходном програму смо најмању вредност за позицију i одређивали на основу вредности са позиције $i - 1$, међутим, алтернативно можемо и експлицитно да одржавамо промењиве n_{min} и n_{max} које одређују скуп $\{n_{min}, \ldots, n_{max}\}$ чији се елементи распоређују у комбинацији на позицијама из интервала $[i, k)$. Ако је тај интервал празан, комбинација је попуњена и може се обрадити. У супротном, ако је $n_{min} > n_{max}$, тада не постоји вредност коју је могуће ставити на позицију i, па можемо изаћи из рекурзије, јер се тренутна комбинација не може попунити до краја. У супротном можемо размотрити две могућности. Прво на позицију i можемо поставити елемент n_{min} и рекурзивно извршити попуњавање низа од позиције $i + 1$, а друго можемо тај елемент прескочити и у рекурзивном позиву поново захтевати да се попуни позиција i. У оба случаја се скуп елемената сужава на $\{n_{min} + 1, \ldots, n_{max}\}$.

Претрагу можемо сасећи и мало раније. Наиме, пошто су понављања забрањена када је број елемената тог скупа (а то је $n - n_{min} + 1$) мањи од броја преосталих позиција које треба попунити (а то је $k - i$), већ тада можемо сасећи претрагу, јер не постоји могућност да се комбинација успешно допуни до краја.



Слика 6.9: Рекурзивно генерисање комбинација - лево од сваког чвора је приказан распон преосталих вредности, испод чвора текућа комбинација. У зеленим чворовима су успешно генерисане комбинације, а у црвеним наступа одсецање, јер распон не садржи довољно вредности да би се комбинација генерисала до краја

```
void obradiSveKombinacije(vector<int>& kombinacija, int i,
                          int n_min, int n_max) {
  // tražena dužina kombinacije
  int k = kombinacija.size();

  // ako je popunjen ceo niz samo ispisujemo kombinaciju
  if (i == k) {
    obradi(kombinacija);
    return;
  }

  // ako tekuću kombinaciju nije moguće popuniti do kraja
  // prekidamo ovaj pokušaj
  if (n_max - n_min + 1 < k - i)
    return;

  // vrednost n_min uključujemo na poziciju i, pa rekurzivno
  // proširujemo tako dobijenu kombinaciju
  kombinacija[i] = n_min;
  obradiSveKombinacije(kombinacija, i+1, n_min+1, n_max);
  // vrednost n_min preskačemo i isključujemo iz kombinacije
  obradiSveKombinacije(kombinacija, i, n_min+1, n_max);
}

// nabraja i obradjuje sve kombinacije dužine k skupa
// {1, 2, ..., n}
void obradiSveKombinacije(int k, int n) {
  vector<int> kombinacija(k);
  obradiSveKombinacije(kombinacija, 0, 1, n);
}
```

Лексикографски следећа комбинација

Један начин да се задатак реши без рекурзије је да се употреби функција за одређивање наредне комбинације у лексикографском поретку која је описана у задатку Следећа комбинација.

```
// nabraja i obradjuje sve kombinacije dužine k skupa
// {1, 2, ..., n}
void obradiSveKombinacije(int k, int n) {
```

```
  // krecemo od kombinacije 1, 2, ..., k
  vector<int> kombinacija(k);
  for (int i = 0; i < k; i++)
    kombinacija[i] = i + 1;

  // obradjujemo kombinacije dokle god postoji sledeca
  do {
    obradi(kombinacija);
  } while (sledecaKombinacija(n, kombinacija));
}
```

Задатак: Следећа пермутација

Све пермутације бројева од 1 до n се могу поређати лексикографски. На пример за $n = 3$ пермутације у лексикографском поретку су

```
123
132
213
231
312
321
```

Написати програм којим се за дати природан број n и дату пермутацију бројева од 1 до n приказује следећа пермутација у лексикографском поретку (прва пермутација која се налази после дате пермутације).

Улаз: Прва линија стандардног улаза садржи природан број n ($n < 1000$). У свакој од n наредних линија стандардног улаза, налазе се редом елементи пермутације, сваки у посебној линији.

Излаз: На стандардном излазу приказати редом елементе следеће пермутације у лексикографском поретку, сваки елемент у посебној линији. Ако не постоји следећа пермутација (дата пермутација је последња) приказати у једној линији поруку `ne postoji`.

Пример 1

Улаз	*Излаз*
5	3
3	1
1	5
4	2
5	4
2	

Пример 2

Улаз	*Излаз*
3	ne postoji
3	
2	
1	

Решење

Алгортитам за одређивање наредне пермутације у лексикографском редоследу

Пример. Размотримо пермутацију 13542. Заменом елемента 2 и 4 би се добила пермутација 13524 која је лексикографски мања од полазне и то нам не одговара. Слично би се десило и да се замене елементи 5 и 4. Чињеница да је низ 542 строго опадајући нам говори да није могући ни на који начин разменити та три елемента да се добије лексикографски већа пермутација, тј. да је ово највећа пермутација која почиње са 13. Дакле, наредна пермутација ће бити лексикографски најмања пермутација која почиње са 14, а то је 14235.

Дакле, у првом кораку алгоритма проналазимо преломну тачку, тј. прву позицију i здесна, такву да је $a_i < a_{i+1}$ (за све $i+1 \leq k < n-1$ важи да је $a_k > a_{k+1}$). Ово радимо најјобичнијом линеарном претрагом. Ако таква позиција не постоји, наша пермутација је скроз опадајућа и самим тим лексикографски највећа. Након тога, проналазимо прву позицију j здесна такву да је $a_i < a_j$ (опет линеарном претрагом) и размењујемо елементе на позицијама i и j. Пошто је овом разменом реп иза позиције i и даље стриктно опадајући, да бисмо добили жељену пермутацију (лексикографски најмању пермутацију која почиње са $a_0 \dots a_{i-1} a_j$), потребно је обрнути редослед елемената репа тј. део низа од позиције $i+1$ до краја низа.

Пример. Ако означимо позиције елемената добијамо $1^0 3^1 5^2 4^3 2^4$. Зато је $i = 1$ и $a_i = 3$, док је $j = 3$ и $a_j = 4$. Након размене добијамо $1^0 4^1 5^2 3^3 2^4$. Да бисмо добили тражену пермутацију $1^0 4^1 2^2 3^3 5^4$ обрћемо део

низа од позиције $i + 1 = 2$ до краја низа.

```
bool sledecaPermutacija(vector<int>& a){
  int n = a.size();

  // linearnom pretragom pronalazimo prvu poziciju i takvu da
  // je a[i] > a[i+1]
  int i = n - 2;
  while (i >= 0 && a[i] > a[i+1])
    i--;
  // ako takve pozicije nema, permutacija a je leksikografski maksimalna
  if (i < 0) return false;
  // linearnom pretragom pronalazimo prvu poziciju j takvu da
  // je a[j] > a[i]
  int j = n - 1;
  while (a[j] < a[i])
    j--;
  // razmenjujemo elemente na pozicijama i i j
  swap(a[i], a[j]);
  // obrcemo deo niza od pozicije i+1 do kraja
  for (j = n - 1, i++; i < j; i++, j--)
    swap(a[i], a[j]);
  return true;
}
```

Библиотечка функција

У језику C++ постоји библиотечка функција `next_permutation` која одређује следећу пермутацију у лексикографском редоследу (и враћа информацију о томе да ли она постоји).

```
// ucitavamo polaznu permutaciju
int n;
cin >> n;
vector<int> a(n);
for(int i = 0; i < n; i++)
  cin >> a[i];

// odredjujemo sledecu i ispisujemo rezultat
if (next_permutation(begin(a), end(a)))
  obradi(a);
else
  cout << "ne postoji" << endl;
```

Задатак: Све пермутације

Напиши програм који генерише и исписује све пермутације скупа $\{1, 2, \ldots, n\}$.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 8$).

Излаз: На стандардни излаз исписати тражене пермутације. Сваку пермутацију исписати у посебном реду, а елементе раздвојити по једним размаком. Редослед пермутација може бити произвољан.

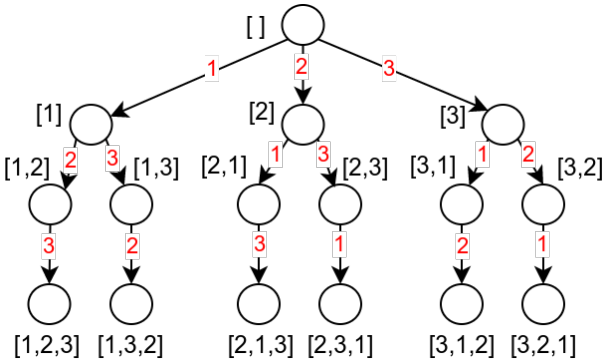
Пример

Улаз	*Излаз*
3	1 2 3
	1 3 2
	2 1 3
	2 3 1
	3 1 2
	3 2 1

Решење

Рекурзивно генерисање permутација са експлицитном провером да ли је елемент већ употребљен

Пермутације се могу рекурзивно генерисати веома слично поступку генерисања варијација без понављања. У рекурзивној функцији обрађујемо једну по једну позицију и на њу стављамо оне елементе скупа $\{1, \ldots, n\}$ који се не налазе на претходним позицијама. Да би се избегла линеарна претрага претходних позиција, могуће је користити помоћни низ логичких вредности у коме се за сваки елемент означава да ли је већ искоришћен или није. Овакав приступ је објашњен у задатку Варијације без понављања.



Слика 6.10: Рекурзивно генерисање пермутација низа 123 - на текућу позицију се поставља један по један елемент низа 123, који није већ постављен на претходне позиције

```
// popunjava se permutacija a od pozicije i nadalje elementima skupa
// {1, ..., n} pri cemu se u nizu upotrebljen beleze upotrebljeni
// elementi u delu permutacije pre pozicije i
void permutacije(vector<int>& a, int n, vector<bool>& upotrebljen, int i) {
  // permutacija je cela popunjena, pa je ispisujemo
  if (i == a.size())
    obradi(a);
  else {
    // na poziciju i stavljamo redom svaki neupotrebljen element
    for (int x = 1; x <= n; x++)
      if(!upotrebljen[x]) {
        a[i] = x;
        upotrebljen[x] = true;
        permutacije(a, n, upotrebljen, i+1);
        upotrebljen[x] = false;
      }
  }
}

// ispisuje sve permutacije skupa {1, ..., n}
void permutacije(int n) {
  vector<int> a(n);
  vector<bool> upotrebljen(n+1, false);
  permutacije(a, n, upotrebljen, 0);
}
```

Рекурзивно генерисање permутација без експлицитне провере да ли је елемент већ употребљен

Ако се одрекнемо услова да пермутације буду уређене лексикографски, није неопходно вршити проверу да ли је текући елемент већ употребљен тј. можемо поступити на следећи начин.

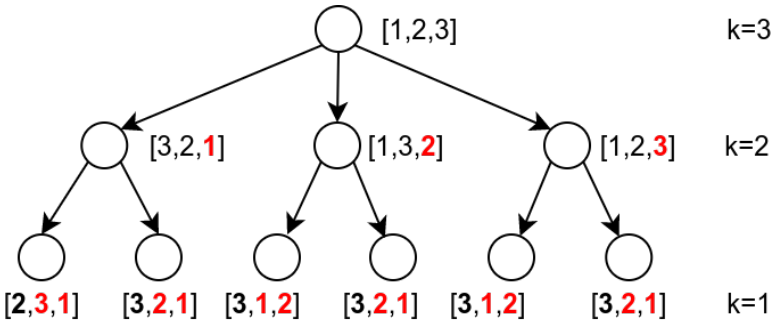
На последњу позицију у низу у којем чувамо текућу пермутацију треба да постављамо један по један елемент скупа, а затим да рекурзивно одређујемо све пермутације преосталих елемената (алтернативно бисмо могли да кренемо и од прве позиције). Тиме се на сваком нивоу рекурзије разликују префикс пермутације који садржи

елементе које тек треба пермутовати и суфикс пермутације који садржи елементе који су фиксирани и већ се налазе на својим позицијама. Фиксиране елементе и елементе које треба пермутовати можемо чувати у истом низу. Нека на позицијама $[0, k)$ чувамо елементе које треба пермутовати, а на позицијама $[k, n)$ чувамо фиксиране елементе. Рекурзивна функција, дакле, прима низ, и број k и покушава да фиксиран суфикс елемената на позицијама $[k, n)$ на све могуће начине прошири пермутацијама елемената на позицијама $[0, k)$.

- Ако је $k = 1$, тада постоји само једна пермутација једночланог низа на позицији 0, њу придружујемо фиксираним елементима (пошто је она већ на месту 0 нема потребе ништа додатно радити) и обрађујемо је (тј. исписујемо).
- Ако је $k > 1$, тада је ситуација компликованија. Размотримо позицију $k-1$. Један по један елемент дела низа са позиција $[0, k)$ треба да доводимо на место $k - 1$ и да рекурзивно позивамо пермутовање дела низа на позицијама $[0, k-1)$. Идеја која се природно јавља је да вршимо размену елемента на позицији $k-1$ редом са свим елементима из интервала $[0, k)$ и да након сваке размене вршимо рекурзивне позиве.

Пример. На пример, ако је низ на почетку 123, онда мењамо елемент 3 са елементом 1, добијамо 321 и позивамо рекурзивно генерисање пермутација низа 32 са фиксираним елементом 1 на крају. Затим у почетном низу мењамо елемент 3 са елементом 2, добијамо 132 и позивамо рекурзивно генерисање пермутација низа 13 са фиксираним елементом 2 на крају. Затим у почетном низу мењамо елемент 3 са самим собом, добијамо 123 и позивамо рекурзивно генерисање пермутација низа 12 са фиксираним елементом 3 на крају.

Овај поступак је приказан и на слици.



Слика 6.11: Рекурзивно генерисање пермутација низа 123 коришћењем размена елемената низа

Међутим, са тим приступом може бити проблема. Наиме, да бисмо били сигурни да ће на последњу позицију стизати сви елементи низа, размене морамо да вршимо у односу на *почетно* стање низа. Један начин је да се пре сваког рекурзивног позива прави копија низа, али постоји и ефикасније решење. Наиме, можемо као инваријанту функције наметнути да је након сваког рекурзивног позива распоред елемената у низу исти као пре позива функције. Уједно то треба да буде и инваријанта петље у којој се врше размене. На уласку у петљу распоред елемената у низу биће исти као на уласку у функцију. Вршимо прву размену, рекурзивно позивамо функцију и на основу инваријанте рекурзивне функције знамо да ће распоред након рекурзивног позива бити исти као пре њега. Да бисмо одржали инваријанту петље, потребно је низ вратити у почетно стање. Међутим, знамо да је низ промењен само једном разменом, тако да је довољно урадити исту ту размену и низ ће бити враћен у почетно стање. Тиме је инваријанта петље очувана и може се прећи на следећу позицију. Када се петља заврши, на основу инваријанте петље знаћемо да је низ исти као на улазу у функцију. На основу тога знамо и да ће инваријанта функције бити одржана и није потребно урадити ништа додатно након петље.

```
void obradiSvePermutacije(vector<int>& permutacija, int k) {
  if (k == 1)
    obradi(permutacija);
  else {
    for (int i = 0; i < k; i++) {
      swap(permutacija[i], permutacija[k-1]);
      obradiSvePermutacije(permutacija, k-1);
      swap(permutacija[i], permutacija[k-1]);
    }
  }
}
```

```
void obradiSvePermutacije(int n) {
  vector<int> permutacija(n);
  for (int i = 1; i <= n; i++)
    permutacija[i-1] = i;
  obradiSvePermutacije(permutacija, n);
}
```

Библиотечка функција за следећу пермутацију

У језику C++ функција `next_permutation` декларисана у заглављу `<algorithm>` одређује наредну пермутацију у односу на дату. Функцији се прослеђују два итератора који ограничавају распон елемената у којима се налази пермутација.

```
// inicijalizujemo permutaciju na 1, 2, ..., n
vector<int> permutacija(n);
iota(begin(permutacija), end(permutacija), 1);

// ispisujemo permutaciju i trazimo narednu, sve dok naredna postoji
do {
  obradi(permutacija);
} while (next_permutation(begin(permutacija), end(permutacija)));
```

Глава 7

Бектрекинг и груба сила

У наставку ћемо разматрати проблеме следећег типа:

- Испитати да ли постоји неки комбинаторни објекат (често представљен торком бројева) који задовољава неке дате услове. Овакви проблеми се називају *проблеми задовољавања ограничења* (енгл. constraint satisfaction problems, CSP). На пример, потребно је проверити да ли постоји неки подскуп датог скупа бројева чији је збир једнак датој вредности.
- Међу свим комбинаторним објектима који задовољавају неке дате услове наћи најбољи тј. наћи онај објекат на коме је вредност неке дате функције минимална или максимална. Овакви проблеми се називају *проблеми оптимизације уз ограничења* (енгл. constraint optimization problems, COP). Ови проблеми се називају и проблеми *комбинаторне оптимизације*.

Овакви проблеми се често решавају разним варијантама алгоритама претраге у којима се набрајају и проверавају неки кандидати за решења.

Алгоритми грубе силе подразумевају да се у претрази исцрпно наброје сви кандидати за решење и да се за сваког од њих провери да ли задовољава услов (ако је потребно пронаћи било који елемент који задовољава дати услов) или да ли је оптималан (ако је потребно пронаћи елемент који минимализује или максимализује дату циљну функцију). Сви кандидати се понекада могу набројати једноставно, угнежђеним петљама, док је некада потребно користити неки сложенији поступак за набрајање одређене фамилије комбинаторних објеката (комбинација, пермутација, варијација, подскупова, партиција итд. али и других, специфичних фамилија комбинаторних објеката).

Алгоритми претраге са повратком тј. **бектрекинг** (енгл. backtracking) побољшавају технику грубе силе тако што врше провере и током генерисања кандидата за решења и тако што се одбацују парцијално попуњени кандидати, за које се може унапред утврдити да се не могу проширити до исправног тј. оптималног решења. Дакле, бектрекинг подразумева да се током обиласка у дубину дрвета којим се представља простор потенцијалних решења одсецају они делови дрвета за које се унапред може утврдити да не садрже ни једно решење проблема тј. да не садрже оптимално решење, при чему се одсецање врши и у чворовима блиским корену који могу да садрже и само парцијално попуњене кандидате за решења. Дакле, уместо да се чека да се током претраге стигне до листа (или евентуално унутрашњег чвора који представља неког кандидата за решење) и да се провера задовољености услова или оптималности врши тек тада, приликом претраге са повратком провера се врши у сваком кораку и врши се провера парцијално попуњених решења (обично су то неке парцијално попуњене торке бројева).

На пример, ако се проверава да ли постоји подскуп неког скупа чији је збир елемената једнак датом броју и ако се установи да прва два елемента полазног скупа имају збир већи од тог броја, тај део простора претраге се одмах може одсећи и нема потребе генерисати све подскупове у којима су та два прва елемента укључена (јер унапред знамо да ниједан од њих неће представљати задовољавајуће решење).

Ефикасност алгоритама заснованог на овом облику претраге увелико зависи од квалитета критеријума на основу којих се врши одсецање. Иако обично сложеност најгорег случаја остаје експоненцијална (каква је по правилу код алгоритама грубе силе тј. исцрпне претраге), пажљиво одабрани критеријуми одсецања могу одсећи јако велике делове претраге (који су често такође експоненцијалне величине у односу на димензије улазног проблема) и тиме значајно убрзати процес претраге.

Нагласимо и да неки аутори не праве експлицитну разлику између алгоритама грубе силе (исцрпне претраге) и алгоритама претраге са повратком и да у алгоритме типа претраге са повратком убрајају све алгоритме у којима се дрво које садржи сва потенцијална решења обилази у дубину.

Формулишимо општу схему рекурзивне имплентације претраге са повратком. Претпостављамо, једноставности ради, да су параметри процедуре претраге тренутни низ `v` (у коме се смештају торке бројева који представљају кандидате за решења) и дужина тренутно попуњеног дела низа `k`, при чему је низ алоциран тако да се у њега може сместити и најдуже решење. Такође, претпостављамо да на располагању имамо функцију `odsecanje` која проверава да ли је тренутна торка смештена у низ (на првих `k` позиција) кандидат да буде решење или део неког решења. Претпостављамо и да знамо да ли тренутна торка представља решење (то утврђујемо функцијом `jestePotencijalnoResenje`, међутим, у реалним ситуацијама се тај услов често сведе или на то да је увек тачан, што се дешава када је сваки чвор дрвета потенцијални кандидат за решење или на то да је тачан само у листовима, што се детектује тако што се провери да је `k` достигло дужину низа `v`). На крају, претпостављамо и да за сваку торку дужине `k` можемо експлицитно одредити све кандидате за вредност на позицији `k` (позивом функције `kandidati(v, k)`). Рекурзивну претрагу тада можемо реализовати наредним (псеудо)кодом.

```
void pretraga(const vector<int>& v, int k) {
   if (odsecanje(v, k))
      return;
   if (jesteResenje(v, k))
      ispisi(v, k);
   for (int x : kandidati(v, k)) {
      v[k] = x;
      pretraga(v, k+1);
   }
}
```

Алтернативно, провере уместо на улазу у функцију можемо вршити пре рекурзивних позива (чиме се мало штеди на броју рекурзивних позива, али се понекад имплементација може мало закомпликовати).

```
void pretraga(vector<int>& v, int k) {
   if (jesteResenje(v, k))
       ispisi(v, k);
   for (int x : kandidati(v, k)) {
      v[k] = x;
      if (!odsecanje(v, k+1)) {
         pretraga(v, k+1);
      }
   }
}
```

Рекурзије је могуће ослободити се уз коришћење стека.

У свим чворовима који представљају кандидате за решење (то су обично потпуно попуњене торки бројева) потребно је потпуно прецизно испитати да ли је текући кандидат исправно тј. оптимално решење. То значи да у претходном коду функција `jesteResenje` мора потпуно прецизно да детектује да ли трентуна торка јесте или није исправно тј. оптимално решење (нити сме исправно решење да прогласи неисправним, јер ће се тада оно пропустити, нити сме неисправно решење да прогласи исправним, јер ће се тада оно грешком појавити у списку решења). Са друге стране, у чворовима у којима се проверавају парцијално попуњена решења, провера критеријума одсецања не мора бити потпуно прецизна — са једне стране допуштено је да се не одсеку делови дрвета у којима нема решења (тиме алгоритам остаје коректан, али је неефикаснији), међутим, ако се одсецање изврши морамо бити апсолутно сигурни да се у одсеченом делу дрвета не налази ниједно исправно решење тј. да се не налази се оптимално решење. У претходном коду то значи да када функција `odsecanje` врати вредност тачно, морамо бити апсолутно сигурни да се тренутна торка смештена на првих `k` позиција у низу `v` не може никако допунити до исправног тј. оптималног решења (јер бисмо у супротном пропустили нека решења). Са друге стране, та функција може да врати вредност нетачно практично било када и тиме неће бити нарушена коректност (али се нарушава ефикасност). У пракси се понекад дешава да је веома компликовано направити функцију `odsecanje` која потпуно прецизно одређује да ли се торка може продужити до траженог решења проблема тако да се задовољавамо критеријумима одсецања који се могу релативно једноставно проверити, а гарантују коректност.

Додатно убрзавање алгоритма може да се направи ако се на неки начин може дефинисати функција која понекад може да погоди вредност x коју треба уписати на позицију k, без испробавања различитих кандидата. На пример, ако се приликом попуњавања магичног квадрата (квадрата у ком су бројеви распоређени тако да све врсте, све колоне и обе дијагонале имају исти, унапред познат, збир) у некој врсти попуне сви елементи осим једног, лако можемо да израчунамо која вредност мора да буде уписана на том преосталом месту. Такве кораке зовемо кораци **закључивања** (енгл. inferrence). Ни у овом случају није потребна потпуна прецизност. Само је битно обезбедити да када се закључивањем предложи нека конкретна вредност, да је остале могућности безбедно одсећи, јер се међу њима не крије ниједно исправно решење. Са друге стране, ако је закључивање превише компликовано остварити у неком кораку претраге, оно се може прескочити (алгоритам је коректан и без икаквог облика закључивања).

Претходне функције исписују сва решења проблема тј. све торке које задовољавају дате услове. Претрагу је могуће прекинути и након проналаска првог решења (тада функција обично враћа податак о томе да ли јесте или није пронашла решење у делу дрвета који претражује).

```
bool pretraga(const vector<int>& v, int k) {
   if (odsecanje(v, k))
      return false;
   if (jesteResenje(v, k)) {
      ispisi(v, k);
      return true;
   }
   for (int x : kandidati(v, k)) {
      v[k] = x;
      if (pretraga(v, k+1))
         return true;
   }
   return false;
}
```

С обзиром на то да се исцрпна претрага, али и претрага са одсецањем често реализују алгоритмима претраге у дубину и у ширину, приказаћемо неколико задатака који користе ове алгоритме.

Код решавања оптимизационих проблема, одсецање може да наступи и када се процени да у делу дрвета које се тренутно претражује не постоји решење које је боље од најбољег тренутно пронађеног решења. Дакле, решења пронађена у досадашњем делу претраге се користе да би се одредиле границе на основу којих се врши одсецање у другим деловима претраге. Овакав облик оптимизације назива се понекад **гранање са ограничавањем** (енгл. branch and bound).

Задатак: Број белих области

Написати програм којим се за црно-белу матрицу (0 – бела, 1 – црна боја) одређује број белих области. Белу област чине повезана бела поља. Два бела поља су повезана ако су она почетно и крајње поље неког низа белих поља у коме узастопна поља имају заједничку страницу.

Улаз: У првој линији стандардног улаза се учитава број редова матрице n ($2 \leq n \leq 20$), у другој број колона m ($2 \leq m \leq 20$). У следећих n редова учитава се по m бројева чија је вредност 0 или 1.

Излаз: Број белих области.

Пример

Улаз	*Излаз*
5 6 1 1 1 1 1 0 1 1 0 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0	3

Решење

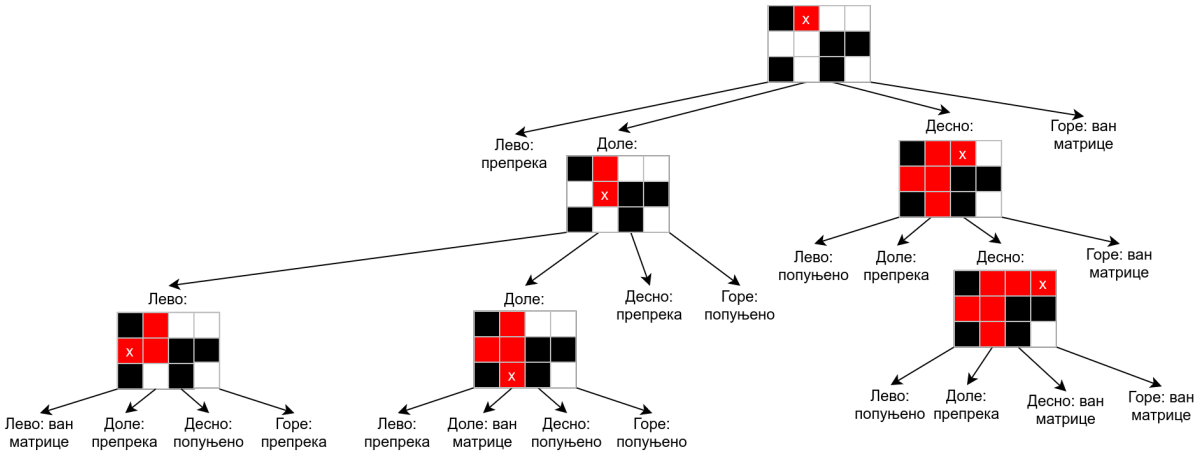
Основу решења чини рекурзивна функција која детектује и обележава белу област почевши од једног њеног поља. Функција се позива за поље у матрици које је тренутно обојено бело (у матрици на том пољу пише 0), обележава га (уписује неку другу вредност, на пример, -1), и затим се рекурзивно позива за све његове суседе беле боје (при том треба водити рачуна о томе да суседни случајно не испадну из матрице).

У главном програму обилазимо сва поља матрице (коришћењем угнежђених петљи) и за свако бело поље (оно је део неке нове области) позивамо рекурзивну функцију која ће детектовати и обележити сва поља те области и увећавамо бројач обележених области. Када се обиђе цела матрица, све беле области су обележене, па можемо исписати њихов број.

Пример. Рекурзивни позиви се могу представити дрветом. Размотримо матрицу:

```
1000
0011
1010
```

Рекурзивни позив за поље $(0, 0)$ се не врши, јер је поље црно. Након тога се врши рекурзивни позив за поље $(0, 1)$ и тада се велика бела област пролази и обележава. Дрво рекурзивних позива (каже се и дрво претраге) је приказано на слици.



Слика 7.1: Дрво рекурзивних позива - текуће поље је обележно са x, а обележена поља су обојена црвеном бојом

Након тога, се редом обилазе поља све до оног у доњем десном углу и пошто су сва или црна или већ обележена, за њих се не врше рекурзивни позиви. На крају се врши позив за поље $(3, 4)$ којим се обележева преостало бело поље.

```
// maksimalne dimenzije matrice
const int N = 20, M = 20;

// oznake polja u matrici
const int BELA = 0, CRNA = 1, OBELEZENO = -1;

// provera da li se polje (x, y) nalazi u matrici dimenzije mxn
bool UMatrici(int x, int y, int m, int n) {
  return x >= 0 && x < m && y >= 0 && y < n;
}

// obilazi se belo, neobelezeno polje sa koordinatama (x, y)
void Obelezi(int x, int y, int a[M][N], int m, int n) {
  // obelezavamo polje
  a[x][y] = OBELEZENO;
  // obilazimo sve susede
  int dx[] = { -1, 0, 1, 0 };
  int dy[] = { 0, 1, 0, -1 };
```

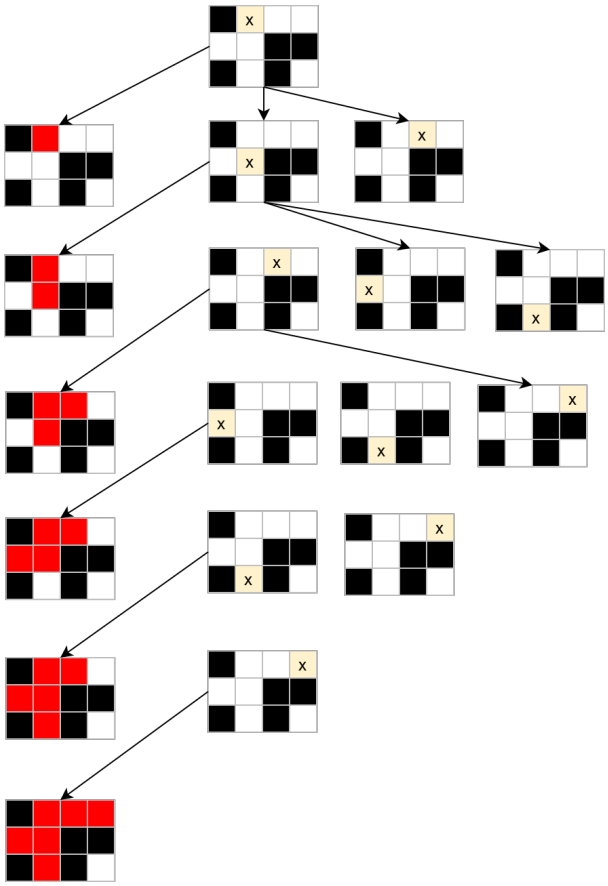
```
  for (int i = 0; i < 4; i++) {
    int xx = x + dx[i], yy = y + dy[i];
    // ako je susedno polje belo (i neobelezeno), obelezavamo ga
    if (UMatrici(xx, yy, m, n) && a[xx][yy] == BELA)
      Obelezi(xx, yy, a, m, n);
  }
}

int BrojOblasti(int a[M][N], int m, int n) {
  int oblast = 0;   // broj obelezenih oblasti
  // obilazimo sva polja u matrici
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      if (a[i][j] == BELA) {
        Obelezi(i, j, a, m, n);
        oblast++;
      }
  return oblast;
}
```

Претрагу у дубину је могуће имплементирати и уз помоћ стека.

```
// obilazi se belo, neobelezeno polje sa koordinatama (x0, y0)
void Obelezi(int x0, int y0, int a[M][N], int m, int n) {
  stack<pair<int, int>> obeleziti;
  obeleziti.emplace(x0, y0);
  while (!obeleziti.empty()) {
    auto p = obeleziti.top(); obeleziti.pop();
    int x = p.first, y = p.second;
    // obelezavamo polje
    a[x][y] = OBELEZENO;
    // obilazimo sve susede
    int dx[] = { -1, 0, 1, 0 };
    int dy[] = { 0, 1, 0, -1 };
    for (int i = 0; i < 4; i++) {
      int xx = x + dx[i], yy = y + dy[i];
      // ako je susedno polje belo (i neobelezeno), obelezavamo ga
      if (UMatrici(xx, yy, m, n) && a[xx][yy] == BELA)
        obeleziti.emplace(xx, yy);
    }
  }
}
```

Обилазак сваке беле области можемо реализовати и претрагом у ширину. Претрага у ширину се реализује уз помоћ реда.

Пример. На наредној слици је приказан поступак бојења једне беле области у матрици коришћењем претраге у ширину. Са десне стране приказан је садржај реда, из корака у корак, а са леве стране бојење, које се добија вађењем једног по једног елемента са почетка реда. Приметимо да се прво боји полазно поље, затим поља која су на растојању од њега, а након тога поља која су на растојању два од њега.



Слика 7.2: Обилазак матрице у ширину

```
// obilazi se belo, neobelezeno polje sa koordinatama (x, y)
void Obelezi(int x0, int y0, int a[N][M], int m, int n) {
  queue<pair<int, int>> obeleziti;
  obeleziti.emplace(x0, y0);
  while (!obeleziti.empty()) {
    auto p = obeleziti.front(); obeleziti.pop();
    int x = p.first, y = p.second;
    // obelezavamo polje
    a[x][y] = OBELEZENO;
    // obilazimo sve susede
    int dx[] = { -1, 0, 1, 0 };
    int dy[] = { 0, 1, 0, -1 };
    for (int i = 0; i < 4; i++) {
      int xx = x + dx[i], yy = y + dy[i];
      // ako je susedno polje belo (i neobelezeno), obelezavamo ga
      if (UMatrici(xx, yy, m, n) && a[xx][yy] == BELA)
        obeleziti.emplace(xx, yy);
    }
  }
}
```

Задатак: Minesweeper отварање

Напиши програм који приказује садржај поља у игрици Minewsweeper (ако не знаш како изгледа, потражи на интернету) након отварања једног поља.

Улаз: Са стандардног улаза се учитава матрица димензије 10 пута 10 која на пољима где су бомбе садржи јединице, а на слободним пољима садржи нуле. У наредном реду се учитавају координате поља које се отвара

(број врсте и број колоне између 0 и 9, раздвојени једним размаком).

Излаз: На стандардни излаз исписати стање које се кориснику приказује након отварања тог поља. Ако је на пољу бомба, исписати `boom`. У супротном приказати матрицу димензије 10 пута 10 тако да се на неотвореним пољима приказује `x` на празним пољима (пољима која су отворена и немају бомби у околини) приказује `.`, а на отвореним пољима која имају бомбе у околини приказује број бомби у околини.

Пример

Улаз	*Излаз*
0100010100	xxxxxxxxxx
0100111000	xxxxxxxxxx
1000000000	xx21233xxx
1100000110	xx1...1xxx
1000000000	xx2...2xxx
1100000100	xx421.1xxx
0111000011	xxxx113xxx
1100001110	xxxxxxxxxx
0011100000	xxxxxxxxxx
1110000001	xxxxxxxxxx
5 5	

Решење

На основу учитане матрице бомби израчунавамо број бомби у околини сваког поља и те бројеве записујемо у помоћну матрицу.

Користимо алгоритам претраге у дубину. Дефинишемо рекурзивну функцију за отварање поља. У другој помоћној матрици региструјемо која су поља отворена (у почетку су сва поља затворена). Ако приликом отварања неког поља установимо да у његовој околини нема бомби, тада рекурзивно аутоматски отварамо сва поља у његовој околини. Када се такав обилазак у дубину заврши, исписујемо резултат (у двострукој петљи). Ако за поље установимо да је затворено исписујемо `x`, а ако је отворено проверавамо да ли му је број бомби у околини једнак нули и тада исписујемо `.`, док у супротном исписујемо број бомби у околини.

Пошто је димензија поља увек 10 пута 10, можемо користити и статички алоциране матрице.

```
void otvoriPolje(int okoloBombi[DIM][DIM], bool otvoreno[DIM][DIM],
                 int v, int k) {
  otvoreno[v][k] = true;
  if (okoloBombi[v][k] == 0) {
    for (int dv = -1; dv <= 1; dv++)
      for (int dk = -1; dk <= 1; dk++) {
        if (dv == 0 && dk == 0) continue;
        int v1 = v + dv, k1 = k + dk;
        if (0 <= v1 && v1 < DIM && 0 <= k1 && k1 < DIM &&
            !otvoreno[v1][k1])
          otvoriPolje(okoloBombi, otvoreno, v1, k1);
      }
  }
}
```

Процедура отварања поља може бити реализована и нерекурзивно, коришћењем стека. На стек стављамо почетно поље и понављамо следећи поступак док се стек не испразни. Скидамо текуће поље са врха стека, отварамо га и анализирамо његова околна поља. Свако његово нетворено околоно поље које се налази унутар граница матрице и које нема бомби у својој околини стављамо на стек.

```
void otvoriPolje(int okoloBombi[DIM][DIM], bool otvoreno[DIM][DIM],
                 int v0, int k0) {
  stack<pair<int, int>> otvoriti;
  otvoriti.emplace(v0, k0);
  while (!otvoriti.empty()) {
    auto p = otvoriti.top(); otvoriti.pop();
    int v = p.first, k = p.second;
    otvoreno[v][k] = true;
```

```
    if (okoloBombi[v][k] == 0) {
      for (int dv = -1; dv <= 1; dv++)
        for (int dk = -1; dk <= 1; dk++) {
          if (dv == 0 && dk == 0) continue;
          int v1 = v + dv, k1 = k + dk;
          if (0 <= v1 && v1 < DIM && 0 <= k1 && k1 < DIM &&
              !otvoreno[v1][k1])
            otvoriti.emplace(v1, k1);
        }
    }
  }
}
```

Задатак: Пут кроз лавиринт

Напиши програм који испитује да ли се у правоуаоном лавиринту може стићи од горњег левог до доњег десног угла.

Улаз: Са стандардног улаза се учитавају димензије лавиринта m и n раздвојене једним размаком. Након тога се учитава матрица којом је представљен лавиринт. Поља кроз која се може проћи су обележена карактером `.`, а поља на којима је препрека карактером `x`.

Излаз: На стандардни излаз исписати `da` ако пут постоји тј. `ne` ако пут не постоји.

Пример 1

Улаз
```
8 8
.x.....x
.x.x.x.x
.x.x.x.x
.x.x.x.x
.x.x.x.x
.x.x.x.x
.x.x.x.x
...x.x..
```

Излаз
```
da
```

Пример 2

Улаз
```
8 8
.x..x..x
.x.x.x.x
.x.x.x.x
.x.x.x.x
.x...x.x
.x.x.x.x
.x.x.x.x
...x.x..
```

Излаз
```
ne
```

Решење

Претрага у дубину

Задатак решавамо исцрпном претрагом свих могућих путања. Претрагу можемо организовати "у дубину" помоћу рекрузивне функције. Функција прима матрицу препрека и поље на ком се тренутно налазимо, које се мења током рекурзије. Ако је стартно поље поклапа са циљним (доњим десним углом), пут је успешно пронађен. У супротном, испитујемо 4 суседа стартног поља (осим у случају поља на рубу лавиринта, када је суседа мање) и претрагу рекурзивно настављамо од сваког од њих (суседно поље постаје ново стартно поље). Ако се на пољу на које смо прешли налази препрека, претрагу моментално прекидамо јер тај корак није дозвољен (функција враћа да на тај начин није могуће пронаћи пут). Потребно је и да обезбедимо да се већ посећена стартна поља не обрађују поново и за то користимо помоћну матрицу у којој за свако поље региструјемо да ли је посећено или није. Ако приликом позива функције установимо да је поље на које смо дошли већ раније посећено, претрагу одмах прекидамо, док у супротном означавамо да је то поље посећено и прелазимо на анализу њега и његових суседа.

```
bool postojiPut(const vector<vector<bool>>& prepreke, int m, int n,
                vector<vector<bool>>& poseceno,
                int v, int k) {

  if (prepreke[v][k] || poseceno[v][k])
    return false;

  poseceno[v][k] = true;
```

```
  if (v == m - 1 && k == n - 1)
    return true;

  if (v > 0 && postojiPut(prepreke, m, n, poseceno, v-1, k))
    return true;
  if (v < m-1 && postojiPut(prepreke, m, n, poseceno, v+1, k))
    return true;
  if (k > 0 && postojiPut(prepreke, m, n, poseceno, v, k-1))
    return true;
  if (k < n-1 && postojiPut(prepreke, m, n, poseceno, v, k+1))
    return true;
  return false;
}

bool postojiPut(const vector<vector<bool>>& prepreke, int m, int n) {
  vector<vector<bool>> poseceno(m, vector<bool>(n, false));
  return postojiPut(prepreke, m, n, poseceno, 0, 0);
}
```

Претрагу у дубину је могуће имплементирати и нерекурзивно, тако што експлицитно одржавамо стек.

Постоји још неколико једноставних трикова који могу олакшати имплементацију. Четири суседна поља можемо обрађивати у петљи (тако што у низу сачувамо 4 помераја $(-1, 0)$, $(1, 0)$, $(0, -1)$ и $(0, 1)$, чиме избегавамо понављање сличног кода 4 пута. Да би се приликом обиласка 4 суседа избегло испитивање да ли ти суседи постоје тј. да ли је текуће поље на рубу, можемо матрицу проширити оквиром који садржи препреке.

```
bool postojiPut(const Matrica<bool>& prepreke, int m, int n) {
  Matrica<bool> poseceno = napraviMatricu(m + 2, n + 2, false);
  poseceno[1][1] = true;

  stack<pair<int, int>> stek;
  stek.emplace(1, 1);

  while (!stek.empty()) {
    int v = stek.top().first, k = stek.top().second;
    stek.pop();

    if (v == m && k == n)
      return true;

    int pravac[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    for (int i = 0; i < 4; i++) {
      int v1 = v + pravac[i][0], k1 = k + pravac[i][1];
      if (!prepreke[v1][k1] && !poseceno[v1][k1]) {
        poseceno[v1][k1] = true;
        stek.emplace(v1, k1);
      }
    }
  }
  return false;
}
```

Претрага у ширину

Уместо претраге у дубину, могуће је употребити и претрагу у ширину. Имплементација је веома слична нерекурзивно имплементираној претрази у дубину, при чему се уместо стека користи ред, чиме се постиже да се поља обрађују у редоследу њиховог растојања од полазног поља, што омогућава да се лако очита и дужина најкраћег пута до крајњег поља.

```
bool postojiPut(const Matrica<bool>& prepreke, int m, int n) {
  Matrica<bool> poseceno = napraviMatricu(m, n, false);
```

```
  queue<pair<int, int>> red;
  red.emplace(0, 0);
  while (!red.empty()) {
    int v = red.front().first, k = red.front().second;
    red.pop();
    if (poseceno[v][k] || prepreke[v][k])
      continue;
    poseceno[v][k] = true;
    if (v == m-1 && k == n-1)
      return true;
    int pravac[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    for (int i = 0; i < 4; i++) {
      int v1 = v + pravac[i][0], k1 = k + pravac[i][1];
      if (0 <= v1 && v1 < m && 0 <= k1 && k1 < n)
        red.emplace(v1, k1);
    }
  }
  return false;
}
```

Задатак: Уклањање погрешних заграда

Ниска поред осталих карактера садржи и мале заграде (и), али могуће је да оне нису исправно упарене. Потребно је обрисати што мање карактера ниске да би се добила ниска у којој су заграде исправно упарене. Напиши програм који исписује све могуће ниске у којима су заграде исправно упарене а које су добијене брисањем што мањег броја карактера.

Улаз: Са стандардног улаза се учитава ниска дужине највише 50 карактера.

Излаз: На стандардни излаз исписати све тражене ниске у лексикографском редоследу.

Пример 1

Улаз	*Излаз*
`()())()`	`(())()` `()()()`

Пример 2

Улаз	*Излаз*
`(abc()+def))(`	`(abc()+def)` `(abc(+def))`

Решење

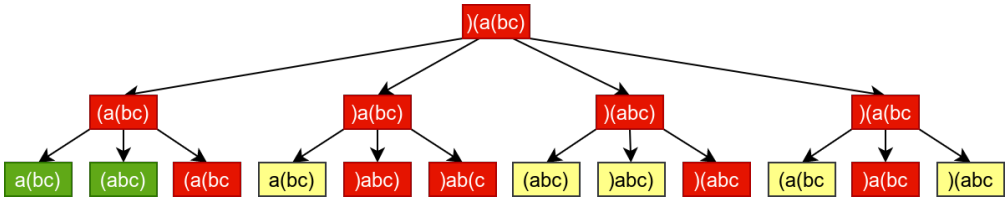
Задатак можемо решити претрагом у ширину. Претпоставићемо да су полазна ниска и ниске које се од ње добијају брисањем заграда (друге карактере нема смисла брисати, јер не утичу на упареност заграда) стања кроз која се пролази. Полазно стање је оно које одговара полазној ниски, а завршна стања су сва она која одговарају нискама са исправно упареним заградама. Тада се задатак решава слично проналажењу најкраћег пута кроз лавиринт. Оно је описано у задатку Пут кроз лавиринт.

Дефинисаћемо помоћну функцију која проверава да ли су у датој листи заграде исправно упарене. Довољно је одржавати бројач отворених заграда, увећавати га при наиласку на отворену заграду, умањивати га при наиласку на затворену заграду и водити рачуна да никада не постане негативан а да на крају дође на нулу.

Стања (ниске) стављамо у ред, кренувши од почетне ниске. Након тога узимамо ниске из реда, све док се ред не испразни или док се на почетку реда не појави нека ниска која добијена уклањањем већег броја заграда него што је то неопходно. За сваку узету ниску проверавамо да ли су у њој заграде исправно упарене. Ако јесу, ниску смештамо у скуп ниски које треба на крају исписати у лексикографски сортираном редоследу (за то можемо користити библиотечку колекција за представљање сортираних скупова). У том тренутку престајемо да додајемо нове ниске у ред (јер се у реду сигурно већ налазе оне ниске које су добијене брисањем истог броја заграда као ова текућа ниска, док би се у ред даље додавале само речи добијене већим бројем избрисаних заграда). У супротном пролазимо кроз ниску, уклањамо једну по једну заграду и све тако добијене ниске смештамо у ред.

Једна могућа оптимизација се може направити ако се примети да се у ред често убацују потпуно идентичне ниске, за чим очигледно нема потребе. Ако се нека ниска добијена избацивањем неке заграде већ налази у реду (што можемо утврдити одржавањем скупа ниски које се налазе у реду), нема је потребе додатно убацивати у ред.

На слици је приказано дрво претраге у ширину за ниску `)(a(bc)`. Бојама су обележене исправне ниске, неисправне ниске и ниске које нису убачене у ред, јер су поновљене.



Слика 7.3: Уклањање погрешних заграда

Иако је ово решење могуће додатно оптимизовати, с обзиром на релативно малу димензију улаза, за тим нема потребе.

```
// provera da li su u niski s zagrade ispravno uparene
bool ispravneZagrade(const string& s) {
  int otvorenoZagrada = 0;
  for (char c : s)
    if (c == '(')
      otvorenoZagrada++;
    else if (c == ')') {
      if (otvorenoZagrada == 0)
        return false;
      otvorenoZagrada--;
    }
  return otvorenoZagrada == 0;
}

// grade se sve niske sa ispravno uparenim zagradama koje se od s
// dobijaju uklanjanjem najmanjeg moguceg broja zagrada
set<string> uklanjanjePogresnihZagrada(const string& s) {
  // skup svih resenja
  set<string> resenja;
  // skup niski koje su vec ubacene u red (da bi se izbegli duplikati)
  set<string> uRedu;
  // red u kome pamtimo niske sa izbacenim zagradama i broj izbacenih zagrada za svaku nisku
  queue<pair<string, int>> red;
  // krecemo od polazne niske
  red.emplace(s, 0);
  uRedu.insert(s);
  // minimalan broj zagrada koje treba ukloniti da bi se dobilo ispravno resenje
  // vrednost -1 oznacava da je taj broj nepoznat
  int minimalanBrojUklonjenih = -1;
  // dok se red ne isprazni
  while (!red.empty()) {
    // skidamo nisku sa pocetka reda
    auto p = red.front(); red.pop();
    string s = p.first; int brojUklonjenih = p.second;

    // naisli smo na niske u kojima je uklonjeno vise zagrada nego sto
    // je potrebno pa mozemo prekinuti postupak
    if (minimalanBrojUklonjenih != -1 && brojUklonjenih > minimalanBrojUklonjenih)
      break;

    // proveravamo da li su u trenutnoj niski zagrade ispravno uparene
    if (ispravneZagrade(s)) {
      // pamtimo najmanji broj zagrada koje je bilo potrebno ukloniti
```

```
      if (minimalanBrojUklonjenih == -1)
        minimalanBrojUklonjenih = brojUklonjenih;
      // ovo je jedno od trazenih resenja
      resenja.insert(s);
    }
    // ako jos nismo pronasli ispravnu nisku
    if (minimalanBrojUklonjenih == -1) {
      // izbacujemo jednu po jednu zagradu iz tekuce niske i dobijene niske ubacujemo u red
      for (int i = 0; i < s.size(); i++) {
        if (s[i] == '(' || s[i] == ')') {
          // izbacujemo zagradu na poziciji i
          string t = s.substr(0, i) + s.substr(i+1);
          // proveravamo da li je dobijena niska mozda jos ranije ubacena u red
          if (uRedu.find(t) == uRedu.end()) {
            red.emplace(t, brojUklonjenih+1);
            uRedu.insert(t);
          }
        }
      }
    }
  }

  // vracamo skup resenja
  return resenja;
}
```

Задатак: Подела на палиндромске подниске

Напиши програм који одређује све начине да се ниска подели на узастопне непразне подниске које су палиндроми (читају се исто са лева и са десна).

Улаз: Са стандардног улаза се учитава ниска која има највише 50 карактера енглеског алфабета.

Излаз: На стандардни излаз исписати све тражене поделе ниске на палиндромске подниске (подниске писати у истом реду, раздвојене са по једним размаком). Ако постоји више решења, исписивати прво она која почињу краћим поднискама.

Пример

Улаз	*Излаз*
`sirimamiris`	`s i r i m a m i r i s`
	`s i r i m a m iri s`
	`s i r i mam i r i s`
	`s i r i mam iri s`
	`s i r imami r i s`
	`s i rimamir i s`
	`s iri m a m i r i s`
	`s iri m a m iri s`
	`s iri mam i r i s`
	`s iri mam iri s`
	`s irimamiri s`
	`sirimamiris`

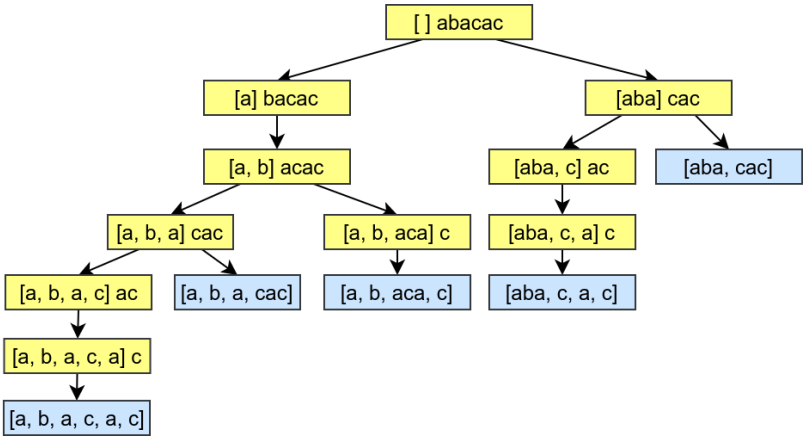
Решење

С обзиром на релативно малу дужину улазне ниске, задатак можемо решити рекурзивном исцрпном претрагом. Функција врши рекурзивну претрагу и прима низ палиндрома који је добијен разлагањем неког почетног дела ниске, као и преостали део ниске који треба разложити на палиндромске подниске. Иницијално је тај низ палиндрома празан и потребно је разложити целу ниску. Ако је преостали део ниске празан, тада је цела ниска разложена и исписујемо палиндроме смештене у низ. У супротном, анализирамо све префиксе преосталог дела (кренувши од једночланог) и сваки од њих који је палиндром уклањамо из ниске, премештамо га

у низ и настављамо рекурзивно разлагање дела ниске иза тог префикса.

За проверу да ли је тренутни префикс палиндром дефинисаћемо помоћну функцију која проверава да ли је подниска одређена позицијама првог и последњег слова палиндром.

На слици је приказано дрво претраге за ниску `abacac`. У сваком чвору је прво приказан низ палиндрома, а затим и преостали део ниске који треба разложити на палиндроме.



Слика 7.4: Дрво претраге

```
bool jePalindrom(const string& s, int Od, int Do) {
  for (int i = Od, j = Do; i < j; i++, j--)
    if (s[i] != s[j])
      return false;
  return true;
}

void podelaNaPalindrome(const string& s, int i, vector<string>& palindromi) {
  if (i == s.length()) {
    for (const string& s : palindromi)
      cout << s << " ";
    cout << endl;
  } else {
    for (int j = i; j < s.length(); j++)
      if (jePalindrom(s, i, j)) {
        palindromi.push_back(s.substr(i, j-i+1));
        podelaNaPalindrome(s, j+1, palindromi);
        palindromi.pop_back();
      }
  }
}

void podelaNaPalindrome(const string& s) {
  vector<string> palindromi;
  podelaNaPalindrome(s, 0, palindromi);
}
```

Задатак: Збир суседних пун квадрат

Низ `8 1 15 10 6 3 13 12 4 5 11 14 2 7 9` је карактеристичан по томе што преставља пермутацију бројева од 1 до 15 и збир било која два суседна елемента је квадрат неког природног броја. Напиши програм који за дато n одређује лексикографски најмању пермутацију бројева од 1 до n у којој је збир било која два суседна елемента квадрат неког природног броја.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 45$).

Излаз: На стандардни излаз исписати тражену пермутацију (сви бројеви у истом реду праћени са по једним размаком) или текст `nema` ако тражена пермутација не постоји.

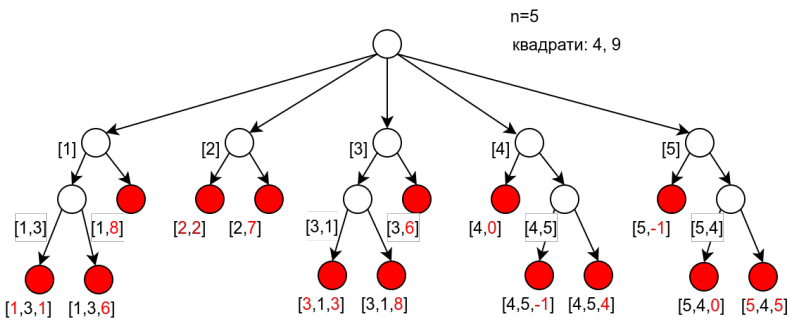
Пример 1		**Пример 2**	
Улаз	*Излаз*	*Улаз*	*Излаз*
`12`	`nema`	`17`	`16 9 7 2 14 11 5 4 12 13 3 6 10 15 1 8 17`

Решење

Задатак решавамо исцрпном бектрекинг претрагом. Тражени низ градимо елемент по елемент. На прво место уписујемо редом један по један елемент од 1 до n, а затим позивамо рекурзивну функцију да попуни остатак низа. Та функција на текуће место у низу покушава да постави све оне елементе који са претходним елементом низа дају потпун квадрат, а који се не јављају у до тада попуњеном делу низа. Низ свих пуних квадрата можемо израчунати унапред. Пошто се збирови формирају од различитих елемената из целобројног интервала $[1, n]$, најмањи квадрат мора бити већи од $1 + 2$ (па је једнак $2^2 = 4$, а највећи квадрат мора бити мањи или једнак од $(n - 1) + n$. Да бисмо ефикасно могли да проверимо да ли се неки елемент већ јавио у досадашњем делу низа, посебно ћемо одржавати скуп свих до тада попуњених елемената у облику низа логичких вредности таквог да је на месту a вредност тачно ако и само ако се елемент a јавља у попуњеном делу низа). Успешан излаз из рекурзије је када је цео низ попуњен (када текућа позиција која се попуњава постане једнака дужини низа).

Ако се дрво претраге обилази у дубину, прво решење на које се наиђе биће уједно и прво у лексикографском редоследу.

Пример. Ако је $n = 5$, разматрају се квадрати 4 и 9 (који је једнак максималном збиру $4+5$). Дрво претраге је приказано на слици (претрага се прекида у обојеним чворовима, јер се у њима добија или низ који садржи елемент ван интервала $[1, 5]$ или низ који садржи дупликате).



Слика 7.5: Претрага за $n = 5$

```
bool zbir_susednih_pun_kvadrat(vector<int>& niz, int m,
                               vector<bool>& iskoriscen, const vector<int>& kvadrati) {
  if (m == niz.size()) {
    ispisi(niz);
    return true;
  } else {
    for (int k : kvadrati) {
      int dopuna = k - niz[m-1];
      if (1 <= dopuna && dopuna <= niz.size() && !iskoriscen[dopuna]) {
        niz[m] = dopuna;
        iskoriscen[dopuna] = true;
        if (zbir_susednih_pun_kvadrat(niz, m+1, iskoriscen, kvadrati))
          return true;
        iskoriscen[dopuna] = false;
      }
    }
  }
  return false;
}
```

```
bool zbir_susednih_pun_kvadrat(int n) {
  vector<bool> iskoriscen(n+1, false);
  vector<int> niz(n);
  vector<int> kvadrati;
  for (int k = 2; k*k <= n+(n-1); k++)
    kvadrati.push_back(k*k);

  for (int i = 1; i <= n; i++) {
    niz[0] = i;
    iskoriscen[i] = true;
    if (zbir_susednih_pun_kvadrat(niz, 1, iskoriscen, kvadrati))
      return true;
    iskoriscen[i] = false;
  }
  return  false;
}
```

Задатак: Распоређивање n дама на шаховској табли

Напиши програм који одређује све положаје n дама на шаховској табли димензије $n \times n$ такве да се никоје две даме међусобно не нападају. Да се даме не би нападале у свакој врсти мора бити тачно једна дама, при чему никоје две даме нису у истој колони. Распоред је зато одређен низом од n различитих бројева од 1 до n који редом представљају бројеве колона у којима се даме налазе у врстама од 1 до n.

Улаз: Са стандардног улаза се уноси број n ($4 \leq n \leq 11$).

Излаз: На стандардни излаз исписати све могуће распореде дама (у произвољном редоследу).

Пример

Улаз	*Излаз*
4	3 1 4 2
	2 4 1 3

Решење

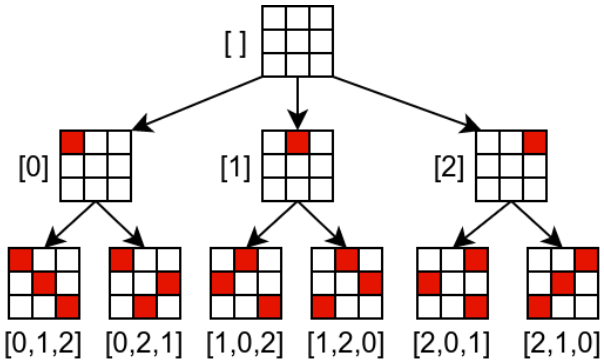
Груба сила - провера свих пермутација

Један (наиван) начин да се одреде сви могући распореди је да се грубом силом наброје сви могући распореди, да се испита који од њих представљају исправан распоред (у ком се даме не нападају) и да се испишу само они који тај критеријум задовољавају. Важно питање је репрезентација позиција.

Наиме, ако се у старту допусте сви могући распореди, онда сваки распоред представља једну комбинацију у којој се бира n елемената из скупа од n^2 елемената (свака дама је на једној од n позиција, а укупан број позиција је n^2). Међутим, многе од тих позиција сасвим очигледно не задовољавају услове задатка, јер се две даме налазе у истој врсти (или у истој колони).

Боље решење се добија ако се распореди представе пермутацијама елемената скупа $\{1, 2, \ldots, n\}$. Наиме ако се даме не нападају, у свакој врсти и у свакој колони се налази по тачно једна дама. Ако и врсте и колоне обележимо бројевима од 0 до $n-1$, тада је свакој врсти једнозначно придружена колона у којој се налази дама у тој врсти и распоред можемо представити низом тих бројева. Свакој врсти је придружена различита колона (јер даме не смеју да се нападају), тако да је заиста у питању пермутација. Пермутација n елемената има $n!$ што је знатно мање од $\binom{n^2}{n}$, али је и даље јако пуно.

Ова репрезентација одмах гарантује да се даме неће нападати ни хоризонтално, ни вертикално и једино је потребно одредити да ли се нападају по дијагонали. Две даме се налазе на истој дијагонали ако и само ако је хоризонтални размак између колона у којима се налазе једнак вертикалном размаку врста. За сваки пар дама проверавамо да ли се нападају дијагонално. Све пермутације можемо набројати на било који од начина описаних у задатку Све пермутације.

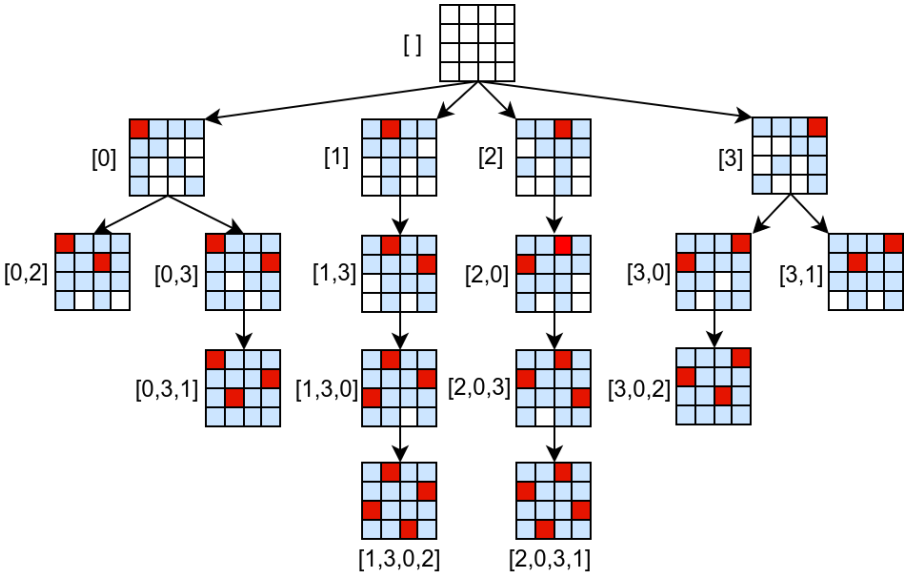


Слика 7.6: Распоређивање 3 даме - све пермутације и позиције које су њима представљене

```
// niz kolone sadrži kolone u kojima se nalaze dame u vrstama iz
// intervala [0, v]
// pretpostavlja se da se dame na pozicijama [0, v) ne napadaju i
// proverava se da li se dama na poziciji v napada sa nekom od njih
bool dameSeNapadaju(const vector<int>& kolone, int v) {
  for (int vi = 0; vi < v; vi++) {
    if (kolone[vi] == kolone[v])
      return true;
    if (abs(v-vi) == abs(kolone[v] - kolone[vi]))
      return true;
  }
  return false;
}

// niz kolone sadrži kolone u kojima se nalaze dame u vrstama iz
// intervala [0, v) za koji se pretpostavlja da predstavlja raspored
// dama koje se ne napadaju
// procedura ispisuje sve moguće raspored dama koje proširuju taj raspored
// i u kojima se dame ne napadaju
void nDama(vector<int>& kolone, int v) {
  // sve dame su postavljene
  if (v == kolone.size())
    ispisi(kolone);
  else {
    // postavljamo damu u vrstu v
    // isprobavamo sve moguće kolone
    for (int k = 0; k < kolone.size(); k++) {
      // postavljamo damu u vrsti v u kolonu k
      kolone[v] = k;
      // proveravamo da li se dame i dalje ne napadaju
      if (!dameSeNapadaju(kolone, v))
        // ako se ne napadaju, nastavljamo sa proširivanjem tekućeg rasporeda
        nDama(kolone, v+1);
    }
  }
}

void nDama(int n) {
   // niz koji za svaku vrstu beleži kolonu dame u toj vrsti
   vector<int> kolone(n);
   // krećemo pretragu od prazne table
   nDama(kolone, 0);
}
```

Бектрекинг и одсецање

Прикажимо сада решење проблема постављања n дама на шаховску таблу техноком бектрекинга. Основна разлика овог у односу на решење грубом силом је то што се коректност пермутација не проверава тек након што су генерисане у целости, већ се проверава коректност и сваке делимично попуњене пермутације. У многим случајевима веома рано ће бити откривено да су постављене даме на истој дијагонали и цела та грана претраге биће напуштена, што даје потенцијално велике добитке у ефикасности.



Слика 7.7: Распоређивање 4 даме - претрага са одсецањем

Основна рекурзивна функција прима вектор у коме се на позицијама из интервала $[0, k)$ налазе даме које су постављене у првих k врста и задатак функције ће бити да испише све могуће распореде који проширују тај (додајући преостале даме у преостале врсте). Важна инваријанта ће бити да се даме које су постављене у тих првих k врста не нападају. Ако је $k = n$, тада су све даме већ постављене, на основу инваријанте знамо да се не нападају и можемо да испишемо то решење (оно је јединствено и не може се проширити). У супротном, разматрамо све опције за постављање даме на позицију k, тако да се даме у тако проширеном скупу не нападају. Пошто се зна да се даме на позицијама $[0, k)$ не нападају потребно је само проверити да ли се дама у врсти k напада са неком од дама постављених у првих k врста. Размотримо шта су кандидати за вредности на позицији k. Пошто не смемо имати два иста елемента низа тј. две исте врсте на којима се налазе даме, могли бисмо у делу низа на позицијама $[k, n)$ одржавати скуп елемената који су потенцијални кандидати за позицију k. Међутим, пошто за проверу дијагонала морамо упоредити даму k са свим претходно постављеним дамама, имплементацију можемо олакшати тако што на позицију k стављамо шири скуп могућих кандидата (скуп свих колона од 0 до $n - 1$) а онда за сваки од тих бројева проверавамо да ли се јавио у претходном делу низа чиме би се даме нападале по хоризонтали и да ли се постављањем даме у ту колону она по дијагонали нападала са неком претходно постављеном дамом. Ако се установи да то није случај, тј. да се постављањем даме у ту колону она не напада ни са једном од претходно постављених дама, онда је инваријанта задовољена и рекурзивно прелазимо на попуњавање наредних дама. Нема потребе за експлицитним поништавањем одлука које доносемо, јер ће се у свакој новој итерацији допуштена вредност уписати на позицију k, аутоматски поништавајући вредност коју смо ту раније уписали.

```
bool dameSeNapadaju(const vector<int>& permutacija) {
  for (int i = 0; i < permutacija.size(); i++)
    for (int j = i + 1; j < permutacija.size(); j++)
       if (abs(i - j) == abs(permutacija[i] - permutacija[j]))
          return true;
  return false;
}

void obradi(const vector<int>& permutacija) {
  if (!dameSeNapadaju(permutacija))
```

```
    ispisi(permutacija);
}

void obradiSvePermutacije(vector<int>& permutacija, int k) {
  if (k == 1)
    obradi(permutacija);
  else {
    for (int i = 0; i < k; i++) {
      swap(permutacija[i], permutacija[k-1]);
      obradiSvePermutacije(permutacija, k-1);
      swap(permutacija[i], permutacija[k-1]);
    }
  }
}

void obradiSvePermutacije(int n) {
  vector<int> permutacija(n);
  for (int i = 1; i <= n; i++)
    permutacija[i-1] = i;
  obradiSvePermutacije(permutacija, n);
}
```

Можемо приметити да је дрво претраге апсолутно симетрично у односу на вертикалну средину табле. Стога је довољно претражити само једну његову половину, а решења из друге половине директно добити осносиметричним пресликавањем решења из прве половине табле. Дакле, можемо претпоставити да ћемо директно одређивати само оне распореде где се дама у првој врсти поставља само у неку од првих $\lceil \frac{n}{2} \rceil$ колона. Ако је непарна димензија табле, онда у првој врсти дама може бити у средишњој колони. Тада се у другој врсти даме могу постављати само у прву половину колона, а остала решења се могу добити симетричним пресликавањем.

Рецимо и да постоје и друге симетрије (на пример у односу на хоризонталну средину табле или у односу на дијагонале), на основу којих је додатно могуће смањити простор претраге.

```
// odredjivanje simetricne permutacije datoj permutaciji u odnosu na vertikalnu sredinu table
vector<int> simetricna(const vector<int>& kolone) {
  int n = kolone.size();
  vector<int> sim(n);
  for (int i = 0; i < n; i++)
    sim[i] = n - kolone[i] - 1;
  return sim;
}

// niz kolone sadrži kolone u kojima se nalaze dame u vrstama iz
// intervala [0, v) za koji se pretpostavlja da predstavlja raspored
// dama koje se ne napadaju
// procedura vraća sve moguće rasporede dama koje proširuju taj raspored
// i u kojima se dame ne napadaju
void nDama(vector<int>& kolone, int v, vector<vector<int>>& resenja) {
  // dimenzija table
  int n = kolone.size();

  // sve dame su postavljene
  if (v == n) {
    // dodajemo trenutni raspored skupu resenja
    resenja.push_back(kolone);
  } else {
    // postavljamo dame u kolone [0, maks)
    int maksK = n;
    // zahvaljujuci simetriji, u prvoj vrsti posmatramo samo prvu polovinu kolona
    if (v == 0)
```

```
      maksK = (n + 1) / 2;
    // ako je u prvoj vrsti dama na sredisnjem polju, tada u drugoj
    // vrsti mozemo posmatrati samo prvu polovinu kolona
    if (v == 1 && n % 2 == 1 && kolone[0] == n / 2)
      maksK = n / 2;
    // postavljamo redom dame na odabrane kolone
    for (int k = 0; k < maksK; k++) {
      // postavljamo damu u vrsti v u kolonu k
      kolone[v] = k;
      // proveravamo da li se dame i dalje ne napadaju
      if (!dameSeNapadaju(kolone, v))
        // ako se ne napadaju, nastavljamo sa proširivanjem tekućeg rasporeda
        nDama(kolone, v+1, resenja);
    }
  }
}

void nDama(int n) {
  // niz ispravnih rasporeda (gde je dama u prvoj vrsti samo u prvoj polovini kolona)
  vector<vector<int>> resenja;
  // niz koji za svaku kolonu beleži vrstu dame u toj koloni
  vector<int> kolone(n);
  // krećemo pretragu od prazne table
  nDama(kolone, 0, resenja);
  // ispisujemo pronadjena resenja
  for (int i = 0; i < resenja.size(); i++)
    ispisi(resenja[i]);
  // ispisujemo njima simetricna resenja
  for (int i = resenja.size() - 1; i >= 0; i--)
    ispisi(simetricna(resenja[i]));
}
```

Задатак: Судоку

Напиши програм који попуњава Судоку загонетку чији је циљ да се у матрицу димензије 9 пута 9 распореде бројеви од 1 до 9, тако да у свакој врсти, у свакој колони и у сваком од 9 квадрата димензије 3 пута 3 сви бројеви различити.

Улаз: Са стандардног улаза се учитава матрица димензије 9 пута 9 у којој су већ уписани неки бројеви, а на пољима која су празна уписана је нула.

Излаз: На стандардни излаз исписати решење загонетке (тест-примери ће бити такви да је решење сигурно јединствено).

Пример

Улаз	*Излаз*
749030680	749132685
006508000	326548179
000760324	518769324
800057060	892357461
407000508	437621598
050980002	651984732
184076000	184276953
000403800	275493816
063010247	963815247

Решење

Задатак решавамо бектрекингом тј. рекурзивно имплементираном претрагом у дубину. Рекрузивна функција уз матрицу која се попуњава добија и редни број поља које треба попунити (она враћа информацију о томе

да ли је матрицу било могуће потпуно попунити). Попуњавање креће од горњег левог угла и тече врсту по врсту, све док не попунимо целу матрицу. Координате поља се веома једноставно могу одредити на основу његовог редног броја. Ако је тренутно поље већ попуњено (јер су у старту нека поља већ попуњена), тада одмах прелазимо на наредно поље. У супротном проверавамо све могуће вредности за то поље. Након уписа вредности проверавамо да ли је тиме направљен неки конфликт тј. да ли се десило да је у истој врсти, у истој колони или у истом квадрату већ постојао број који је уписан. Ако јесте, претрагу прекидамо, а ако није, настављамо је даље, попуњавањем наредног поља. Чим неки од рекурзивних позива успе да попуни целу матрицу, претрага се прекида и наредни рекурзивни позиви се не врше.

```
const int n = 3;

bool konflikt(const vector<vector<int>>& A, int i, int j) {
  // da li se A[i][j] nalazi već u koloni j
  for (int k = 0; k < n * n; k++)
    if (k != i && A[i][j] == A[k][j])
      return true;

  // da li se A[i][j] nalazi već u vrsti i
  for (int k = 0; k < n * n; k++)
    if (k != j && A[i][j] == A[i][k])
      return true;

  // da li se A[i][j] već nalazi u kvadratu koji sadrži polje (i, j)
  int x = i / n, y = j / n;
  for (int k = x * n; k < (x + 1) * n; k++)
    for (int l = y * n; l < (y + 1) * n; l++)
      if ((k != i || l != j) && A[i][j] == A[k][l])
        return true;

  // ne postoji konflikt
  return false;
}

bool sudoku(vector<vector<int>>& A, int rbr) {
  int i = rbr / (n*n), j = rbr % (n*n);
  // ako je polje (i, j) već popunjeno
  if (A[i][j] != 0) {
    // ako je u pitanju poslednje polje, uspešno smo popunili ceo
    // sudoku
    if (rbr == n * n * n * n - 1)
      return true;
    // rekurzivno nastavljamo sa popunjavanjem
    return sudoku(A, rbr + 1);
  } else {
    // razmatramo sve moguće vrednosti koje možemo da upišemo na polje
    // (i, j)
    for (int k = 1; k <= n*n; k++) {
      // upisujeAo vrednost k
      A[i][j] = k;
      // ako time napravljen neki konflikt, nastavljamo popunjavanje
      // (pošto je polje popunjeno, na sledeće polje će se automatski
      // preći u rekurzivnom pozivu)
      // ako se sudoku uspešno popuni, prekidaAo dalju pretragu
      if (!konflikt(A, i, j))
        if (sudoku(A, rbr))
          return true;
    }
    // poništavamo vrednost upisanu na polje (i, j), jer se popunjena polja
    // smatraju fiksiranim (datim u postavci problema)
```

```
    A[i][j] = 0;
    // konstatujemo da ne postoji rešenje
    return false;
  }
}
```

Задатак: Број поднизова датог збира

Напиши програм који одређује колико поднизова (не обавезно узастопних елемената) датог низа позитивних бројева има збир једнак датом броју.

Улаз: Са стандардног улаза се учитава број $1 \leq n \leq 30$, а затим у наредном реду n позитивних реалних бројева (заокружених на две децимале), раздвојених размацима.

Излаз: На стандардни излаз исписати тражени број поднизова (два реална броја се могу сматрати једнакима ако се разликују за мање од 10^{-5}).

Пример

Улаз	*Излаз*
4 3.2 5.7 9.4 6.9 12.6	2

Решење

Груба сила

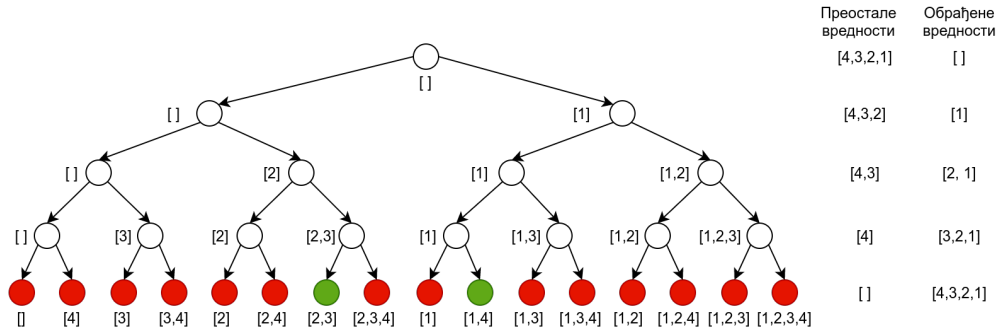
Једна могућност је да се задатак реши грубом силом, тј. да се наброје сви поднизови и да се за сваки од њих провери да ли му је збир једнак траженом. Набрајање поднизова се може постићи било помоћу функције која одређује наредни подниз у лексикографском редоследу, било помоћу рекурзивног набрајања свих могућности. Разни начини су приказани у задатку Сви подскупови. Једна могућност рекурзивне имплементације је заснована на томе да сваки непразни низ разложимо на његов префикс без последњег елемента и последњи елемент и да независно разматрамо могућности када последњи елемент јесте и када није укључен у подниз. То подразумева да чувамо тренутно одабрани подниз обрађеног дела низа на позицијама $[n, n_0)$, где је n_0 дужина целог низа, а n параметар који се смањује током рекурзије. Задатак рекурзивне функције је да на све могуће начине допуни тај подниз елементима са позиција $[0, n)$ из полазног низа и да врати број тако направљених поднизова који имају дати збир. У старту је $n = n_0$, а подниз је празан (то је заиста једини могући подниз дела низа на позицијама $[n_0, n_0)$).

- Ако је $n = 0$, цео низ је обрађен и подниз не може више да се допуњава (јер је скуп елемената на позицијама $[0, n) = [0, 0)$ празан). Израчунава се његов збир и ако је он једнак циљном збиру, тада функција враћа 1 (пронађен је један тражени низ који проширује тренутни подниз елементима празног скупа и има збир једнак циљаном), а у супротном враћа 0 (не постоји ни један низ који проширује тренутни подниз елементима празног скупа и има збир једнак циљаном).

- Ако је n позитивно, разматрамо посебно могућности да последњи елемент префикса $[0, n)$ тј. да елемент niz_{n-1} буде или да не буде укључен у подниз. У оба случаја рекурзивно позивамо функцију за скраћени префикс (тј. за вредност $n - 1$).

Да бисмо избегли реалокације, подниз можемо унапред алоцирати на дужину n_0, али тада параметар функције треба да буде и број елемената подниза p.

Дрво рекурзивних позива које настаје приликом тражења свих поднизова низа $[4, 3, 2, 1]$, чији је збир 5 је приказано на наредној слици (једноставности ради смо претпоставили да су бројеви у низу и циљни збир цели).



Слика 7.8: Груба сила – испитивање свих поднизова

```
const double EPS = 0.00001;

// funkcija izracunava broj nacina da se dati podniz duzine p
// prosiri elementima niza sa pozicija [0, n) tako da se dobije podniz
// ciji je zbir jednak datom ciljnom zbiru
int brojPodnizovaDatogZbira(const vector<double>& niz, int n,
                            double ciljniZbir,
                            vector<double>& podniz, int p) {
  // u nizu nema preostalih elemenata, pa je trenutni podniz jedini kandidat
  if (n == 0) {
    // racunamo zbir elemenata trenutnog podniza
    double zbirPodniza = 0.0;
    for (int i = 0; i < p; i++)
      zbirPodniza += podniz[i];
    // proveravamo da li je jednak ciljnom zbiru
    if (abs(zbirPodniza - ciljniZbir) < EPS)
      return 1;
    else
      return 0;
  } else {
    // broj podnizova bez ukljucenog poslednjeg elementa niza
    int broj = 0;
    broj += brojPodnizovaDatogZbira(niz, n-1, ciljniZbir, podniz, p);
    // broj podnizova sa ukljucenim poslednjim elementom niza
    podniz[p] = niz[n-1];
    broj += brojPodnizovaDatogZbira(niz, n-1, ciljniZbir, podniz, p+1);
    return broj;
  }
}

// funkcija racuna koliko podnizova datog niza ima zbir jednak ciljnom
int brojPodnizovaDatogZbira(vector<double>& niz, double ciljniZbir) {
  // broj elemenata niza
  int n = niz.size();
  vector<double> podniz(n);
  // rekurzivnom funkcijom racunamo trazeni broj podnizova
  return  brojPodnizovaDatogZbira(niz, n, ciljniZbir, podniz, 0);
}
```

Друга могућност имплементације претраге грубом силом је да као параметар рекурзивне функције прослеђујемо тренутни циљни збир тј. разлику између траженог збира и збира елемената тренутно укључених у подниз обрађеног дела низа. Сам тај подниз није неопходно одржавати. Другим речима, задатак рекурзивне функције је да врати колико поднизова тренутно необрађеног дела низа има збир једнак датом циљном збиру, при чему се тај циљни збир сада смањује кроз рекурзивне позиве (када год се укључи неки нови елемент).

Илустрације ради, рецимо да рекурзивна конструкција може бити таква да непразне низове разлаже на њихов

први елемент и суфикс низа иза тог првог елемента. То значи да ће тренутно обрађени део низа бити на позицијама $[0, k)$, док ће необрађени део низа бити на позицијама $[k, n)$ где је k тренутни параметар рекурзије, а n дужина целог низа. Рекурзија почиње када је $k = 0$, а завршава се када је $k = n$.

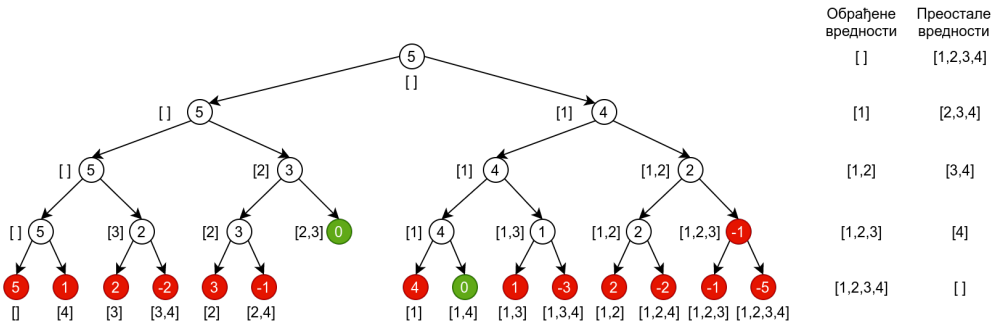
Ако је циљни збир једнак нули, то значи да је збир тренутног подниза елемената на позицијама $[0, k)$ једнак полазном траженом збиру и да смо нашли један задовољавајући подниз. Додатно, пошто су сви елементи низа позитивни, подниз се не може никако проширити додатним елементима тако да збир и даље остане једнак циљном, тако да није потребно даље настављати претрагу. Другим речима, једино празан низ елемената са позиција $[k, n)$ може имати збир 0, па функција може да врати резултат 1 (она враћа број низова).

Ако је циљни збир различит од нуле, настављамо као и раније.

Ако је $k = n$, тада у полазном низу нема необрађених елемената тј. преостали низ чије подскупове разматрамо је празан и он не може садржати подскуп чији ће циљни збир бити позитиван.

Ако је $k < n$, тада разматрамо могућност да се елемент niz_k укључи и могућност да се не укључи у подниз полазног низа. У првом случају умањујемо циљни збир за вредност тог елемента (то значи да тражимо број поднизова елемената са позиција $[k + 1, n)$ који дају тај умањени збир), а у другом циљни збир остаје непромењен.

На слици је приказано дрво рекурзивних позива када се одређује број поднизова низа $[1, 2, 3, 4]$ чији је збир једнак 5.



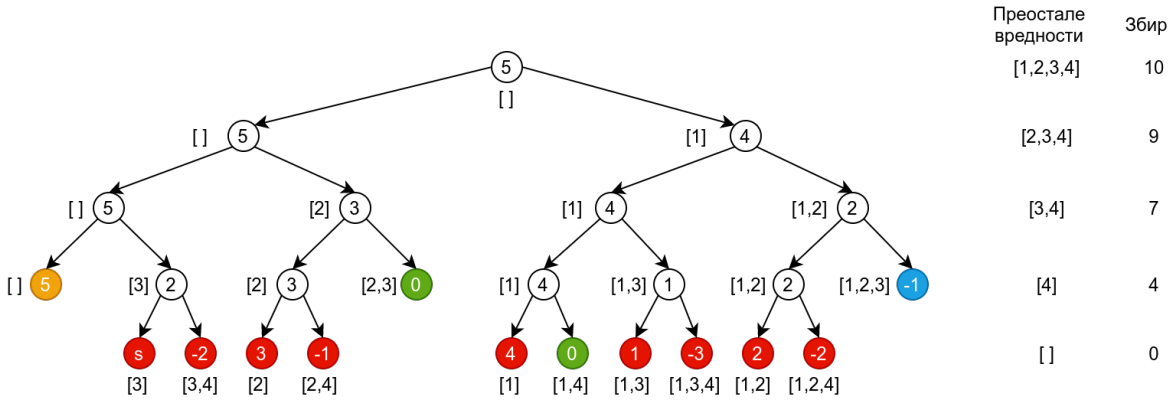
Слика 7.9: Груба сила – преостали циљни збир

```
const double EPS = 0.00001;

// funkcija odredjuje broj podnizova niza odredjenog elementima na
// pozicijama [k, n) takvih da je zbir elemenata podniza jednak ciljnom zbiru
int brojPodnizovaDatogZbira(const vector<double>& niz, double ciljniZbir, int k) {
  // jedino prazan niz ima zbir nula
  if (abs(ciljniZbir) < EPS)
    return 1;

  // jedini podniz praznog niza je prazan, a ciljni zbir je pozitivan
  if (k == niz.size())
    return 0;

  // posebno brojimo podnizove koji ukljucuju niz[k] i one koji ne ukljucuju
  // niz[k]
  return brojPodnizovaDatogZbira(niz, ciljniZbir - niz[k], k+1) +
         brojPodnizovaDatogZbira(niz, ciljniZbir, k+1);
}

int brojPodnizovaDatogZbira(const vector<double>& niz, double ciljniZbir) {
  // brojimo podnizove niza odredjenog elementima na pozicijama [0, n)
  return  brojPodnizovaDatogZbira(niz, ciljniZbir, 0);
}
```

Одсецања

Ефикаснија решења од решења грубом силом се могу добити применом различитих одсецања. Једно важно одсецање се може спровести на основу познавања интервала у коме могу лежати збирови свих поднизова преосталих елемената низа. Пошто су сви елементи позитивни, најмања могућа вредност збира подниза је нула (у случају празног низа), док је највећа могућа вредност збира подниза једнака збиру свих елемената низа. Дакле, ако је циљни збир строго мањи од нуле или строго већи од збира свих елемената преосталог дела низа, тада не постоји ни један подниз чији је збир једнак циљном и могуће је извршити одсецање претраге. Уместо да збир свих елемената преосталог дела низа (дела низа на позицијама $[k, n)$) рачунамо изнова у сваком рекурзивном позиву, можемо приметити да се у сваком наредном рекурзивном позиву низ само може смањити за један елемент, па се збир може рачунати инкрементално, умањивањем током рекурзије збира полазног низа за елементе уклоњене из низа.

На слици је приказано дрво рекурзивних позива са овим одсецањима када се у низу $[1, 2, 3, 4]$ траже поднизови чији је збир једнак 5. Приметимо да је једно одсецање извршено када је циљни збир био једнак 5 и када је у необрађеном делу низа остала само вредност 4, јер је збир свих преосталих вредности био мањи од циљног збира, а да је друго одсецање настало јер је након укључивања вредности $[1, 2, 3]$ збир већ претекао вредност 5 (добијен је чвор чија је нова циљна вредност -1).

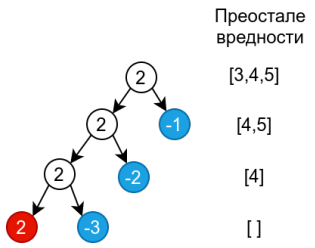


Слика 7.10: Претрага уз основно одсецање

```
const double EPS = 0.00001;

// racuna se broj podnizova elemenata niza na pozicijama [k, n) koji
// imaju dati zbir, pri cemu se zna da je zbir tih elemenata jednak
// zbirPreostalih
int brojPodnizovaDatogZbira(const vector<double>& niz, double ciljniZbir,
                            double zbirPreostalih, int k) {
  // ciljni zbir 0 se dobija samo ako se ne uzme ni jedan element
  if (abs(ciljniZbir) < EPS)
    return 1;

  // jedini podniz praznog niza je prazan, a ciljni zbir je pozitivan
  if (k == niz.size())
    return 0;

  // posto su svi brojevi pozitivni, nije moguce dobiti negativan ciljni zbir
  if (ciljniZbir + EPS < 0)
    return 0;

  // cak ni uzimanje svih elemenata ne moze dovesti do ciljnog zbira,
  // pa nema podnizova koji bi dali ciljni zbir
  if (zbirPreostalih + EPS < ciljniZbir)
    return 0;

  // broj podnizova u kojima ucestvuje element a[k]
```

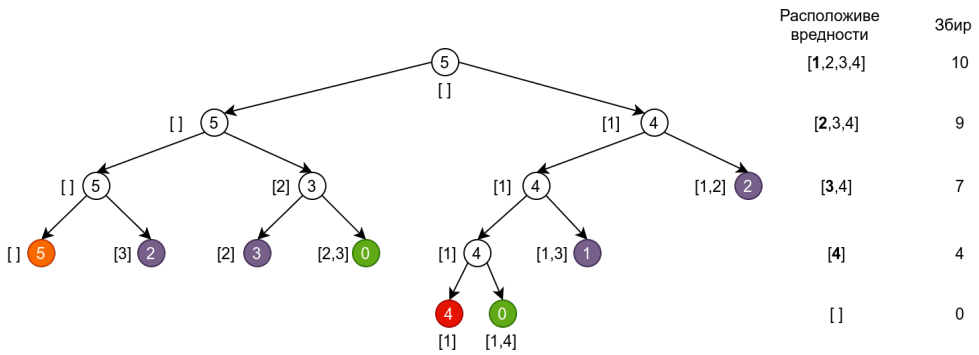
```
  return brojPodnizovaDatogZbira(niz, ciljniZbir - niz[k],
                                 zbirPreostalih - niz[k], k+1) +
    // broj podnizova u kojima ne ucestvuje element a[k]
    brojPodnizovaDatogZbira(niz, ciljniZbir,
                            zbirPreostalih - niz[k], k+1);
}

// funkcija racuna koliko podnizova datog niza ima zbir jednak ciljnom
int brojPodnizovaDatogZbira(vector<double>& niz, double ciljniZbir) {
  // broj elemenata niza
  int n = niz.size();
  // izracunavamo zbir elemenata niza
  double zbirNiza = 0;
  for (int i = 0; i < n; i++)
    zbirNiza += niz[i];
  // rekurzivnom funkcijom racunamo trazeni broj podnizova
  return  brojPodnizovaDatogZbira(niz, ciljniZbir, zbirNiza, 0);
}
```

Још једно могуће одсецање се може извршити када се установи да је најмањи од преосталих бројева у низу већи од циљног збира. Ако је тај циљни збир позитиван, тада није могуће достићи га (јер празан подниз има збир нула, а било који непразан подниз има збир већи или једнак од тог минималног елемента). Минимални елемент преосталог дела низа је једноставно одредити ако се низ сортира (што можемо урадити пре почетка претраге). Нагласимо да је ова одсецање само мала оптимизација одсецања чворова који имају негативан циљни збир. На primer, ако бисмо имали циљни збир 2 и преостале елементе 3, 4, 5 и 6, помоћу ове оптимизације бисмо одмах могли прекинути претрагу, док би се без ње добило дрво претрате приказано на наредној слици. Дакле, дрвета која се одсецају овом оптимизацијом, а не би била одсечена без ње, су само линеарне (а не експоненцијалне) величине у односу на број преосталих елемената низа.



Слика 7.11: Допринос одсецања на основу вредности најмањег елемента

На слици је приказано дрво рекурзивних позива са овим одсецањима када се у низу $[1, 2, 3, 4]$ траже поднизови чији је збир једнак 5. Када су одабрани елементи $[1, 2]$, тада је циљни збир 2, па пошто је најмањи преостали елемент 3, може да се изврши одсецање. Слично се догађа и када су одабрани елементи $[3]$ (циљни збир је 2), $[2]$ (циљни збир је 3) и $[1, 3]$ (циљни збир је 1), и када је најмањи (заправо једини) преостали елемент једнак 4. Одсецање наступа и када није изабран ниједан елемент (циљни збир је 5), а једини преостали елемент је 4 (тада је циљни збир већи од збира свих преосталих елемената).



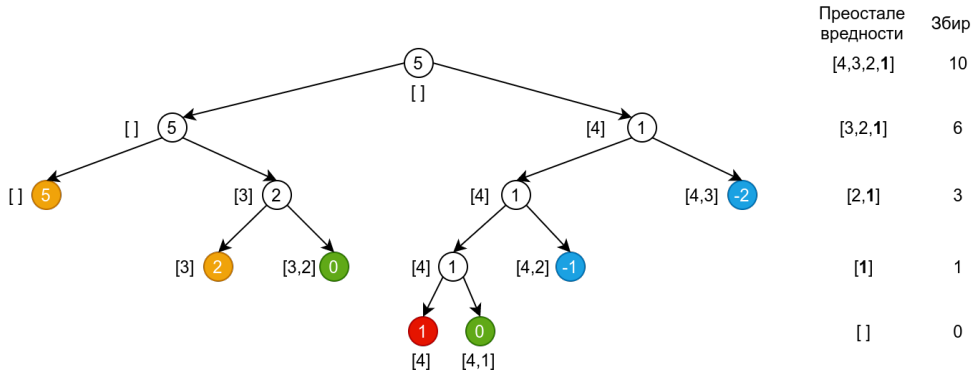
Слика 7.12: Претрага са одсецањем

```
const double EPS = 0.00001;

// racuna se broj podnizova elemenata niza na pozicijama [k, n) koji
// imaju dati zbir, pri cemu se zna da je zbir tih elemenata jednak
// zbirPreostalih
int brojPodnizovaDatogZbira(const vector<double>& niz, double ciljniZbir,
                            double zbirPreostalih, int k) {
  // ciljni zbir 0 se dobija samo ako se ne uzme ni jedan element
  if (abs(ciljniZbir) < EPS)
    return 1;

  // jedini podniz praznog niza je prazan, a ciljni zbir je pozitivan
  if (k == niz.size())
    return 0;

  // cak ni uzimanje svih elemenata ne moze dovesti do ciljnog zbira,
  // pa nema podnizova koji bi dali ciljni zbir
  if (zbirPreostalih + EPS < ciljniZbir)
    return 0;

  // vec uzimanje najmanjeg elementa prevazilazi ciljni zbir, pa
  // nema podnizova koji bi dali ciljni zbir
  if (niz[k] > ciljniZbir + EPS)
    return 0;

        // broj podnizova u kojima ucestvuje element a[k]
  return brojPodnizovaDatogZbira(niz, ciljniZbir - niz[k],
                                 zbirPreostalih - niz[k], k+1) +
        // broj podnizova u kojima ne ucestvuje element a[k]
        brojPodnizovaDatogZbira(niz, ciljniZbir,
                                 zbirPreostalih - niz[k], k+1);
}

// funkcija racuna koliko podnizova datog niza ima zbir jednak ciljnom
int brojPodnizovaDatogZbira(vector<double>& niz, double ciljniZbir) {
  // broj elemenata niza
  int n = niz.size();
  // sortiramo elemente niza neopadajuce
  sort(begin(niz), end(niz));
  // izracunavamo zbir elemenata niza
  double zbirNiza = 0;
  for (int i = 0; i < n; i++)
    zbirNiza += niz[i];
  // rekurzivnom funkcijom racunamo trazeni broj podnizova
```

```
  return  brojPodnizovaDatogZbira(niz, ciljniZbir, zbirNiza, 0);
}
```

Када се врше одсецања, редослед обиласка вредности у дрвету претраге може значајно да утиче на величину дрвета. На пример, уместо да се прво разматра укључивање најмањег елемента низа у подниз, може се прво разматрати укључивање највећег елемента уз подниз (уз иста одсецања која су раније описана). На почетку је низ потребно сортирати нерастуће (уместо неопадајуће), а најмањи елемент необрађеног дела низа се увек налази на крају самог низа.

На слици је приказано дрво рекурзивних позива са овим одсецањима када се у низу $[1, 2, 3, 4]$ траже поднизови чији је збир једнак 5. Приметимо да се у овом примеру већина одсецања врши већ на основу тога да ли преостали циљни збир припада интервалу од 0 до збира свих преосталих елемената, тако да ова стратегија гранања отвара много више могућности за та основна одсецања (у овом малом примеру се чак ниједном није десило да је наступило одсецање на основу вредности минималног елемента у преосталом делу низа).



Слика 7.13: Претрага са одсецањем

```
const double EPS = 0.00001;

// racuna se broj podnizova elemenata niza na pozicijama [k, n) koji
// imaju dati zbir, pri cemu se zna da je zbir tih elemenata jednak
// zbirPreostalih
int brojPodnizovaDatogZbira(const vector<double>& niz, double ciljniZbir,
                            double zbirPreostalih, int k) {
  // ciljni zbir 0 se dobija samo ako se ne uzme ni jedan element
  if (abs(ciljniZbir) < EPS)
    return 1;

  // jedini podniz praznog niza je prazan, a ciljni zbir je pozitivan
  if (k == niz.size())
    return 0;

  // cak ni uzimanje svih elemenata ne moze dovesti do ciljnog zbira,
  // pa nema podnizova koji bi dali ciljni zbir
  if (zbirPreostalih + EPS < ciljniZbir)
    return 0;

  // vec uzimanje najmanjeg elementa prevazilazi ciljni zbir, pa
  // nema podnizova koji bi dali ciljni zbir
  if (niz[niz.size()-1] > ciljniZbir + EPS)
    return 0;

        // broj podnizova u kojima ucestvuje element a[k]
  return brojPodnizovaDatogZbira(niz, ciljniZbir - niz[k],
                                 zbirPreostalih - niz[k], k+1) +
        // broj podnizova u kojima ne ucestvuje element a[k]
        brojPodnizovaDatogZbira(niz, ciljniZbir,
```

```
                                zbirPreostalih - niz[k], k+1);
}

// funkcija racuna koliko podnizova datog niza ima zbir jednak ciljnom
int brojPodnizovaDatogZbira(vector<double>& niz, double ciljniZbir) {
  // broj elemenata niza
  int n = niz.size();
  // sortiramo elemente niza nerastuce
  sort(begin(niz), end(niz), greater<int>());
  // izracunavamo zbir elemenata niza
  double zbirNiza = 0;
  for (int i = 0; i < n; i++)
    zbirNiza += niz[i];
  // rekurzivnom funkcijom racunamo trazeni broj podnizova
  return  brojPodnizovaDatogZbira(niz, ciljniZbir, zbirNiza, 0);
}
```

Задатак: Мерење са n тегова

Дато је n тегова, за сваки тег позната је његова маса. Датим теговима треба измерити масу S тако да се укупна маса изабраних тегова најмање разликује од S. Написати програм којим се одређује минимална разлика при таквом мерењу.

Улаз: Прва линија стандардног улаза садржи природан број n ($n \leq 10$). Следећих n линија садрже реалне бројеве, сваки у посебној линији, који представљају масе датих тегова. Последња линија стандардног улаза садржи реалан број S који представља масу коју меримо.

Излаз: На стандардном излазу приказати у једној линији минималну разлику добијену при мерењу, разлику приказати на две децимале.

Пример

Улаз	*Излаз*
5 2.3 1.0 0.5 2.0 0.25 4.0	0.05

Објашњење

Најбољи резултат се добије када се укључе тегови чије су масе $2,3$, затим $1,0$, затим $0,5$ и $0,25$. Укупна измерена маса је тада $4,05$, па је одступање $0,05$.

Решење

Груба сила - провера свих подскупова

Решење грубом силом подразумева да се испитају сви могући подскупови датог скупа тегова. Генерисање свих подскупова се разматра у задатку Сви подскупови.

На пример, набрајање можемо остварити помоћу функције која проналази лексикографски следећи подскуп. За сваки генерисани подскуп израчунавамо масу тегова, рачунамо одступање од жељене масе и ако је оно мање од тренутно најмањег одступања, ажурирамо минимум. Најмање одступање је могуће иницијализовати на жељену масу, јер масу 0 увек можемо добити помоћу празног скупа тегова. Рецимо да бисмо заједно са подскупом могли одржавати и масу предмета у подскупу и приликом проналажења наредног подскупа ажурирати ту масу, чиме би се програм мало убрзао.

```
// funkcija pronalazi sledeci podskup (leksikografski sledecu
// varijaciju elemenata false i true)
bool sledeciPodskup(vector<bool>& uSkupu) {
```

```
  int i;
  for (i = uSkupu.size() - 1; i >= 0 && uSkupu[i]; i--)
    uSkupu[i] = false;

  if (i < 0)
    return false;

  uSkupu[i] = true;
  return true;
}

double merenje(const vector<double>& tegovi, double ciljnaMasa) {
  // broj tegova
  int n = tegovi.size();
  // krecemo od praznog skupa tegova
  vector<bool> uSkupu(n, false);
  double minRazlika = ciljnaMasa;
  do {
    // izracunavamo masu tegova u tekucem skupu
    double tekucaMasa = 0;
    for(int i = 0; i < n; i++)
      if (uSkupu[i])
        tekucaMasa += tegovi[i];
    // azuriramo minimalnu razliku, ako je to potrebno
    double tekucaRazlika = abs(ciljnaMasa - tekucaMasa);
    if (tekucaRazlika < minRazlika)
      minRazlika = tekucaRazlika;
  } while (sledeciPodskup(uSkupu));
  return minRazlika;
}
```

Рекурзивно набрајање подскупова

Рекурзивну функцију која генерише све подскупове можемо модификовати тако да враћа вредност најмањег одступања од циљне тежине. Подсетимо се, функција прима текући подскуп елемената низа са позиција из интервала $[0, k)$ и на све начине га проширује елементима низа из интервала $[k, n)$. Када је $k = n$, генерисан је подскуп целог низа, израчунава се његово одступање и враћа се резултат (то је једино, па уједно и најмање одступање). У супротном се разматрају две могућности: елемент на позицији k се или додаје у подскуп или се из подскупа изоставља. Вршимо два рекурзивна позива и мање од њихова два одступања нам даје тражено минимално одступање (то је најмање одступање за све подскупове који садрже елементе са позиција $[0, k)$ који су тренутно одабрани). Иницијални позив се врши за $k = 0$ (у почетку ништа није одабрано и сви елементи тегови нам на располагању). Приметимо да заправо није неопходно да знамо који су тачно тегови тренутно одабрани, већ само њихову укупну масу.

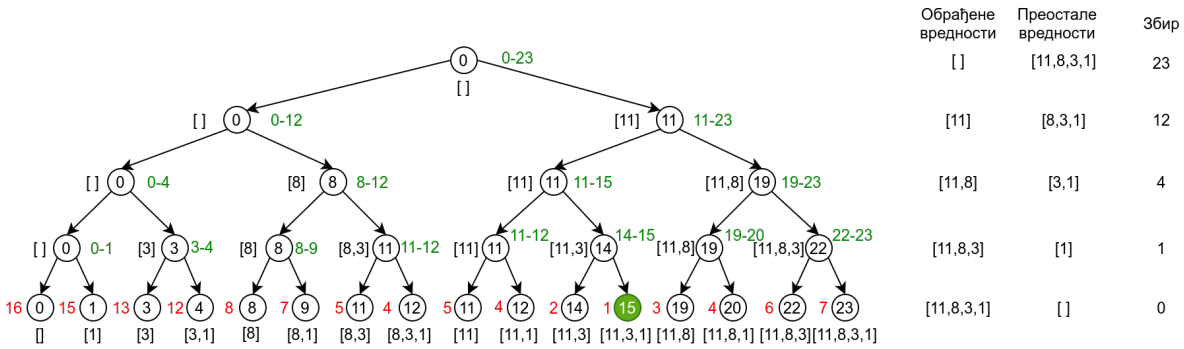
```
double merenje(const vector<double>& tegovi, double ciljnaMasa,
               int k, double tekucaMasa) {
  // nemamo vise tegova koje mozemo uzimati
  if (k == tegovi.size())
    // najmanja razlika je odredjena tekucom masom ukljucenih tegova
    return abs(ciljnaMasa - tekucaMasa);

  // gledamo bolju mogućnost od one kada preskacemo teg na poziciji k
  // i one kada uključimo teg na poziciji k
  return min(merenje(tegovi, ciljnaMasa, k+1, tekucaMasa),
             merenje(tegovi, ciljnaMasa, k+1, tekucaMasa + tegovi[k]));
}

double merenje(const vector<double>& tegovi, double ciljnaMasa) {
  // krecemo od pozicije 0 i praznog skupa ukljucenih tegova
```

```
  return merenje(tegovi, ciljnaMasa, 0, 0.0);
}

int main() {
  int n;
  cin >> n;
  vector<double> tegovi(n);
  for (int i = 0; i < n; i++)
    cin >> tegovi[i];
  double ciljnaMasa;
  cin >> ciljnaMasa;
  cout << fixed << setprecision(2) << showpoint
       << merenje(tegovi, ciljnaMasa) << endl;
  return 0;
}
```

Одсецање

Ефикасније решење можемо добити ако током исцрпне претраге применимо одсецање. У сваком рекурзивном позиву могуће је проценити интервал вредности ком припадају сва проширења текућег скупа одабраних тегова. Претпоставићемо да у сваком чвору претраге знамо укупну масу M_o свих одабраних тегова из већ обрађеног дела низа (то је део низа на позицијама $[0, k)$) и укупну масу M_p свих тегова у преосталом делу низа (то је део низа на позицијама $[k, n)$). Ако се не одабере ни један додатни тег постиже се маса M_o, а ако се одаберу сви додатни тегови, постиже се маса $M_o + M_p$, што значи да сва проширења тренутно одабраног скупа тегова припадају интервалу $[M_o, M_o + M_p]$.

На слици је приказано дрво исцрпне претраге ако су масе тегова $[11, 8, 3, 1]$, и ако је циљна маса 16. Поред сваког чвора приказан је низ одабраних тегова, а у сваком чвору је приказана и њихова укупна маса M_o. Поред чворова је приказан и интервал $[M_o, M_o + M_p]$, коме припадају масе могућих проширења тренутно одабраног скупа тегова. Поред сваког листа приказана је маса одабраних предмета и њено одступање од циљне масе.



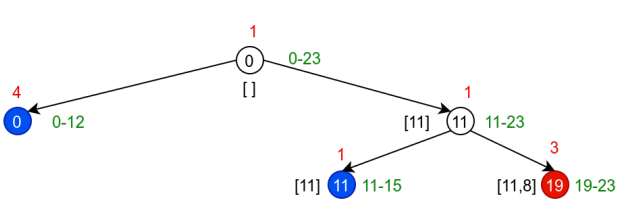
Слика 7.14: Исцрпна претрага

Ако је вредност $M_o + M_p$ мања или једнака од циљне масе коју треба измерити, онда је најбоље узети све тегове (изостављање било ког тела даће мању разлику), па не треба испробавати разне могућности. Слично, ако је вредност M_o већа или једнака од циљне масе, тада не треба узети ни један додатан тег (узимање било ког додатног тега даће већу разлику). Претрагу, тј. испробавање разних могућности вршимо само ако је циљна маса у интервалу $(M_o, M + _p)$.

Пошто нам је циљ да се ови интервали сузе, пожељно је на почетку сортирати све тегове у нерастући редослед (тиме преостале масе M_p на сваком наредном нивоу постају све мање и мање).

Након одсецања на основу наведена два правила, добија се веома једноставно дрво, приказано на наредној слици. Када је маса тренутних тегова 0, а маса преосталих тегова 12, јасно је да се најбољи резултат добија када се узму сви преостали тегови, па се тада не врши претрага, него се одмах враћа да је најмање одступање једнако $16 - 12 = 4$. Слично се догађа и када је маса одбраних тегова 11, а маса преосталих тегова 4 – тада се као резултат враћа $16 - 15 = 1$. Друга врста одсецања наступа када је маса тренутно одабраних тегова

једнака 19. Тада је најбоље не додавати тегове, па се као резултат враћа $19 - 16 = 3$. У остала два случаја се гранањем испробавају обе могућности и као резултат се враћа мање од два добијена одступања.



Обрађене вредности	Преостале вредности	Збир
[ ]	[11,8,3,1]	23
[1]	[8,3,1]	12
[1,3]	[3,1]	4
[1,3,8]	[1]	1
[1,3,8,11]	[ ]	0

Слика 7.15: Претрага са одсецањем

```
// funkcija odredjuje najmanju razliku izmedju ciljne mase koju treba izmeriti i
// mase koja se može postici pomoću tegova iz intervala [k, n) cija je ukupna
// masa jednaka broju preostalaMasa
double merenje(const vector<double>& tegovi, double ciljnaMasa,
               int k, double tekucaMasa, double preostalaMasa) {
  // nemamo vise tegova koje mozemo uzimati
  if (k == tegovi.size())
    // najmanja razlika je odredjena tekucom masom ukljucenih tegova
    return abs(ciljnaMasa - tekucaMasa);

  // ako je optimum da se uzmu svi tegovi
  if (tekucaMasa + preostalaMasa <= ciljnaMasa)
    return ciljnaMasa - (tekucaMasa + preostalaMasa);

  // ako je optimum da se ne uzme ni jedan teg
  if (tekucaMasa >= ciljnaMasa)
    return tekucaMasa - ciljnaMasa;

  // analiziramo mogućnost da je teg na poziciji k preskocen i da je uzet
  return min(merenje(tegovi, ciljnaMasa,
                     k+1, tekucaMasa, preostalaMasa - tegovi[k]),
             merenje(tegovi, ciljnaMasa,
                     k+1, tekucaMasa + tegovi[k], preostalaMasa - tegovi[k]));
}

double merenje(vector<double>& tegovi, double ciljnaMasa) {
  // sortiramo mase tegova u nerastucem redosledu
  sort(begin(tegovi), end(tegovi), greater<int>());
  // racunamo ukupnu masu svih tegova
  double ukupnaMasa = 0.0;
  for (int i = 0; i < tegovi.size(); i++)
    ukupnaMasa += tegovi[i];
  // krecemo od pozicije 0 i praznog skupa ukljucenih tegova
  return merenje(tegovi, ciljnaMasa, 0, 0.0, ukupnaMasa);
}
```

Рецимо и да имплементацију можемо направити мало другачије. Уместо функције која враћа најмању вредност одступања, можемо направити процедуру (функцију без повратне вредности) која ажурира глобалну променљиву. Ипак, резоновање о програму је много једноставније ако се не користе глобалне променљиве.

```
// maksimalni broj tegova
const int MAX_TEGOVA = 50;
```

```
// broj tegova
int n;
// mase tegova
double tegovi[MAX_TEGOVA];
// masa koju je potrebno izmeriti
double ciljnaMasa;
// najmanja razlika izmedju ciljne mase i mase nekog podskupa tegova
double minRazlika;
// masa svih odabranih tegova
double tekucaMasa;
// masa svih preostalih tegova
double preostalaMasa;

void merenje(int k) {
  // razmotrili smo svih n tegova
  if (k == n) {
    // azuriramo razliku ako je potrebno
    minRazlika = min(minRazlika, abs(ciljnaMasa - tekucaMasa));
    return;
  }

  // ako je optimum da se uzmu svi tegovi
  if (tekucaMasa + preostalaMasa <= ciljnaMasa) {
    minRazlika = min(minRazlika, ciljnaMasa - (tekucaMasa + preostalaMasa));
    return;
  }

  // ako je optimum da se ne uzme ni jedan teg
  if (tekucaMasa >= ciljnaMasa) {
    minRazlika = min(minRazlika, tekucaMasa - ciljnaMasa);
    return;
  }

  // analiziramo mogućnost da je teg na poziciji k preskocen i da je uzet
  preostalaMasa -= tegovi[k];
  merenje(k+1);
  tekucaMasa += tegovi[k];
  merenje(k+1);
  tekucaMasa -= tegovi[k];
  preostalaMasa += tegovi[k];
}

double merenje() {
  // sortiramo tegove po tezini, nerastuce
  sort(tegovi, tegovi+n, greater<int>());
  // krecemo od toga da smo izmerili 0.0
  tekucaMasa = 0.0;
  minRazlika = ciljnaMasa;
  // preostala masa je ukupna masa svih tegova
  double ukupnaMasa = 0.0;
  for (int i = 0; i < n; i++)
    ukupnaMasa += tegovi[i];
  preostalaMasa = ukupnaMasa;
  // zapocinjemo pretragu
  merenje(0);
  return minRazlika;

}
```

Задатак: Бојење графа са три боје

У једној земљи постоји неколико планинских врхова на којима ће се поставити предајници најсавременије мобилне мреже. Они могу да раде на једној од три различите радио-фреквенције. Сваки предајник може да преноси специјални сигнал другим предајницима који су близу њега, при чему два предајника који су близу један другоме не смеју да користе исту фреквенцију. Написати програм који одређује да ли је могуће доделити фреквенције свим предајницима тако да нема сударања.

Улаз: Са стандардног улаза се учитава број предајника n ($1 \leq n \leq 100$), а након тога број парова блиских предајника m ($n - 1 \leq m \leq \frac{n(n-1)}{2}$). Након тога, у наредних m редова се учитавају парови блиских предајника (сви предајници су обележени бројевима од 0 до $n - 1$). Систем је грађен тако да се сигнал сигурно може пренети од било ког до било ког другог предајника.

Излаз: На стандардни излаз исписати ознаке фреквенција (`1`, `2` и `3`) које су редом додељене предајницима или `-` ако фреквенције није могуће доделити тако да се блиски предајници не сударају. Ако је фреквенције могуће доделити на више начина, исписати онај који је најмањи у лексикографском редоследу.

Пример

Улаз	*Излаз*

`5`
`5`
`0 1`
`0 4`
`1 2`
`1 4`
`2 3`
`2 4`
`3 4`
	`1 2 1 2 3`

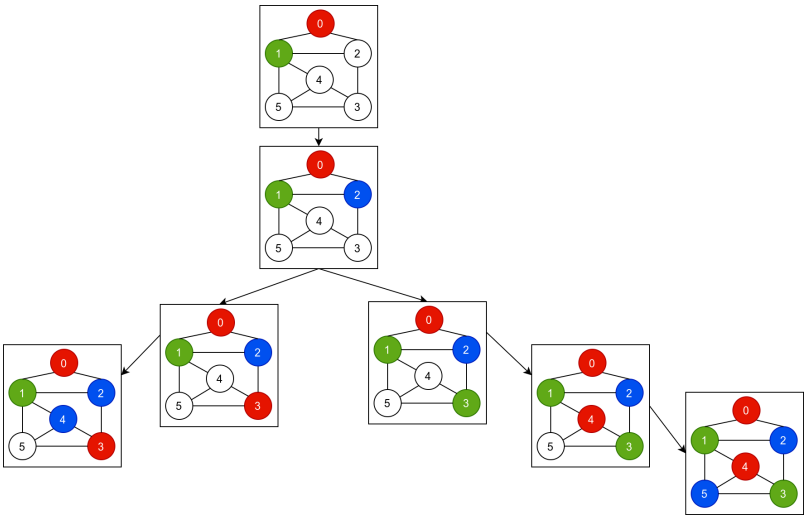
Решење

Овај проблем је у литератури познат као проблем 3-бојења графова (сваки предајник представља чвор графа, а свака фреквенција једну од 3 допуштене боје).

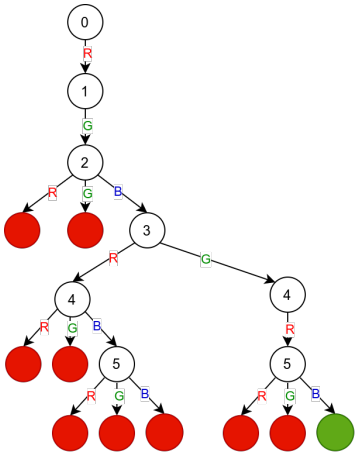
Задатак се једноставно (али не и ефикасно) решава бектрекинг претрагом.

Приметимо да је проблем симетричан и да ако се првом чвору може доделити нека боја, ознаке боја се могу променити тако да се том првом чвору додели било која друга боја. Стога можемо претпоставити да је први чвор обојен бојом 1 (јер се тражи најмањи распоред боја по лексикографском поретку). Ни један од његових суседа (а они сигурно постоје, јер је граф повезан) не може бити обојен бојом 1. Пошто се тражи најмањи распоред боја лексикографском поретку, његов сусед који има најмањи редни број треба да буде обојен бојом 2. За остале чворове покушавамо бојење разним бојама.

Пример. На сликама је приказан пример поступка бојења графа са три боје. Пре почетка претраге чвор 0 се боји у црвено, а чвор 1 (најмањи сусед чвора 0) у зелено. Тада чвор 2 мора да се обоји у плаво. Ако се чвор 3 обоји првом расположивом бојом (црвеном), тада чвор 4 мора да се обоји у плаво и чвор 5 не може да се обоји (јер је суседан и са црвеним и са зеленим и са плавим чвором). Тада се претрага враћа унатраг и чвор 3 боји у наредну расположиву боју – зелену. Чвор 4 се боји у прву расположиву боју – црвену, након чега чвор 5 мора да се обоји у плаву. У том тренутку је цео граф успешно обојен са три боје.



Слика 7.16: Пример бојења графа са 3 боје



Слика 7.17: Дрво претраге придруженом бојењу графа са 3 боје

```
// funkcija boji dati cvor, pri cemu su zadati
// susedi svih cvorova i boje do sada obojenih cvorova
bool oboj(const vector<vector<int>>& susedi, int cvor, vector<int>& boje) {
  // ako su svi cvorovi obojeni, bojenje sa 3 boje je uspesno
  int brojCvorova = susedi.size();
  if (cvor >= brojCvorova)
    return true;

  // ako je cvor vec obojen, preskacemo ga
  if (boje[cvor] != 0)
    return oboj(susedi, cvor + 1, boje);

  // pokusavamo da cvoru dodelimo svaku od 3 raspolozive boje
  for (int boja = 1; boja <= 3; boja++) {
    // linearnom pretragom proveravamo da li je moguce obojiti cvor
    // u tekucu boju
    bool mozeBoja = true;
    // proveravamo sve susede
    for (int sused : susedi[cvor])
      // ako je neki od njih vec obojen u tekucu boju
      if (boje[sused] == boja)
```

```
        // bojenje nije moguce
        mozeBoja = false;
    if (mozeBoja) {
      // bojimo tekuci cvor
      boje[cvor] = boja;
      // pokusavamo rekurzivno bojenje narednog cvora i ako uspemo
      // tada je bojenje moguce
      if (oboj(susedi, cvor+1, boje))
        return true;
    }
  }

  // probali smo sve tri boje i ni jedno bojenje nije moguce
  return false;
}

bool oboj(const vector<vector<int>>& susedi, vector<int>& boje) {
  // broj cvorova grafa
  // prva cvor 0 i njegov prvi sused se boje u boje 1 i 2
  boje[0] = 1; boje[*min_element(begin(susedi[0]), end(susedi[0]))] = 2;
  // krecemo bojenje od cvora 0
  return oboj(susedi, 0, boje);
}
```

Задатак: К бојење

Потребно је распоредити промењиве у регистре процесора. При том, је познато да неке промењиве не смеју да буду смештене у исти регистар (јер се користе истовремено). Написати програм који одређује најмањи број регистара потребан да се сместе све промењиве. На пример, у коду

```
x = a + b;
y = x * b;
return y;
```

Промењиве `a` и `b` као и промењиве `x` и `b` не смеју бити смештене у исти регистар. Могуће је употребити само два регистра. Прво се у регистар 1 смешта промењива `a`, а у регистар 2 промењива `b`. Након тога се вредност промењиве `x` смешта у регистар 1 (јер вредност промењиве `a` није надаље потребна). Након тога се вредност промењиве `y` може сместити било у регистар 1, било у регистар 2, јер надаље нису потребни ни вредност промењиве `x` ни вредност промењиве `b`. Једноставности ради претпоставити да је граф сачињен од промењивих и од ограничења између њих повезан.

Улаз: Са стандардног улаза се уноси број промењивих n ($1 \leq n \leq 50$). Након тога се до краја улаза уносе парови промењивих које не смеју да се сместе у исте регистре (промењиве се броје од 0 до $n-1$).

Излаз: На стандардни излаз исписати најмањи број регистара потребних да се све промењиве сместе.

Пример

Улаз	*Излаз*
6 0 1 0 2 1 2 1 4 1 5 2 3 3 4 3 5 4 5	3

Решење

Овај проблем се своди на одређивање најмањег броја боја потребног да се обоје чворови графа тако да никоја два суседна чвора нису обојена у исту боју. Наиме, променљиве можемо представити чворовима графа тако да гране поставимо између променљивих које не смеју бити обојене истом бојом, док регистре можемо представити бојама чворова графа.

Задатак је могуће решити коришћењем функције која проверава да ли се бојење може извршити помоћу k боја и применом бинарне претраге за одређивање преломне тачке, тј. најмање вредности k такве да се граф може обојити са k боја. Провера да ли се граф може обојити са k боја може се извршити бектрекинг претрагом. У задатку Бојење графа са три боје приказано је како се проверава да ли се граф може обојити са 3 боје и провера да ли се граф може обојити са k боја представља веома једноставну модификацију тог поступка.

```
// funkcija proverava da li se bojenje grafa moze nastaviti od datog
// cvora, pri cemu je dopusteno koristiti samo dati broj boja
bool mozeSeObojiti(const vector<vector<int>>& susedi,
                   int cvor, vector<int>& boje, int brojBoja) {
  // ako su svi cvorovi obojeni, bojenje je uspesno
  int brojCvorova = susedi.size();
  if (cvor >= brojCvorova)
    return true;

  // ako je cvor vec obojen, preskacemo ga
  if (boje[cvor] != 0)
    return mozeSeObojiti(susedi, cvor + 1, boje, brojBoja);

  // pokusavamo da cvoru dodelimo svaku od raspolozivih boja
  for (int boja = 1; boja <= brojBoja; boja++) {
    // linearnom pretragom proveravamo da li je moguce obojiti cvor
    // u tekucu boju
    bool mozeBoja = true;
    // proveravamo sve susede
    for (int sused : susedi[cvor])
      // ako je neki od njih vec obojen u tekucu boju
      if (boje[sused] == boja)
        // bojenje nije moguce
        mozeBoja = false;
    if (mozeBoja) {
      // bojimo tekuci cvor
      boje[cvor] = boja;
      // pokusavamo rekurzivno bojenje narednog cvora i ako uspemo
      // tada je bojenje moguce
      if (mozeSeObojiti(susedi, cvor+1, boje, brojBoja))
        return true;
    }
  }

  boje[cvor] = 0;
  return false;
}

// provera da li se dati graf moze obojiti sa datim brojem boja
bool mozeSeObojiti(const vector<vector<int>>& susedi, int brojBoja) {
  // broj cvorova grafa
  int n = susedi.size();
  // niz boja
  vector<int> boje(n, 0);
  // prva cvor 0 i njegov prvi sused se boje u boje 1 i 2
  boje[0] = 1; boje[susedi[0][0]] = 2;
  return mozeSeObojiti(susedi, 0, boje, brojBoja);
```

```
}

// najmanji broj boja kojima se mogu obojiti cvorovi grafa, tako
// da nikoja dva cvora nisu susedna
int minBrojBoja(const vector<vector<int>>& susedi) {
  // broj cvorova grafa
  int n = susedi.size();
  // binarnom pretragom nalazimo minimalan broj boja
  // sa strogo manje od l graf se ne moze obojiti, a sa strogo vise od d boja moze
  int l = 0, d = n-1;
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (mozeSeObojiti(susedi, s))
      d = s - 1;
    else
      l = s + 1;
  }
  return d+1;
}
```

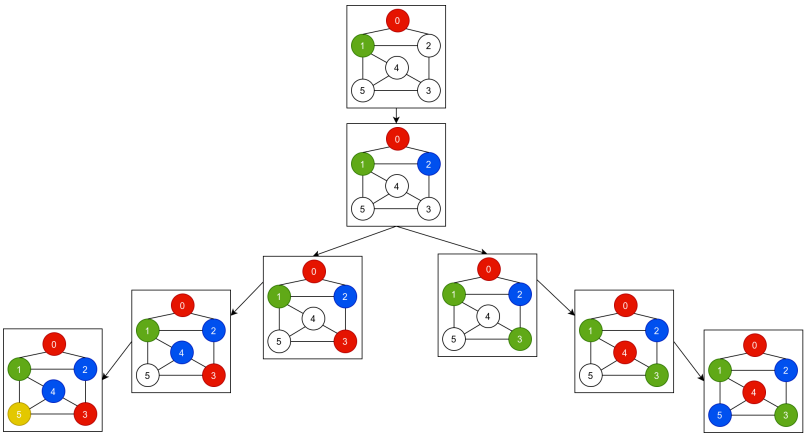
Задатак се може решити и једином бектрекинг претрагом. За разлику од провере да ли се граф може обојити са k боја, која се може зауставити чим се наиђе на прво успешно бојење, у овом алгоритму је потребно обићи цело стабло претраге. Када се пронађе неко успешно бојење са одређеним бројем боја (рецимо k), приликом повратка у претрази врши се ограничавање боја чворова на вредности од 1 до $k-1$ (јер нам је циљ да покушамо да пронађемо бојење са мањим бројем боја од k). Почетно огранчење броја боја је n, јер смо сигурни да се граф са n чворова може обојити помоћу n боја.

Једноставности ради тренутну вредност оптимума чувамо у глобалној променљивој.

Напомена. Слична идеја се може употребити у разним оптимизационим проблемима када се користи техника претраге са одсецањем. Наиме, вредности решења пронађених у једном делу дрвета претраге се могу употребити за одсецање у другом делу дрвета претраге (јер не желимо да разматрамо решења која су лошија од тренутно пронађеног најбољег решења). Ова техника се понекад назива *гранање са одсецањем* (енгл. branch and bound).

Пример.

На слици је приказано бојење графа са најмањим бројем боја. Претпоставимо да су боје обележене бројевима од 1 до n. Без губитка на општости се може претпоставити да се чвор нула може обојити у боју број 1 (црвену), а његов први сусед, чвор 1, у боју број 2 (зелену). За чвор 2 су нам иницијално на располагању боје од 1 до 6 (јер граф има 6 чворова). Бојење у боје 1 и 2 није могуће, па се чвор 2 боји у боју број 3 (плаву). За чвор 3 и даље су могуће све боје од 1 до 6, па се он боји у прву расположиву боју тј. боју 1 (црвену). За чвор 4 су расположиве боје од 1 до 6, па са и он боји у прву расположиву боју тј. боју 3 (плаву). На крају, и за чвор 5 су расположиве боје од 1 до 6 и он се боји у прву расположиву боју тј. боју 4. У том тренутку смо пронашли бојење са 4 боје и пошто у наставку претраге тражимо само боља решења, расположиве ће бити само прве три боје (1, 2 и 3). Враћамо се на чвор 4 и његову боју не можемо да променимо (јер већ има највећу расположиву боју). Зато се враћамо на чвор 3. Његову боју можемо да променимо у боју 2 (зелену). Тада чвор 4 бојимо у прву расположиву боју 1 (црвену), а чвор 5 у боју 3 (плаву), јер боје 1 и 2 нису могуће, а на располагању су нам боје од 1 до 3. У том тренутку је пронађено успешно бојење са 3 боје и разматрамо само бојења са 2 боје (кандидати за боје су сада само 1 и 2). Пошто су на путу до чвора 4 већ употребљене три боје, враћамо се уназад, без разматрања даљих могућности промене боје конкретног чвора све до чвора 2. Пошто чвор 2 не може да промени боју (јер је већ обојен у боју 3, а на располагању су нам сада само боје 1 и 2), враћамо се на чворове 1 и 0 који имају фиксиране боје и претрага се завршава.



Слика 7.18: Пример бојења графа са минималним бројем боја

```
// poznato je da se graf moze obojiti sa ovim brojem boja
// jednostavnosti radi koristimo globalnu promenljivu (mada to nije neohpodno)
int brojBoja;

// funkcija pokusava da prosiri bojenje graf dato nizom boje, u kome
// je trenutno upotrebljen broj boja dat promenljivom
// upotrebljenoBoja, tako sto boji dati cvor, koristeci raspolozive
// boje, koje su odredjene globalnom promenljivom brojBoja
void oboj(const vector<vector<int>>& susedi,
          int cvor, vector<int>& boje, int upotrebljenoBoja) {
  // ako su svi cvorovi obojeni, bojenje je uspesno
  int brojCvorova = susedi.size();
  if (cvor >= brojCvorova) {
    brojBoja = upotrebljenoBoja;
    return;
  }

  // ako je cvor vec obojen, preskacemo ga
  if (boje[cvor] != 0) {
    oboj(susedi, cvor + 1, boje, upotrebljenoBoja);
    return;
  }

  // pokusavamo da cvoru dodelimo svaku od raspolozivih boja
  for (int boja = 1; boja < brojBoja; boja++) {
    // linearnom pretragom proveravamo da li je moguce obojiti cvor
    // u tekucu boju
    bool mozeBoja = true;
    // proveravamo sve susede
    for (int sused : susedi[cvor])
      // ako je neki od njih vec obojen u tekucu boju
      if (boje[sused] == boja)
        // bojenje nije moguce
        mozeBoja = false;
    if (mozeBoja) {
      // bojimo tekuci cvor
      boje[cvor] = boja;
      // pokusavamo rekurzivno bojenje narednog cvora i ako uspemo
      // tada je bojenje moguce
      oboj(susedi, cvor+1, boje, max(upotrebljenoBoja, boja));
      boje[cvor] = 0;
```

```
      // ako smo vec upotrebili previse boja, mozemo odmah preseci pretragu
      if (upotrebljenoBoja >= brojBoja)
        return;
    }
  }
}

// najmanji broj boja kojima se mogu obojiti cvorovi grafa, tako
// da nikoja dva cvora nisu susedna
int minBrojBoja(const vector<vector<int>>& susedi) {
  // broj cvorova grafa
  int n = susedi.size();
  // boja svakog cvora
  vector<int> boje(n, 0);
  // graf se sigurno moze obojiti sa n boja
  brojBoja = n;
  // prva cvor 0 i njegov prvi sused se boje u boje 0 i 1
  boje[0] = 1; boje[susedi[0][0]] = 2;
  // krecemo bojenje od cvora 0
  oboj(susedi, 0, boje, 2);
  return brojBoja;
}
```

Глава 8

Динамичко програмирање

8.1 Појам и облици динамичког програмирања

У многим случајевима се дешава да током извршавања рекурзивне функције долази до *преклапања рекурзивних позива* (енгл. overlapping recursive calls) тј. да се идентични рекурзивни позиви (рекурзивни позиви са идентичним параметрима) извршавају више пута. Ако се то дешава често, програми су по правилу веома неефикасни (у многим случајевима број рекурзивних позива, па самим тим и сложеност бива експоненцијална у односу на величину улаза). До ефикаснијег решења се често може доћи техником **динамичког програмирања**. Оно често временску ефикасност поправља ангажовањем додатне меморије у којој се бележе резултати извршених рекурзивних позива. Динамичко програмирање долази у два облика.

- Техника **мемоизације** или **динамичког програмирања наниже** задржава рекурзивну дефиницију али у додатној структури података (најчешће низу или матрици, ређе мапи тј. речнику) бележи све резултате рекурзивних позива, да бих их у наредним позивима у којима су параметри исти само очитала из те структуре.
- Техника **динамичког програмирања навише** у потпуности уклања рекурзију и ту помоћну структуру података попуњава исцрпно у неком систематичном редоследу. Дакле, рекурзивна конструкција се замењује индуктивном, тј. итеративном.

Док се код мемоизације може десити да се рекурзивна функција не позива за неке вредности параметара, код динамичког програмирања навише се израчунавају вредности функције за све могуће вредности њених параметара мањих од вредности која се заправо тражи у задатку. Иако се на основу овога може помислити да је мемоизација ефикаснија техника, у пракси је чешћи случај да је током одмотавања рекурзије потребно израчунати вредност рекурзивне функције за баш велики број различитих параметара, тако да се ове предности мемоизаиције у пракси ретко сређе.

Најбољи начин да разјаснимо технику динамичког програмирања је да је илуструјемо на низу погодно одабраних примера. Кренућемо од Фибоначијевог низа, који је опште познати проблем и кроз чије се решавање могу илустровати већина основних концепата динамичког програмирања.

Задатак: Пчеле и трутови

Пчела матица носи јајашца. Ако трут оплоди јајашце пчеле, тада се из њега рађа женска пчела. Ако се јајашце не оплоди, онда се из њега излеже трут. Дакле, женска пчела има два родитеља, док трут има само једног (он нема оца, већ само мајку). Пчела има две баке (мамину и татину маму) и једног деду (маминог тату), док трут има једну баку и једног деду (мамине родитеље). Напиши програм који одређује колико предака у некој генерацији има трут.

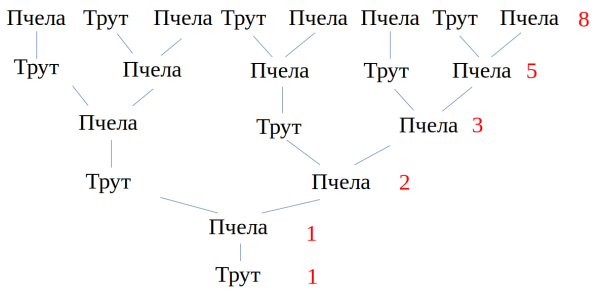
Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 50$) који означава редни број генерације: 0 је генерација самог трута, 1 је генерација његове мајке, 2 је генерација његове баке и деде и тако даље у прошлост.

Излаз: На станадрни излаз исписати укупан број предака трута у генерацији n.

Пример

Улаз	*Излаз*
5	8

Објашњење



Слика 8.1: Породично стабло једног трута

Решење

Обележимо са f_n^m број мушких предака које трут има у генерацији n и f_n^z број женских предака које трут има у генерацији n. Важи да је $f_0^m = 1$ и $f_0^z = 0$ (у нултој генерацији је само трут). За свако $i \geq 0$ важи $f_{i+1}^m = f_i^z$, јер само женске јединке имају очеве, док је $f_{i+1}^z = f_i^m + f_i^z$, јер и мушке и женске јединке имају мајке.

Фибоначијев низ

Уместо два, можемо доћи и до једног рекурентног низа на основу којег добијамо решење. Обележимо са $f_n = f_n^m + f_n^z$ укупан број предака трута у генерацији n. Пошто је $f_0^m = f_1^z = 1$ и $f_0^z = f_1^m = 0$, важи да је $f_0 = f_1 = 1$. За свако $i \geq 0$ важи да је $f_{i+1}^z = f_i^m + f_i^z = f_i$. Зато за свако $i \geq 0$ важи да је $f_{i+2}^m = f_{i+1}^z = f_i$. Зато за свако $i \geq 0$ важи $f_{i+2} = f_{i+2}^m + f_{i+2}^z = f_i + f_{i+1}$. Дакле, важи следеће:

$$f_0 = f_1 = 1 \qquad f_{i+2} = f_{i+1} + f_i.$$

На основу овога можемо закључити да број предака задовољава услове чувеног Фибоначијевог низа бројева у којем је сваки наредни елемент једнак збиру претходна два (елементи тог низа су $1, 1, 2, 3, 5, 8, 13, 21, \ldots$).

Основна рекурзивна имплементација

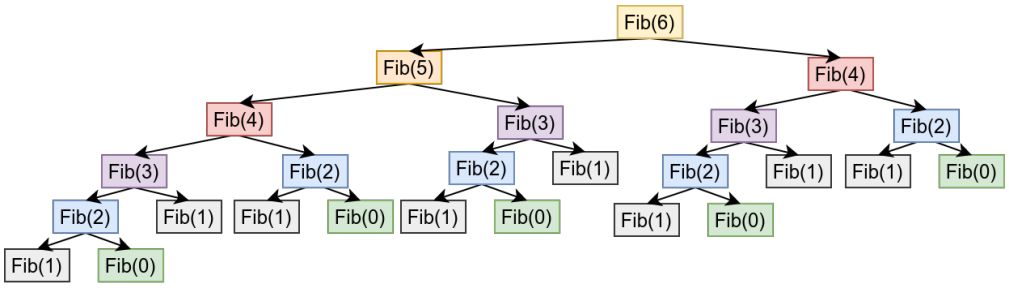
Имплементацију можемо направити рекурзивно, директно на основу дефиниције (ако је $n = 0$ или $n = 1$ функција враћа 1, а у супротном враћа збир резултата рекурзивних позива за вредности $n - 1$ и $n - 2$).

```
long long f(int n) {
  if (n == 0 || n == 1)
    return 1;
  return f(n-1) + f(n-2);
}
```

Директна рекурзивна имплементација је типичан пример неефикасне имплементације јер се рекурзивни позиви за исте вредности параметара понављају више пута. Ако рекурзивну функцију модификујемо тако да на почетку свог извршавања исписује број n, за позив `fib(6)` добијамо следећи испис.

```
6 5 4 3 2 1 0 1 2 1 0 3 2 1 0 1 4 3 2 1 0 1 2 1 0
```

Рекурзивни позиви се могу представити и дрветом.



Слика 8.2: Дрво рекурзивних позива

Позив `fib(6)` врши се један пут, `fib(5)` један пут, `fib(4)` два пута, `fib(3)` три пута, `fib(2)` пет пута, `fib(1)` осам пута и `fib(0)` пет пута. Приметимо да број позива одговара члановима Фибоначијевог низа.

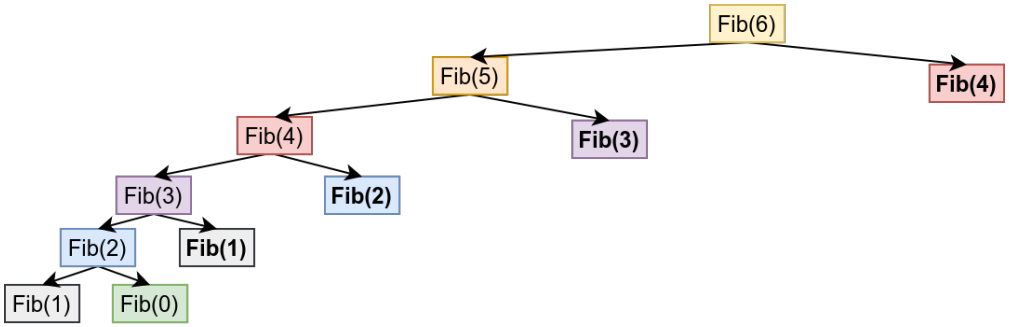
Анализа сложености. Рекурзивна имплементација задовољава једначину $T(n) = T(n-1)+T(n-2)+O(1)$, док је $T(1) = T(0) = O(1)$. Решење ове нехомогене једначине једнако је збиру решења њеног хомогеног дела и неког партикуларног решења, при чему је решење хомогеног дела $F(n) = F(n-1) + F(n-2)$ баш Фибоначијев низ, чије решење се експлицитно може изразити тзв. Бинеовом формулом помоћу $F(n) = \frac{\phi^n - \psi^n}{\phi - \psi}$, где је $\phi = \frac{1+\sqrt{5}}{2}$ и $\psi = \frac{1-\sqrt{5}}{2}$ (напоменимо и да се Бинеова формула може употребити за ефикасно израчунавање чланова Фибоначијевог низа). Решење полазне једначине је $T(n) = O(\frac{1+\sqrt{5}}{2})^n$, тако да је решење експоненцијалне сложености, што је јако неефикасно.

Стога, да би било могуће вршити израчунавање и за веће вредности n, потребно је убрзати имплементацију коришћењем техника динамичког програмирања.

Мемоизација уз коришћење асоцијативног низа

Једна могућност је да се примени мемоизација. Резултате свих рекурзивних позива ћемо памтити у некој помоћној структури података и на почетку сваког рекурзивног позива ћемо претрагом те структуре проверавати да ли је резултат за текућу вредност параметра можда већ израчунат раније. Пошто вредности параметара треба да пресликамо у резултате рекурзивних позива треба да користимо неки облик пресликавања кључева у вредности. То може бити асоцијативни низ, (мапа, односно речник).

Анализа сложености. На овај начин добијамо алгоритам чија је и временска и меморијска сложеност $O(n)$, јер се за свако n израчунавање врши само једном. Дрво рекурзивних позива у овом случају је приказано на слици (спустом дуж леве гране рачунају се вредности за све параметре, док се онда свака од десних грана сасеца јер је резултат већ од раније познат).



Слика 8.3: Дрво рекурзивних позива уз мемоизацију

```
long long fib(int n, unordered_map<int, long long>& memo) {
  // ako je vrednost za parametar n već računata
  // vraćamo ranije izračunatu vrednost
  auto it = memo.find(n);
  if (it != memo.end())
     return it->second;
  // pre nego što vratimo vrednost, pamtimo je u nizu
```

```
  if (n == 0) return memo[n] = 1;
  if (n == 1) return memo[n] = 1;
  return memo[n] = fib(n-1, memo) + fib(n-2, memo);
}

long long fib(int n) {
  // mapa u kojoj se vrednostima parametara rekurzivnog poziva
  // pridružuju vrednosti rekurzivnog poziva
  unordered_map<int, long long> memo;
  return fib(n, memo);
}
```

Мемоизација уз коришћење класичног низа

Иако мапа тј. речник представља природан избор за чување коначног пресликавања, њена употреба у служби мемоизације није честа. Наиме, показује се да се боље перформансе постижу ако се уместо мапе употреби низ (било статички, било динамички алоциран). Тиме се може ангажовати мало више меморије у односу на коришћење асоцијативног низа, међутим, претрага и упис вредности су донекле бржи. У ситуацијама у којима се вредност израчунава за велики број улазних параметара (а код Фибоначија можемо бити сигурни да се приликом израчунавања вредности за параметар n врше позиви за све вредности од 0 до $n-1$), низ може бити чак и меморијски ефикаснији у односу на мапу. Стога се у склопу динамичког програмирања обично користите низови и матрице.

Речник ћемо реализовати помоћу низа тако што ћемо на месту i памтити вредност позива за вредност параметра i. Потребно је још некако обележити вредности параметара у низу за које још не знамо резултате рекурзивних позива. За то се обично користи нека специјална вредност. Ако знамо да ће сви резултати бити ненегативни бројеви, можемо употребити, на пример, -1, а ако знамо да ће бити позитивни бројеви, можемо употребити, на пример 0. Ако немамо таквих претпоставки можемо ангажовати додатни низ логичких вредности којима ћемо експлицитно кодирати да ли за неки параметар знамо или не знамо вредност. Пошто смо сигурни да ће током рекурзије све вредности параметара позитивне и да је највећа вредност која се може јавити као параметар вредност иницијалног позива n, довољно је да алоцирамо низ величине $n+1$.

```
long long fib(int n, vector<long long>& memo) {
  // ako je vrednost za parametar n već računata
  // vraćamo ranije izračunatu vrednost
  if (memo[n] != -1)
    return memo[n];
  // pre nego što vratimo vrednost, pamtimo je u nizu
  if (n == 0) return memo[n] = 1;
  if (n == 1) return memo[n] = 1;
  return memo[n] = fib(n-1, memo) + fib(n-2, memo);
}

long long fib(int n) {
  // alociramo niz veličine n+1 i popunjavamo ga vrednostima -1
  // kojima označavamo da ta vrednost još nije računata
  vector<long long> memo(n+1, -1);
  return fib(n, memo);
}
```

Динамичко програмирање навише

Могуће је применити и динамичко програмирање навише. Техника динамичког програмирања навише подразумева да се уклони рекурзија и да се све вредности у низу попуне неким редоследом. Пошто вредности на вишим позицијама зависе од оних на нижим, низ попуњавамо слева надесно. На прва два места уписујемо нулу и јединицу, а затим у петљи сваку наредну вредност израчунавамо на основу две претходне. На крају враћамо тражену вредност на последњој позицији у низу.

```
vector<long long> dp(n + 1);
dp[0] = dp[1] = 1;
```

```
for (int i = 2; i <= n; i++)
  dp[i] = dp[i-1] + dp[i-2];
cout << dp[n] << endl;
```

Динамичко програмирање навише - меморијска оптимизација

Једноставно се примећује да вредност наредног члана Фибоначијевог низа зависи само од вредности његова претходна два члана. Зато можемо направити меморијску оптимизацију и не морамо истовремено чувати све чланове у низу.

```
long long fpp = 1;
long long fp = 1;
for (int i = 2; i <= n; i++) {
  long long f = fp + fpp;
  fpp = fp;
  fp = f;
}

cout << fp << endl;
```

Анализа сложености. Меморијска сложеност овог решења је $O(1)$, док је временска сложеност $O(n)$.

"Рецепт" за динамичко програмирање

Низ корака који смо применили у овом задатку јављаће се веома често.

1. Индуктивно-рекурзивном конструкцијом конструише се рекурзивна дефиниција која је неефикасна јер се исти позиви прекапају тј. функција се за исте аргументе позива више пута.
2. Техником мемоизације побољшава се сложеност тако што се у помоћном речнику (најчешће имплементираном помоћу низа или матрице) чувају израчунати резултати рекурзивних позива.
3. Уместо технике мемоизације која је вођена рекурзијом и у којој се вредности попуњавају по потреби, рекурзија се може елиминисати и речник (низ тј. матрица) се може цео попунити (неким редоследом).
4. Често је могуће да се изврши меморијска оптимизација на основу тога што се примећује да након попуњавања одређених елемената низа тј. матрице неке вредности (ранији елементи, раније врсте или колоне) више нису потребне, тако да се уместо истовремног памћења свих елемената памти само неколико претходних (они који су потребни за даље попуњавање).

Два рекурентна низа

Имплементацију је могуће направити и коришћењем засебних низова који описују број мушких тј. женских предака. Подсетимо се, важи, $f_0^m = 1, f_0^z = 1$, и за свако $i \geq 0$ важи $f_{i+1}^m = f_i^z$, као и $f_{i+1}^z = f_i^m + f_i^z$. Ове једнакости нам дају две узајамно рекурентно дефинисане серије бројева, на основу чега можемо направити било рекурзивну, било итеративну имплементацију. Коначно решење представља збир $f_n^m + f_n^z$.

У рекурзивној имплементацији је потребно имплементирати две узајамно рекурзивне функције. У језику C++ је потребно да је компилатор познаје сваку од функција пре позива, па је стога на почетку добро декларисати тј. навести прототипове обе функције (мада би довољно било декларисати само другу од њих).

И у оваквој рекурзивној имплементацији рекурзивни позиви за исте вредности улазних параметара се понављају више пута и стога је она прилично неефикасна.

```
// broj muskih jedinki u generaciji n
long long fm(int n);
// broj zenskih jedinki u generaciji n
long long fz(int n);

// broj muskih jedinki u generaciji n
long long fm(int n) {
  // u generaciji 0 postoji samo trut
  if (n == 0)
    return 1;
  // samo zenke iz naredne generacije imaju oceve
```

```
  return fz(n-1);
}

// broj zenskih jedinki u generaciji n
long long fz(int n) {
  // u generaciji 0 postoji samo trut
  if (n == 0)
    return 0;
  // i muzjaci i zenke iz naredne generacije imaju majke
  return fm(n-1) + fz(n-1);
}

// ukupan broj jedinki u generaciji n
long long f(int n) {
  // sabiramo muske i zenske jedinke u generaciji n
  return fm(n) + fz(n);
}
```

Наравно, и у овој имплементацији је могуће применити динамичко програмирање и у крајњој инстанци добити наредну итеративну имплементацију.

У итеративној имплементацији одржавамо две променљиве у којима чувамо текући број мушких и женских јединики у генерацији (иницијализујемо их на 1 и 0, на основу нулте генерације у којој постоји само трут). Затим n пута ажурирамо вредности на основу изведених рекурентних веза. Потребно је једино обратити пажњу да се ажурирање друге променљиве врши увек на основу обе старе вредности, а не на основу ажуриране вредности прве променљиве (за то је потребно употребити помоћну променљиву).

```
int n;
cin >> n;
// broj muskih i zenskih jedinki
// u generaciji 0 postoji samo trut
long long fm = 1, fz = 0;
// n puta azuriramo brojeve prelazeci sa tekuce na prethodnu
// generaciju
for (int i = 1; i <= n; i++) {
  // samo zenke imaju oceve, dok majke imaju i muzjaci i zenke
  long long fm_ = fz, fz_ = fm + fz;
  fm = fm_; fz = fz_;
}
// ukupan broj jedinki u generaciji n
cout << fm + fz << endl;
```

Могуће је дефинисати и репно-рекурзивну функцију која ефикасно израчунава решење.

```
// izracunaj ukupan broj jedinki u generaciji n, ako znas da u
// generaciji i postoji fm muskih i fz zenskih jedinki
long long f_(long long fm, long long fz, int i, int n) {
  // ovo je bas generacija n - izracunaj i vrati ukupan broj jedinki
  if (i == n)
    return fm + fz;
  // mozes da izracunas broj muskih i zenskih jedinki u generaciji i+1
  return f_(fz, fm+fz, i+1, n);
}

// izracunava broj jediniki u generaciji n
long long f(int n) {
  // u generaciji 0 postoji jedna muska i nula zenskih jediniki -
  // kreni od toga
  return f_(1, 0, 0, n);
}
```

8.2 Бројање комбинаторних објеката

Једна веома значајна примена технике динамичког програмирања је у проблемима пребројавања.

Задатак: Број комбинација

Напиши програм који одређује број комбинација без понављања дужине k из скупа од k елемената (тј. број различитих комбинација у игри лото ако се из бубња који садржи n лоптица извлачи њих k).

Улаз: Са стандардног улаза се извлачи број k ($1 \leq k \leq n$) и број n ($1 \leq n \leq 40$).

Излаз: На стандардни излаз исписати тражени број комбинација.

Пример 1

Улаз	*Излаз*
3 5	10

Објашњење

То су комбинације $(1, 2, 3)$, $(1, 2, 4)$, $(1, 2, 5)$ $(1, 3, 4)$, $(1, 3, 5)$, $(1, 4, 5)$, $(2, 3, 4)$, $(2, 3, 5)$, $(2, 4, 5)$ и $(3, 4, 5)$.

Пример 2

Улаз

```
7
39
```

Излаз

```
15380937
```

Решење

Иако постоје разни начини да се до решења овог задатка дође, приказаћемо технику засновану на томе да програм који набраја све комбинаторне објекте мало по мало трансформишемо до ефикасног прорама који их броји. Ова техника није специфична за комбинације без понављања и може се применити на бројање било које врсте комбинаторних објеката које набрајамо рекурзивном функцијом.

Рецимо да је број комбинација једнак биномном коефицијенту

$$\binom{n}{k} = \frac{n!}{(n-k)!k!},$$

међутим израчунавање на основу директне примене ове формуле би веома брзо довело до прекорачења (услед веома брзог раста факторијелске функције). Мало боља ситуација је да се израчуна

$$\binom{n}{k} = \frac{n(n-1)\ldots(n-k+1)}{k!},$$

но ни то у потпуности не уклања проблем прекорачења, јер именилац може бити превелики.

Рекурзивна функција која израчунава број комбинација

Можемо кренути од процедуре за генерисање свих комбинација у којој се одржава интервал $[n_{min}, n_{max}]$ из којег се могу узети вредности којима се проширује започета комбинација и у којој се кроз два рекурзивна позива разматра могућност да се вредност n_{min} уврсти у комбинацију и могућност да се она не уврсти у комбинацију. Та је процедура објашњена у задатку Све комбинације,

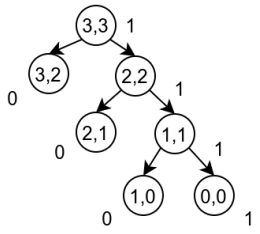
Уместо процедуре која исписује комбинације, дефинишемо фукцију која враћа број комбинација. Пошто нам је битан само број комбинација, а не и саме комбинације, можемо у потпуности избацити низ који се попуњава и уместо њега прослеђивати само његову дужину k.

```
long long brojKombinacija(int i, int k,
                          int n_min, int n_max) {
  // ako je popunjen ceo niz postoji jedna kombinacija
  if (i == k) return 1;
  // ako niz nije moguće popuniti do kraja, tada nema kombinacija
  if (k - i > n_max - n_min + 1)
    return 0;
  // broj kombinacija je jednak zbiru kombinacija u dva slučaja
  return brojKombinacija(i+1, k, n_min+1, n_max) +
         brojKombinacija(i, k, n_min+1, n_max);
}

long long brojKombinacija(int K, int N) {
  return brojKombinacija(0, K, 1, N);
}
```

Можемо приметити да нам конкретне вредности k и i нису битне, већ је битан само број елемената у интервалу $[i, k)$ тј. разлика $k-i$. Слично, нису нам битне ни конкретне вредности n_{max} и n_{min} већ само број елемената у сегменту $[n_{min}, n_{max}]$ тј. вредност $n_{max} - n_{max} + 1$. Ако те две величине заменимо са k тј. n добијамо наредну дефиницију.

```
long long brojKombinacija(int k, int n) {
  // ako je popunjen ceo niz postoji jedna kombinacija
  if (k == 0) return 1;
  // ako niz nije moguće popuniti do kraja, tada nema kombinacija
  if (k > n) return 0;
  // broj kombinacija je jednak zbiru kombinacija u dva slučaja
  return brojKombinacija(k-1, n-1) + brojKombinacija(k, n-1);
}
```

Ако функцију позовемо за вредности $k \leq n$, случај $k > n$ може наступити једино из другог рекурзивног позива за $k = n$ (јер однос између k и n у првом рекурзивном позиву остаје непромењен, а у другом се мења само за 1). Међутим, како је илустровано на наредној слици, у случају позива функције за $k = n$ добиће се увек повратна вредност 1 (један рекурзивни позив ће увек враћати нулу, а други ће проузроковати смањивање оба аргумента све док се не дође до $k = n = 0$, када ће се 1 вратити на основу првог излаза из рекурзије), што је сасвим у складу са тим да тада постоји само једна комбинација.



Слика 8.4: Израчунавање коефицијената за $k = n$

На основу овога из рекурзије можемо изаћи за $k = n$ вративши вредност 1, чиме онда елиминишемо потребу за провером да ли је $k > n$ (наравно, под претпоставком да ћемо функцију позивати само за $k \leq n$).

Примећујемо да смо овом трансформацијом добили чувене особине биномних коефицијената.

$$\binom{n}{0} = 1, \qquad \binom{n}{n} = 1, \qquad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Оне чине основу Паскаловог троугла у ком се налазе биномни коефицијенти (у наредној табели коефицијент $\binom{n}{k}$ је написан у врсти n и колони k, да би се приказао уобичајени облик Паскаловог троугла, који се попуњава врсту по врсту).

```
1                      (0,0)
1  1                   (1,0) (1,1)
1  2  1                (2,0) (2,1) (2,2)
1  3  3  1             (3,0) (3,1) (3,2) (3,3)
1  4  6  4  1          (4,0) (4,1) (4,2) (4,3) (4,4)
1  5 10 10  5  1       (5,0) (5,1) (5,2) (5,3) (5,4) (5,5)
1  6 15 20 15  6  1    (6,0) (6,1) (6,2) (6,3) (6,4) (6,5) (6,6)
```

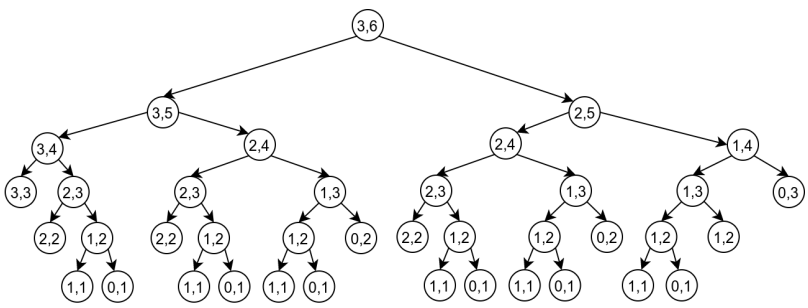
Прва веза говори да су елементи прве колоне увек једнаки 1, друга да су на крају сваке врсте елементи такође једнаки 1, а трећа да је сваки елемент у троуглу једнак збиру елемента непосредно изнад њега и елемента непосредно испред тог.

Наравно, до ових формула и до рекурзивне дефиниције смо могли доћи и директно, разматрањем дефиниција комбинација, на пример, у моделу где се k куглица без враћања извлаче из бубња у ком се налази n различитих куглица. Постоји јединствен начин да се из бубња не извуче ни једна куглица. Такође, постоји јединствен начин да се из бубња извуче свих n куглица. У супротном (ако је $0 < k < n$), тада све начине раздвајамо на оне у којима јесте и на оне у којима није извучена прва куглица (куглица са најмањим бројем). Ако она јесте извучена, преостало је да се извуче још $k-1$ куглица из бубња у ком се налази $n-1$ куглица, а ако није, тада је преостало да се извуче још k куглица из бубња у ком се налази још $n-1$ куглица (пошто смо претпоставили да у другој групи начина куглица са најмањим бројем неће бити међу извученим куглицама, можемо је одмах избацити из бубња и склонити негде са стране).

```
long long brojKombinacija(int k, int n) {
  // ako je popunjen ceo niz postoji jedna kombinacija
  if (k == 0) return 1;
  // ako treba popuniti još tačno n elemenata, tada postoji
  // tačno jedna kombinacija
  if (k == n) return 1;
  // broj kombinacija je jednak zbiru kombinacija u dva slučaja
  return brojKombinacija(k-1, n-1) + brojKombinacija(k, n-1);
}
```

Анализа сложености. Време које се утроши у сваком рекурзивном позиву (не рачунајући рекурзивне позиве који се из њега из позивају) је очигледно $O(1)$. Једначина којом се описује време рада функције је $T(k,n) = T(k-1,n-1) + T(k,n-1) + O(1)$, $T(0,n) = T(n,n) = O(1)$, и време извршавања је $T(k,n) = O(\binom{n}{k})$. За фиксирано k, ово је сложеност описана полиномом променљиве n, који може бити веома високог степена (k), док за фиксирано n овај број расте експоненцијално са порастом k. У сваком случају, јасно је да је сложеност изузетно висока и да је овај програм практично употребљив за веће вредности n и k.

Разлог овој неефикасности су поновљени рекурзивни позиви, што се може видети на слици. Сваком листу дрвета одговара тачно једна комбинација, па пошто укупно комбинација има $\binom{n}{k}$, укупан број рекурзивних позива је ограничен са $2\binom{n}{k}$ (јер у бинарном дрвету не може бити више него дупло више чворова него листова). У сваком позиву се врши $O(1)$ операција, па је сложеност $O(\binom{n}{k})$.



Слика 8.5: Дрво рекурзивних позива - сваки лист одговара тачно једној комбинацији, а приметно је понављање идентичних рекурзивних позива

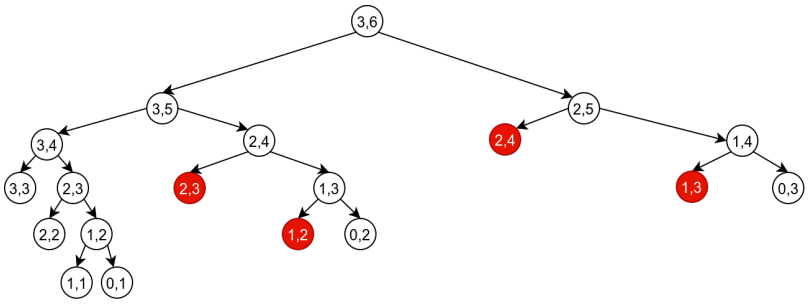
Мемоизација

Иако коректна, горња функција је неефикасна и може се поправити техником динамичког програмирања. Најједноставније прилагођавање је да се употреби мемоизација. Пошто функција има два параметра, за мемоизацију ћемо употребити матрицу. Ако се $\binom{n}{k}$ памти у матрици на позицији (n, k), матрицу можемо алоцирати на $n + 1$ врста, где последња врста има $n + 1$ елемената, а свака претходна један елемент мање (у матрицу ће се попуњавати елементи Паскаловог троугла).

Пошто нас неће занимати вредности веће од полазног k и пошто се и k и n смањују током рекурзије, при чему је $k \leq n$, можемо и одсећи део троугла десно од позиције k.

Пошто су бројеви комбинација увек већи од нуле, вредности 0 у матрици ће нам означавати да позив за те параметре још није извршен.

Анализа сложености. И меморијска и временска сложеност мемоизоване верзије је $O(nk)$. Наиме, у дрвету рекурзивних позива се сваки чвор може јавити највише два пута. Пошто сваки чвор садржи пар бројева таквих да је први од 0 до k, а други од 0 до n, укупан број чворова је одозго ограничен са $2nk$ (а може бити и мањи, јер се неки парови не могу јавити).



Слика 8.6: Дрво рекурзивних позива уз мемоизацију

Ако је познато горње ограничење на вредности n и k тада уместо матрице која се динамички алоцира можемо употребити статички алоцирану матрицу, чиме се избегава непотребно трошење времена на динамичку алокацију, по цену да програм и за мање вредности n и k заузима велику количину меморије. У том случају није могуће алоцирати само троугаони део матрице, већ цео, правоугаони блок меморије.

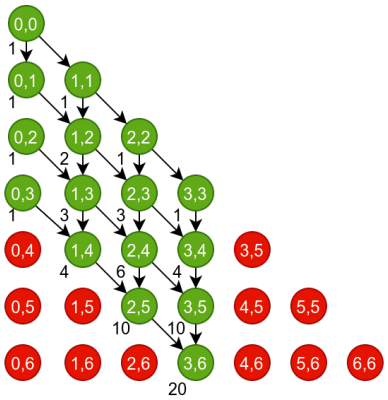
```
long long brojKombinacija(int k, int n, vector<vector<long long>>& memo) {
  // ako smo već računali broj kombinacija, ne računamo ga ponovo
  if (memo[n][k] != 0) return memo[n][k];

  // broj kombinacija na početku i na kraju svake vrste jednak je 1
  if (k == 0 || k == n) return memo[n][k] = 1;
  // broj kombinacija u sredini jednak je zbiru
  // broja kombinacija iznad i iznad levo od tekućeg elementa
  return memo[n][k] = brojKombinacija(k-1, n-1, memo) +
                      brojKombinacija(k, n-1, memo);
}

long long brojKombinacija(int K, int N) {
  // alociramo prostor za rezultate rekurzivnih poziva koji se
  // mogu desiti i popunjavamo matricu nulama
  vector<vector<long long>> memo(N+1);
  for (int n = 0; n <= N; n++)
    memo[n].resize(min(K+1, n+1), 0);
  // pozivamo funkciju koja će izračunati traženi broj
  return brojKombinacija(K, N, memo);
}
```

Динамичко програмирање навише

Уместо мемоизације можемо употребити и динамичко програмирање навише, ослободити се рекурзије и попунити троугао врсту по врсту наниже. Попуњавање целог троугла је прилично једноставно.

```
long long brojKombinacija(int K, int N) {
  // alociramo prostor za smeštanje celog trougla
  vector<vector<long long>> dp(N+1);
  for (int n = 0; n <= N; n++)
    dp[n].resize(n+1);
  // obrađujemo vrstu po vrstu
  for (int n = 0; n <= N; n++) {
    // na početku svake vrste nalazi se 1
    dp[n][0] = 1;
    // unutrašnje elemente izračunavamo kao zbir elemenata iznad njih
    for (int k = 1; k < n; k++)
      dp[n][k] = dp[n-1][k-1] + dp[n-1][k];
    // na kraju svake vrste nalazi se 1
    dp[n][n] = 1;
  }
  // vraćamo traženi rezultat
  return dp[N][K];
}
```

И у овом случају можемо одсећи непотребне десне колоне у троуглу. Могуће је одсећи и део испод одговарајуће дијагонале – таква одсецања се обично не ради у динамичком програмирању навише јер не мењају асимптотску сложеност, док их рекурзивна имплементација заснована на мемоизацији природно избегава. На наредној слици су обележени елементи Паскаловог троугла који су потребни за израчунавање броја $\binom{6}{3} = 20$.



Слика 8.7: Елементи Паскаловог троугла потребни за израчунавање броја $\binom{6}{3}$

```
long long brojKombinacija(int K, int N) {
  // alociramo prostor za smeštanje relevantnog dela trougla
  vector<vector<long long>> dp(N+1);
  for (int n = 0; n <= N; n++)
    dp[n].resize(min(K+1, n+1));
  // trougao popunjavamo kolonu po kolonu
  for (int n = 0; n <= N; n++) {
    // na početku svake vrste nalazi se 1
    dp[n][0] = 1;
    // unutrašnje elemente izračunavamo kao zbir elemenata iznad njih
    for (int k = 1; k <= min(n-1, K); k++)
      dp[n][k] = dp[n-1][k-1] + dp[n-1][k];
    // ako je potrebno da znamo krajnji element kolone, postavljamo
    // ga na vrednost 1
    if (n <= K)
```

```
      dp[n][n] = 1;
  }
  // vraćamo traženi rezultat
  return dp[N][K];
}
```

Пажљивијом анализом претходног кода видимо да, како је то обично случај у динамичком програмирању, не морамо истовремено чувати све елементе матрице, јер свака врста зависи само од претходне и довољно је уместо матрице чувати само њене две врсте (претходну и текућу). Заправо, довољно је чувати само један вектор врсту ако је пажљиво попуњавамо и ако током њеног ажурирања у једном њеном делу чувамо текућу, а у другом наредну врсту. Пошто елемент (n, k) зависи од елемента $(n-1, k-1)$ и од елемента $(n-1, k)$ значи да сваки елемент зависи од елемената који су лево од њега, али не од елемената који су десно од њега. Зато ћемо вектор попуњавати здесна налево. Претпоставићемо да током ажурирања важи инваријанта да се на позицијама строго већим од k налазе елементи врсте n, а да се на позицијама мањим или једнаким од k налазе елементи врсте $n-1$. Ажурирање започиње тиме што на крај врсте допишемо вредност 1 (осим у случају када вршимо сасецање десног дела троугла) и наставља се тако што се елемент на позицији k увећа за вредност на позицији $k-1$. Заиста, пре ажурирања се на позицији k налази вредост троугла са позиције $(n-1, k)$, док се на позицији $k-1$ налази вредност троугла са позиције $(n-1, k-1)$. Њихов збир је вредност троугла на позицији (n, k), па се он уписује на позицију k и након тога се k смањује за 1, чиме се инваријанта одржава. Ажурирање се врши до позиције $k = 1$, јер се на позицији $k = 0$ у свим врстама налази вредност 1.

Меморијска сложеност овог решења је $O(k)$, док је временска $O(n \cdot k)$. Приметимо како смо од веома неефикасног решења експоненцијалне сложености техником динамичког програмирања добили веома ефикасно и уз то прилично једноставно решење.

```
long long brojKombinacija(int K, int N) {
  // tekuća vrsta
  vector<long long> dp(K+1);
  // na početku svake vrste nalazi se 1
  dp[0] = 1;
  // trougao popunjavamo vrstu po vrstu
  for (int n = 1; n <= N; n++) {
    // vrstu ažruriramo zdesna nalevo
    // na kraju svake vrste nalazi se 1
    if (n <= K) dp[n] = 1;
    // ažuriramo unutrašnje elemente
    for (int k = min(n-1, K); k > 0; k--)
        dp[k] += dp[k-1];
  }
  // vraćamo traženi rezultat
  return dp[K];
}
```

Другачија рекурзивна дефиниција

Рецимо и да смо могли кренути од алгоритма набрајања свих комбинација у ком се у петљи разматрају сви кандидати за елемент на текућој позицији. Тиме би се добио алгоритам који би елемент $\binom{n}{k}$ рачунао по следећој формули:

$$\binom{n}{k} = \sum_{n'=k}^{n} \binom{n'}{k-1}.$$

Задатак: Дигитални бројач

$2n$-то цифрени дигитални бројач који одбројава од $000\ldots000$ до $999\ldots999$ емитује звучни сигнал сваки пут када је сума првих n цифара једнака суми последњих n цифара. На пример, за шестоцифрени дигитални бројач звучни сигнал се пушта за $000000, 001001, 001010, \ldots, 999999$. Написати програм који ће одредити колико пута ће бити емитован звучни сигнал.

Улаз: У првој линији стандардног улаза налази се природан број n ($1 \leq n \leq 9$).

Излаз: На стандарном излазу приказати колико постоји $2n$-цифрених бројева са траженим својством.

Пример

Улаз	*Излаз*
3	55252

Решење

Наивна решења

Директно решење је да се у петљи прођу сви бројеви од $00\ldots0$ до $99\ldots9$, да се за сваки број одреди збир леве и десне половине, и ако су они једнаки, да се увећа бројач.

Анализа сложености. Пошто постоји 10^{2n} бројева са $2n$ цифара и пошто се за сваки од њих збир цифара сваке половине израчунава у n корака, сложеност овог алгоритма је $O(2n \cdot 10^{2n})$.

```
// uklanja poslednjih n cifara broja x odredjujuci njihov zbir
int zbirNPoslednjihCifara(long long& x, int n) {
  int zbir = 0;
  for (int i = 0; i < n; i++) {
    zbir += x % 10;
    x /= 10;
  }
  return zbir;
}

// odredjuje zbirove prve i druge polovine broja x koji ima 2n cifara
void zbiroviCifara(long long x, int n,
                   int& zbirPrvePolovine, int& zbirDrugePolovine) {
  zbirDrugePolovine = zbirNPoslednjihCifara(x, n);
  zbirPrvePolovine = zbirNPoslednjihCifara(x, n);
}

// izracunava broj 2n-tocifrenih brojeva kod kojih je zbir prvih n i
// zbir drugih n cifara jednak
int brojBrojeva(int n) {
  long long ukupnoBrojeva = 0;
  long long max = (long long)pow(10, 2*n) - 1;
  for (long long i = 0; i <= max; i++) {
    int zbirPrvePolovine, zbirDrugePolovine;
    zbiroviCifara(i, n, zbirPrvePolovine, zbirDrugePolovine);
    if (zbirPrvePolovine == zbirDrugePolovine)
      ukupnoBrojeva++;
  }
  return ukupnoBrojeva;
}
```

Једно решење је да се лева и десна половина броја набрајају у две угнежђене петље и броји колико пута ће збир цифара спољашње бројачке променљиве бити једнак збиру цифара унутрашње бројачке променљиве. Збир леве половине броја се тада може рачунати само једном по извршавању целокупне унутрашње петље.

Анализа сложености. Сложеност овог алгоритма је $O(n \cdot 10^{2n})$ (кључни проблем је то што се сваки од 10^n n-тоцифрених бројева комбинује са сваким од 10^n n-тоцифрених бројева, што захтева 10^{2n} корака).

```
// izracunava zbir cifara u dekadnom zapisu broja x
int zbirCifara(long long x) {
  int zbir = 0;
  while (x > 0) {
    zbir += x % 10;
    x /= 10;
  }
  return zbir;
```

```
}

// izracunava broj 2n-tocifrenih brojeva kod kojih je zbir prvih n i
// zbir drugih n cifara jednak
int brojBrojeva(int n) {
  long long ukupnoBrojeva = 0;
  long long max = (long long)pow(10, n) - 1;
  for (int prvaPolovina = 0; prvaPolovina <= max; prvaPolovina++) {
    int zbirPrvePolovine = zbirCifara(prvaPolovina);
    for (int drugaPolovina = 0; drugaPolovina <= max; drugaPolovina++) {
      int zbirDrugePolovine = zbirCifara(drugaPolovina);
      if (zbirPrvePolovine == zbirDrugePolovine)
        ukupnoBrojeva++;
    }
  }
  return ukupnoBrojeva;
}
```

Број n-тоцифрених бројева датог збира

Пошто лева и десна половина треба да имају исти збир (на пример, s), и једна и друга половина ће бити n-тоцифрени бројеви чији је збир цифара s. Збир цифара n-тоцифреног броја узима вредности од 0 (за број $00\ldots0$) до $9 \cdot n$ (за број $99\ldots9$). За свако такво s треба да одредимо на колико разних начина можемо да одаберемо два броја чији је збир цифара s. Означимо са b_s број n-тоцифрених бројева којима је збир цифара s (број између 0 и $9 \cdot n$). Дакле и прву и другу половину можемо изабрати на b_s начина (пошто оне не морају бити различите), тако да постоји укупно b_s^2 бројева чија лева и десна половина имају збир s. Зато ће тражени број бити једнак $b_0^2 + b_1^2 + ... + b_{9n}^2$.

Остаје питање како одредити број n-тоцифрених бројева чији је збир s.

Бројање бројева датог збира

Један начин је да се уведе низ бројача, по један за сваку вредност c. Таква употреба низа у функцији пресликавања је приказано у задатку Фреквенција знака. У петљи се обилазе сви бројеви од 0 до $10^n - 1$, за сваки се одређује збир цифара и увећава се бројач бројева добијеног збира.

Анализа сложености. Пошто се за сваки од 10^n n-тоцифрених бројева збир цифара израчунава у сложености $O(n)$, и пошто се коначни број рачуна у $9n$ корака петље тј. у сложености $O(n)$, сложеност овог алгоритма је $n \cdot 10^n$.

```
// izracunava broj 2n-tocifrenih brojeva kod kojih je zbir prvih n i
// zbir drugih n cifara jednak
long long brojBrojeva(int n) {
  // broj brojeva sa datim zbirom cifara
  vector<long long> brojBrojevaDatogZbiraCifara(9*n + 1, 0);

  // za sve n-tocifrene brojeve izmedju 00..0 i 99...9
  long long st10 = Stepen(10, n);
  for (long long broj = 0; broj < st10; broj++)
    // odredjujemo zbir cifara i uvecavamo broj brojeva sa tim zbirom
    // cifara
    brojBrojevaDatogZbiraCifara[ZbirCifara(broj)]++;

  // ukupan broj brojeva cija leva i desna polovina imaju isti broj cifara
  long long ukupnoBrojeva = 0;
  // za svaki moguci zbir cifara polovine broja
  for (int zbir = 0; zbir <= 9*n; zbir++) {
    // postoji b n-tocifrenih brojeva koji imaju zbir cifara zbir
    long long b = brojBrojevaDatogZbiraCifara[zbir];
    // levu polovinu broja mozemo odabrati na b nacina i desnu
    // polovinu na b nacina, tako da ukupno takvih brojeva ima b*b
```

```
    ukupnoBrojeva += b * b;
  }
  return ukupnoBrojeva;
}
```

Бројање партиција динамичким програмирањем

Један начин да се одреди број n-тоцифрених бројева чији је збир цифра b_s је да се примени рекурзивно решење по броју цифара.

Уведимо стога ознаку $b_{k,s}$ за број k-тоцифрених бројева чији је збир цифара s.

- Ако је $k = 1$, бројева $b_{1,s}$ има 1 за $0 \leq s \leq 9$, тј. 0 у супротном.
- Ако је $k > 1$ тада на прво место можемо поставити било коју цифру c од 0 до 9 (тј. до s ако је $s < 9$). Када њу поставимо, остале цифре можемо поставити на $b_{k-1,s-c}$ начина. Дакле,

$$b_{k,s} = \sum_{c=0}^{\min(s,9)} b_{k-1,s-c}.$$

Пошто се у овој рекрузивној формулацији рекурзивни позиви преклапају, потребно је употребити технику динамичког програмирања да би се решење убрзало. Једна могућност је да употребимо мемоизацију.

```
// broj nacina da se napravi broj koji ima brojCifara cifara i kome je
// zbir cifara jednak zbirCifara
long long brojParticija(int zbirCifara, int brojCifara) {
  // koristimo memoizaciju da bismo izbegli da vise puta racunamo
  // jedan te isti rezultat - najjednostavnija (ali ne i
  // najefikasnija) implementacija cuva ranije rezultate u recniku
  static map<pair<int, int>, long long> memo;
  // proveravamo da li smo vec racunali ovaj rezultat i ako jesmo
  // vracamo raniji rezultat
  auto p = make_pair(zbirCifara, brojCifara);
  if (memo.find(p) != memo.end())
    return memo[p];

  if (brojCifara == 1)
    // brojevi izmedju 0 i 9 se mogu na jedan nacin predstaviti kao
    // zbir jedne cifre, a veci brojevi od toga se ne mogu predstaviti
    return memo[p] = zbirCifara < 10 ? 1 : 0;
  else {
    long long rezultat = 0;
    // na prvo mesto postavljamo redom jednu po jednu cifru
    for (int cifra = 0; cifra <= 9 && cifra <= zbirCifara; cifra++)
      // rekurzivno racunamo broj particija preostalog zbira sa jednom
      // cifrom manje
      rezultat += brojParticija(zbirCifara - cifra, brojCifara - 1);
    return memo[p] = rezultat;
  }
}


// izracunava broj 2n-tocifrenih brojeva kod kojih je zbir prvih n i
// zbir drugih n cifara jednak
long long brojBrojeva(int n) {
  long long ukupnoBrojeva = 0;
  for (int zbirCifara = 0; zbirCifara <= 9*n; zbirCifara++) {
    long long b = brojParticija(zbirCifara, n);
    ukupnoBrojeva += b * b;
  }
```

```
  return ukupnoBrojeva;
}
```

Динамичко програмирање навише

Уместо мемоизације можемо употребити и динамичко програмирање навише. Вредности $b_{k,s}$ можемо чувати у матрици, чије попуњавање почињемо од првог реда (за $k = 1$) и попуњавамо је затим ред по ред, за растуће вредности k.

Пример. На пример, да бисмо одредили број $b_{3,15}$ попуњавамо следећу матрицу.

```
    0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
-------------------------------------------------
1 | 1  1  1  1  1  1  1  1  1  1   0  0  0  0  0  0
2 | 1  2  3  4  5  6  7  8  9 10   9  8  7  6  5  4
3 | 1  3  6 10 15 21 28 36 45 55  63 69 73 75 75 73
```

```
// alociramo matricu dimenzije m x n
vector<vector<long long>> alociraj(int m, int n) {
  vector<vector<long long>> rezultat(m);
  for (int i = 0; i < m; i++)
    rezultat[i].resize(n);
  return rezultat;
}

// vraca matricu koja za svaki brojCifara od 1 do maxBrojCifara i
// svaki zbirCifara od 1 do maxZbirCifara na mestu
// [brojCifara][zbirCifara] sadrzi broj brojeva koji imaju brojCifara
// cifara i zbir cifara jednak broju zbirCifara
vector<vector<long long>> brojParticija(int maxBrojCifara, int maxZbirCifara) {
  vector<vector<long long>> dp = alociraj(maxBrojCifara + 1, maxZbirCifara + 1);

  // brojevi izmedju 0 i 9 se mogu na jedan nacin predstaviti kao zbir
  // jedne cifre, a veci brojevi od toga se ne mogu predstaviti
  for (int zbirCifara = 0; zbirCifara <= maxZbirCifara; zbirCifara++)
    dp[1][zbirCifara] = zbirCifara < 10 ? 1 : 0;

  // odredjujemo broj razlaganja na vise cifara
  for (int brojCifara = 2; brojCifara <= maxBrojCifara; brojCifara++) {
    // zbir moramo da obilazimo unazad da bismo mogli da koristimo
    // samo jedan niz
    for (int zbirCifara = 0; zbirCifara <= maxZbirCifara; zbirCifara++) {
      dp[brojCifara][zbirCifara] = 0;
      // na prvo mesto postavljamo redom jednu po jednu cifru
      for (int cifra = 0; cifra <= 9 && cifra <= zbirCifara; cifra++)
        // rekurzivno racunamo broj particija preostalog zbira sa jednom
        // cifrom manje
        dp[brojCifara][zbirCifara] += dp[brojCifara - 1][zbirCifara - cifra];
    }
  }
  return dp;
}

// izracunava broj 2n-tocifrenih brojeva kod kojih je zbir prvih n i
// zbir drugih n cifara jednak
long long brojBrojeva(int n) {
  // za svaki broj od 0 do 9*n nas zanima koliko postoji razlicitih
  // n-tocifrenih brojeva ciji je zbir cifara taj broj
  vector<vector<long long>> dp = brojParticija(n, 9*n);

  // ukupan broj brojeva cija leva i desna polovina imaju isti broj cifara
```

```
  long long ukupnoBrojeva = 0;
  // za svaki moguci zbir cifara polovine broja
  for (int zbirCifara = 0; zbirCifara <= 9*n; zbirCifara++)
    // postoji dp[n][zbirCifara] nacina da napravimo levu polovinu i
    // dp[n][zbirCifara] nacina da napravimo desnu polovinu broja
    // tj. dp[zbirCifara]*dp[zbirCifara] nacina da odaberemo broj kome
    // i leva i desna polovina imaju zbir cifara zbirCifara
    ukupnoBrojeva += dp[n][zbirCifara] * dp[n][zbirCifara];

  return ukupnoBrojeva;
}
```

Анализа сложености. Максимални број цифара је n, а максимални збир цифара је $9n$, па је димензија матрице $(n+1) \times (9n+1)$. Дакле, попуњава се $O(n^2)$ поља матрице. Одређено време које се троши на алокацију матрице би се могло избећи коришћењем статички алоциране матрице (под претпоставком да је унапред познато горње ограничење Свако поље захтева највише 10 корака сабирања (за цифре од 0 до 9), па се свако поље попуњава у времену $O(1)$, додуше уз велики константни фактор. То би се могло поправити ако би се чували парцијални збирови елемената сваке врсте. Употреба парцијалних збирова за оптимизацију израчунавања збирова сегмената датог низа приказана је, на пример, у задатку Збирови сегмената, а у контексту оптимизације динамичког програмирања у задатку Број цик-цак партиција. Након попуњавања матрице, број елемената се израчунава у времену $O(n)$, без коришћења додатне меморије. Дакле, и временска и меморијска сложеност је $O(n^2)$. вредности n).

Меморијска оптимизација

Приметимо да се за израчунавање вредности у реду k користе само вредности у реду $k-1$. Зато није потребно чувати целу матрицу, већ само њену претходну врсту. Додатно, ако приметимо да за израчунавање вредности $b_{k,s}$ нису потребне вредности из претходног реда за збирове $s' > s$, довољно је да податке чувамо само у једном низу у којем ћемо вредности наредног реда од вредности претходног добијати тако што ћемо кренути од краја и ажурирати једну по једну вредност. Пошто $b_{k,s}$ зависи од $b_{k-1,s}$, потребна нам је једна помоћна променљива у којој ћемо израчунати резултат пре него што га упишемо на место s у низу или вредност можемо добити тако што вредност на месту s која одговара цифри 0 увећамо за вредности на позицијама $s-c$ које одговарају цифрама 1, 2 и тако даље.

Анализа сложености. Димензија матрице је $O(n^2)$, што је и време потребно за њено попуњавање (уз велики константни фактор, због сабирања 10 елемената). Укупна временска сложеност овог приступа је $O(n^2)$, јер је након попуњавања матрице потребно још само $O(n)$ операција. Пошто није потребно чувати целу матрицу, већ само један текући ред, меморијска сложеност је само $O(n)$.

```
// za svaki broj iz [0, maxZbir] odredjujemo broj nacina da se taj
// broj predstavi kao zbir brojCifara cifara (pri cemu se u obzir
// uzima i redosled cifara)
vector<long long> brojParticija(int brojCifara, int maxZbir) {
  // resenje gradimo dinamickim programiranje
  vector<long long> dp(maxZbir + 1);

  // brojevi izmedju 0 i 9 se mogu na jedan nacin predstaviti kao zbir
  // jedne cifre, a veci brojevi od toga se ne mogu predstaviti
  for (int zbir = 0; zbir <= maxZbir; zbir++)
    dp[zbir] = zbir < 10 ? 1 : 0;

  // odredjujemo broj razlaganja na vise cifara
  for (int cifara = 2; cifara <= brojCifara; cifara++) {
    // zbir moramo da obilazimo unazad da bismo mogli da koristimo
    // samo jedan niz
    for (int zbirCifara = maxZbir; zbirCifara >= 0; zbirCifara--) {
      // na prvo mesto postavljamo redom jednu po jednu cifru
      for (int cifra = 1; cifra <= 9 && cifra <= zbirCifara; cifra++)
        // rekurzivno racunamo broj particija preostalog zbira sa jednom
        // cifrom manje
        dp[zbirCifara] += dp[zbirCifara - cifra];
```

```
    }
  }
  return dp;
}

// izracunava broj 2n-tocifrenih brojeva kod kojih je zbir prvih n i
// zbir drugih n cifara jednak
long long brojBrojeva(int n) {
  // za svaki broj od 0 do 9*n nas zanima koliko postoji razlicitih
  // n-tocifrenih brojeva ciji je zbir cifara taj broj
  vector<long long> bp = brojParticija(n, 9*n);

  // ukupan broj brojeva cija leva i desna polovina imaju isti broj cifara
  long long ukupnoBrojeva = 0;
  // za svaki moguci zbir cifara polovine broja
  for (int zbirCifara = 0; zbirCifara <= 9*n; zbirCifara++) {
    // postoji bp[zbirCifara] nacina da napravimo levu polovinu i
    // bp[zbirCifara] nacina da napravimo desnu polovinu broja
    // tj. bp[zbirCifara]*bp[zbirCifara] nacina da odaberemo broj kome
    // i leva i desna polovina imaju zbir cifara zbirCifara
    ukupnoBrojeva += bp[zbirCifara] * bp[zbirCifara];
  }
  return ukupnoBrojeva;
}
```

Задатак: Број цик-цак партиција

Напиши програм који одређује број партиција природног броја n (разбијања на сабирке који су позитивни природни бројеви) таквих да сабирци наизменично расту и опадају (или опадају па расту).

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 5000$), а затим и број a_0 ($1 \leq a_0 \leq n$), који представља први сабирак у партицији.

Излаз: На стандардни излаз исписати остатак при дељењу броја партиција са $10^9 + 7$.

Пример 1

Улаз	*Излаз*
5 2	2

Објашњење

Партције су $2 + 1 + 2$ и $2 + 3$.

Пример 2

Улаз

```
7
3
```

Излаз

```
3
```

Објашњење

Партције су $3 + 1 + 2 + 1$, $3 + 1 + 3$ и $3 + 4$.

Пример 3

Улаз

```
1000
500
```

Излаз

```
562222907
```

Решење

Рекурзивна формулација

Бројање партиција се може вршити рекурзивном функцијом. Након постављања вредности одређеног сабирка, рекурзивно се партиционише преостали збир (који се добије умањивањем тренутног збира за постављени сабирак). Наравно, овај поступак је потребно модификовати, тако да се броје само цик-цак партиције (оне у којима сабирци наизменично расту и опадају, тј. опадају и расту).

Пошто је први сабирак фиксиран, можемо размотрити све могућности за други сабирак. Пошто су допуштене и партиције које прво расту, па онда опадају и партиције које прво опадају, па онда расту, други сабирак може бити већи, било мањи од првог, међутим, већ након постављања другог сабирка, за сваки наредни сабирак је једнозначно одређено да ли мора да буде мањи или већи од њему претходног сабирка. Зато ће наша рекурзивна функција поред збира добијати и први сабирак и податак о томе да ли наредни сабирак мора да буде мањи или већи од првог. У почетку вршимо два рекурзивна позива, један који израчунава број партиција када је други сабирак већи, а други када је други сабирак мањи од првог, и коначан резултат добијамо сабирањем њихових резултата (изузетак је случај када је први сабирак у старту једнак збиру, када знамо да постоји само једна, тривијална, партиција).

Излаз из рекурзије представља случај када је збир једнак првом сабирку и тада постоји тачно једна таква партиција. У супротном анализирамо све могућности за други сабирак и вршимо даље рекурзивне позиве. Ако други сабирак треба да буде већи од првог, тада разматрамо вредности из интервала $[p+1, z-p]$, где је p вредност првог сабирка, а z вредност збира. Ако други сабирак треба да буде мањи од првог, тада разматрамо вредности из интервала $[1, \min(p-1, z-p)]$. Сабирамо бројеве партиција за сваку вредност другог сабирка (по задатом модулу) и враћамо збир.

Дакле, ако обележимо са $r_{z,p}$ број партиција збира z, где је први сабирак p, а други већи од њега, а са $o_{z,p}$ број партиција збира z, где је први сабирак p, а други мањи од њега, важе следеће рекурентне везе.

$$\begin{aligned} r_{z,z} &= 1 \\ o_{z,z} &= 1 \\ r_{z,p} &= \sum_{d=p+1}^{z-p} o_{z-p,d}, \quad \text{за } p < z \\ o_{z,p} &= \sum_{d=1}^{\min(p-1,z-p)} r_{z-p,d}, \quad \text{за } p < z \end{aligned}$$

Наравно, све ове збирове треба рачунати по датом модулу.

```
const int MOD = 1e9 + 7;

int brojCikCakParticija(int prvi, int zbir, bool raste) {
  if (zbir == prvi)
    return 1;
  int broj = 0;
  if (raste)
    for (int sledeci = prvi+1; sledeci <= zbir - prvi; sledeci++)
      broj = (broj + brojCikCakParticija(sledeci, zbir - prvi, false)) % MOD;
  else
    for (int sledeci = 1; sledeci < prvi && sledeci <= zbir - prvi; sledeci++)
      broj = (broj + brojCikCakParticija(sledeci, zbir - prvi, true)) % MOD;
  return broj;
}

int brojCikCakParticija(int prvi, int zbir) {
```

```
  if (zbir == prvi)
    return 1;

  return (brojCikCakParticija(prvi, zbir, true) +
          brojCikCakParticija(prvi, zbir, false)) % MOD;
}
```

Могуће је дефинисати и две узајамно рекурзивне функције.

```
const int MOD = 1e9 + 7;

int brojCikCakParticijaRaste(int prvi, int zbir);
int brojCikCakParticijaOpada(int prvi, int zbir);

int brojCikCakParticijaRaste(int prvi, int zbir) {
  if (zbir == prvi)
    return 1;
  int broj = 0;
  for (int sledeci = prvi+1; sledeci <= zbir - prvi; sledeci++)
    broj = (broj + brojCikCakParticijaOpada(sledeci, zbir - prvi)) % MOD;
  return broj;
}

int brojCikCakParticijaOpada(int prvi, int zbir) {
  if (zbir == prvi)
    return 1;
  int broj = 0;
  for (int sledeci = 1; sledeci < prvi && sledeci <= zbir - prvi; sledeci++)
    broj = (broj + brojCikCakParticijaRaste(sledeci, zbir - prvi)) % MOD;
  return broj;
}


int brojCikCakParticija(int prvi, int zbir) {
  if (zbir == prvi)
    return 1;

  return (brojCikCakParticijaRaste(prvi, zbir) +
          brojCikCakParticijaOpada(prvi, zbir)) % MOD;
}
```

Мемоизација

Пошто се у рекурзивној формулацији често врше идентични рекурзивни позиви, ефикасност се може поправити динамичким програмирањем. На пример, могуће је применити мемоизацију. За мемоизацију можемо употребити две матрице (једну за вредности $r_{z,p}$, а другу за вредности $o_{z,p}$, а можемо и те две матрице спаковати у тродимензионални низ (чија је последња димензија једнака 2, тако да нпр. индекс 0 одређује вредности $r_{z,p}$, а индекс 1 одређује вредности $o_{z,p}$).

Анализа сложености. Сложеност се грубо може проценити на следећи начин. Вредности у матрици се рачунају у линеарној сложености, па пошто је димензија матрице квадратна (у односу на полазни збир z), сложеност је $O(z^3)$.

```
const int MOD = 1e9 + 7;

int brojCikCakParticija(int prvi, int zbir, bool raste,
                        vector<vector<array<int, 2>>>& memo) {
  if (memo[prvi][zbir][raste] != -1)
    return memo[prvi][zbir][raste];
  if (zbir == prvi)
    return memo[prvi][zbir][raste] = 1;
```

```
  int broj = 0;
  if (raste)
    for (int sledeci = prvi+1; sledeci <= zbir - prvi; sledeci++)
      broj = (broj + brojCikCakParticija(sledeci, zbir - prvi, false, memo)) % MOD;
  else
    for (int sledeci = 1; sledeci < prvi && sledeci <= zbir - prvi; sledeci++)
      broj = (broj + brojCikCakParticija(sledeci, zbir - prvi, true, memo)) % MOD;
  return memo[prvi][zbir][raste] = broj;
}

int brojCikCakParticija(int prvi, int zbir) {
  if (prvi == zbir)
    return 1;

  vector<vector<array<int, 2>>> memo(zbir + 1, vector<array<int, 2>>(zbir + 1, {-1, -1}));
  return (brojCikCakParticija(prvi, zbir, true, memo) +
          brojCikCakParticija(prvi, zbir, false, memo)) % MOD;
}
```

Динамичко програмирање навише

Рекурзивна конструкција се може преточити у индуктивну и задатак се може решити динамичким програмирањем навише. Матрице можемо попуњавати врсту по врсту, али услед узајамне зависности вредности у матрицама, чим попунимо једну врсту у матрици $r_{z,p}$, одмах ту врсту морамо попунити у матрици $o_{z,p}$. Приликом попуњавања вредности у текућој врсти имамо на располагању све потребне вредности. Наиме $r_{z,p}$ зависи од вредности облика $o_{z-p,k}$, што је већ израчунато, јер је $p \geq 1$. Слично је и за $o_{z,p}$.

Анализа сложености. Сложеност попуњавања матрица је $O(z^3)$, где је z почетна вредност збира.

```
const int MOD = 1e9 + 7;

int brojCikCakParticija(int prvi, int zbir) {
  if (prvi == zbir)
    return 1;

  vector<vector<int>> raste(zbir+1, vector<int>(zbir+1, 0));
  vector<vector<int>> opada(zbir+1, vector<int>(zbir+1, 0));

  for (int z = 1; z <= zbir; z++) {
    for (int p = 1; p < z; p++) {
      raste[z][p] = 0;
      for (int k = p+1; k <= z-p; k++)
        raste[z][p] = (raste[z][p] + opada[z-p][k]) % MOD;
      opada[z][p] = 0;
      for (int k = 1; k <= p-1 && k <= z-p; k++)
        opada[z][p] = (opada[z][p] + raste[z-p][k]) % MOD;
    }
    raste[z][z] = opada[z][z] = 1;
  }

  return (raste[zbir][prvi] + opada[zbir][prvi]) % MOD;
}
```

Оптимизација коришћењем префиксних збирова

Приметимо да се вредности $r_{z,p}$ и $o_{z,p}$ израчунавају као збир одређених сегмената (узастопних елемената) претходних врста матрице. Збирови елемената сегмената могу израчунати ефикасно ако се израчунају збирови префикса низа. Та је техника детаљно објашњена у задатку Збирови сегмената. Збирове префикса је могуће применити и у овом задатку и уместо вредности $r_{z,p}$ можемо чувати збир префикса $R_{z,p} = \sum_{k=1}^{p} r_{z,k}$ и слично уместо вредности $o_{z,p}$ можемо чувати вредности $O_{z,p} = \sum_{k=1}^{p} o_{z,k}$. Тада за свако $p < z$ важи:

$$\begin{aligned} r_{z,p} &= \sum_{d=p+1}^{z-p} o_{z-p,d} = O_{z-p,z-p} - O_{z-p,p} \\ R_{z,p} &= R_{z,p-1} + r_{z,p} \\ o_{z,p} &= \sum_{d=1}^{\min(p-1,z-p)} r_{z-p,d} = R_{z-p,\min(p-1,z-p)} \\ O_{z,p} &= O_{z,p-1} + o_{z,p} \end{aligned}$$

Такође, пошто је $r_{z,z} = o_{z,z} = 1$, важи

$$\begin{aligned} R_{z,z} &= R_{z,z-1} + 1 \\ O_{z,z} &= O_{z,z-1} + 1 \end{aligned}$$

Наравно, све ове збирове треба рачунати по датом модулу.

На крају, у случају да је $p < z$, вредности $r_{z,p}$ и $o_{z,p}$ се израчунавају као $O_{z,p} - O_{z,p-1}$ и $R_{z,p} - R_{z,p-1}$ (у супротном, ако је $z = p$, знамо да је резултат 1 и без икаквог рачунања).

Анализа сложености. Пошто се сада свака вредност у матрици рачуна у времену $O(1)$, а у матрици постоји $O(z^2)$ елемената, сложеност алгоритма је $O(z^2)$.

```
const int MOD = 1e9 + 7;

int brojCikCakParticija(int prvi, int zbir) {
  if (prvi == zbir)
    return 1;

  vector<vector<int>> zbirRaste(zbir+1);
  for (int i = 0; i <= zbir; i++)
    zbirRaste[i].resize(zbir+1, 0);
  vector<vector<int>> zbirOpada(zbir+1);
  for (int i = 0; i <= zbir; i++)
    zbirOpada[i].resize(zbir+1, 0);

  for (int z = 1; z <= zbir; z++) {
    for (int p = 1; p < z; p++) {
      if (p < z-p) {
        int raste_zp = (zbirOpada[z-p][z-p] - zbirOpada[z-p][p] + MOD) % MOD;
        zbirRaste[z][p] = (zbirRaste[z][p-1] + raste_zp) % MOD;
      } else
        zbirRaste[z][p] = zbirRaste[z][p-1];

      int opada_zp = zbirRaste[z-p][min(p-1, z-p)];
      zbirOpada[z][p] = (zbirOpada[z][p-1] + opada_zp) % MOD;
    }
    zbirRaste[z][z] = (zbirRaste[z][z-1] + 1) % MOD;
    zbirOpada[z][z] = (zbirOpada[z][z-1] + 1) % MOD;
  }
  int raste = (zbirRaste[zbir][prvi] - zbirRaste[zbir][prvi-1] + MOD) % MOD;
  int opada = (zbirOpada[zbir][prvi] - zbirOpada[zbir][prvi-1] + MOD) % MOD;
  return (raste + opada) % MOD;
}
```

Задатак: Број појављивања подниске

Написати програм којим се за две дате ниске x и y, одређује број појављивања ниске y као подниза ниске x. Ниска y је подниз ниске x ако се може добити од ниске x брисањем произвољног броја карактера.

Улаз: Прва линија стандардног улаза садржи ниску x, а друга линија ниску y. Дужине ниски су највише 100 карактера.

Излаз: На стандардном излазу приказати само број појављивања ниске y као подниза ниске x.

Пример

Улаз	*Излаз*
abcbca	3
abc	

Објашњење:

Позиције појављивања подниза су обележене великим словима.

ABCbca
ABcbCa
AbcBCa

Решење

Размотримо прво рекурзивно решење.

- Ако су обе ниске непразне, тада се може упоредити њихово последње слово. Ако су им последња слова различита онда је укупан број појављивања подниске једнак броју њених појављивања у префиксу прве ниске без последњег карактера, а ако су им последња слова једнака, онда је укупан број појављивања једнак том броју увећаном за број појављивања префикса друге ниске без последњег карактера у префиксу прве ниске без последњег карактера.
- Базу чине случајеви када је нека ниска празна. Ако је друга ниска празна тада се онда једном јавља као подниз прве ниске, а ако је прва ниска празна (а друга није) тада она не садржи подниз који одговара другој ниски.

Пошто се кроз рекурзију стално разматрају префикси ниски, ниске не морамо мењати кроз рекурзију, већ можемо прослеђивати само дужине префикса (нека је n дужина префикса прве, а m дужина префикса друге ниске). Ако је $f(n, m)$ тражени број појављивања, тада важи:

$$
\begin{aligned}
f(n,0) &= 1 \\
f(0,m) &= 0, \quad \text{za } m > 0 \\
f(n,m) &= f(n-1,m) + f(n-1,m-1), \quad \text{za } n,m > 0 \ \ a_n = a_m \\
f(n,m) &= f(n-1,m), \quad \text{za } n,m > 0 \ \ a_n \neq a_m
\end{aligned}
$$

На основу овога је веома једноставно направити рекурзивну имплементацију.

```
long long brojPojavljivanjaPodniske(const string& niska, int duzina_niske,
                                    const string& podniska, int duzina_podniske) {
  if (duzina_podniske == 0)
    return 1;
  if (duzina_niske == 0)
    return 0;
  long long broj = brojPojavljivanjaPodniske(niska, duzina_niske-1,
                                             podniska, duzina_podniske);
  if (niska[duzina_niske-1] == podniska[duzina_podniske-1])
    broj += brojPojavljivanjaPodniske(niska, duzina_niske-1,
                                      podniska, duzina_podniske-1);
  return broj;
}

long long brojPojavljivanjaPodniske(const string& niska,
                                    const string& podniska) {
  return brojPojavljivanjaPodniske(niska, niska.length(), podniska, podniska.length());
}
```

С обзиром на то да ће се у наивној рекурзивној имплементацији исти рекурзивни позиви извршавати више пута, таква имплементација биће неефикасна. Ефикасност се лако може поправити техником динамичког програмирања (било мемоизацијом, било динамичким програмирањем навише). Пошто функција има два параметра, вредности рекурзивних позива можемо памтити у матрици.

Пример. Прикажимо ову матрицу на примеру бројања појављивања подниске `abc` унутар ниске `abcbca`.

```
    0  1  2  3
  | -----------
0 | 1  0  0  0
1 | 1  1  0  0
2 | 1  1  1  0
3 | 1  1  1  1
4 | 1  1  2  1
5 | 1  1  2  3
6 | 1  2  2  3
```

Пошто елемент на позицији (n, m) зависи од елемената на позицијама $(n-1, m)$ и $(n-1, m-1)$ матрицу можемо попуњавати било врсту по врсту, било колону по колону.

```
long long brojPojavljivanjaPodniskaa(const string& niska,
                                     const string& podniska) {
  int n = niska.length();
  int m = podniska.length();
  vector<vector<long long>> dp(n + 1);
  for (int i = 0; i <= n; i++)
    dp[i].resize(m + 1);

  for (int i = 0; i <= n; i++)
    dp[i][0] = 1;
  for (int j = 1; j <= m; j++)
    dp[0][j] = 0;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++) {
      dp[i][j] = dp[i-1][j];
      if (niska[i-1] == podniska[j-1])
        dp[i][j] += dp[i-1][j-1];
    }
  return dp[n][m];
}
```

Ако претпоставимо да попуњавамо врсту по врсту матрице можемо извршити меморијску оптимизацију тако што само чувамо елементе једне врсте. У првој врсти иницијализујемо све елементе на нулу, осим првог који иницијализујемо на јединицу. Приликом преласка са текуће на наредну врсту, ажурирање можемо вршити с десна на лево. Ако су одговарајући карактери ниски једнаки, тада текући елемент увећавамо за њему претходни.

```
long long brojPojavljivanjaPodniske(const string& niska,
                                    const string& podniska) {
  int n = niska.length();
  int m = podniska.length();
  vector<long long> dp(m + 1, 0);
  dp[0] = 1;
  for (int i = 1; i <= n; i++)
    for (int j = m; j >= 1; j--)
      if (niska[i-1] == podniska[j-1])
        dp[j] += dp[j-1];
  return dp[m];
}
```

8.3 Оптимизација коришћењем динамичког програмирања

Динамичко програмирање се често примењује у решавању оптимизационих задатака (задатака у којима се тражи да се одреди најмања или највећа вредност која задовољава неки услов). Да би оптимизациони задатак могао да се решава динамичким програмирањем, потребно је формулисати његово рекурзивно решење, што значи да проблем мора да задовољава такозвано својство **оптималне подструктуре**. То значи да се оптимално решење полазног проблема може одредити на основу оптималних решења потпроблема истог облика, али мање димензије.

На пример, најкраћи пут између тачке A и тачке B који пролази преко тачке C се добија тако што се искомбинују најкраћи пут између тачке A и тачке C и накраћи пут између тачке C и тачке B, што значи да проблем одређивања најкраћих путева има својство оптималне подструктуре и може се решавати динамичким програмирањем (на тој техници је заснован Дајкстрин алгоритам, који је један од најчувенијих алгоритама за решвање тог проблема).

Међутим, најјефтинији авионски лет од аеродрома A до аеродрома B преко аеродрома C не мора да буде комбинација најјефтинијих летова од A до C и од C до B (због посебних попуста које авио-компаније нуде за повезане летове), тако да се проблем одређивања најјефтинијег лета нема својство оптималне подструктуре и не може се решавати динамичким програмирањем.

Задатак: Максимални збир на путу кроз матрицу

У табели димензија $n \times n$ поља су попуњена цифрама од 0 до 9. Играч који се налази у горњем левом углу табеле може да у једном кораку пређе у суседно десно поље или суседно доње поље. Циљ му је да стигне до доњег десног поља тако да збир вредности на пређеним пољима буде максималан. Написати програм којим се одређује максималан збир који може да оствари играч при кретању од горњег левог до доњег десног угла.

Улаз: У првој линији стандардног улаза се уноси број редова табеле n ($1 \leq n \leq 30$), а у следећих n редова по n цифара од 0 до 9, раздвојене по једним размаком.

Излаз: У првој линији стандардног излаза приказати тражену вредност максималног збира.

Пример

Улаз	*Излаз*
5 4 3 5 7 5 1 9 4 1 3 2 3 5 1 2 1 3 1 2 0 4 6 7 2 1	38

Објашњење

Максималан збир се добија када се редом пролазе поља са вредношћу 4 + 3 + 9 + 3 + 3 + 6 + 7 + 2 + 1.

Решење

Рекурзивно решење

Покушајмо прво да дефинишемо рекурзивно решење. Дефинишемо рекурзивну функцију која одређује максимални збир вредности који се може постићи крећући се од поља $(0, 0)$ до датог поља (v, k), које је параметар функције. Тражени резултат добијамо када је (v, k) једнако $(n - 1, n - 1)$.

Излаз из рекурзије представља случај када је $(v, k) = (0, 0)$ – постоји само једна путања од поља $(0, 0)$ до њега самог (када се не померамо) и збир вредности на њој је једнак вредности на пољу $(0, 0)$.

У супротном посматрајмо неко поље (v, k) различито од $(0, 0)$. На то смо поље могли доћи или са поља $(v - 1, k)$ или са поља $(v, k - 1)$, при чему први случај није могућ када се поље налази у почетној врсти (када је $v = 0$), а други није могућ када се поље налази у почетној колони (када је $k = 0$). Да би се постигао оптимални збир до поља (v, k), мора се постићи оптимални збир до поља са ког смо на њега прешли. Заиста ако би се до претходног поља могло доћи путањом са већим збиром, тада бисмо ту путању могли проширити последњим преласком на поље (v, k) и добити већу вредност збира на путањи до поља (v, k). Зато вредност одређујемо додајући елемент матрице на позицији (v, k) на вредности рекурзивних позива за поља $(v - 1, k)$

и $(v, k-1)$ (водивши рачуна да се прескочи неодговарајући рекурзивни позив у случајевима када је $v = 0$ тј. $k = 0$) и бирајући повољнију од те две варијанте (ону која даје већи збир).

Ако са $z(v, k)$ обележимо вредност максималног збира до поља (v, k), тада важи следеће:

$$z(v,k) = \begin{cases} 0 & \text{ако је } v = 0 \text{ и } k = 0 \\ z(v, k-1) & \text{ако је } v = 0 \text{ и } k > 0 \\ z(v-1, k) & \text{ако је } v > 0 \text{ и } k = 0 \\ \max(z(v-1,k), z(v,k-1)) & \text{ако је } v > 0 \text{ и } k > 0 \end{cases}$$

Овај проблем поседује својство оптималне подструктуре. Рекурзивно решење овог оптимизационог проблема је било могуће захваљујући томе што овај проблем задовољава принцип оптималности, по коме оптимално решење проблема мора садржати и оптимално решење сваког свог потпроблема. У овом проблему то својство је испуњено јер оптимална путања, која представља решење проблема, мора бити оптимална на сваком делу пута.

Пример. Ради илустровања проблема размотримо квадратну матрицу A следећег садржаја:

```
4   3   5   7   5
1   9   4   1   3
2   3   5   1   2
1   3   1   2   0
4   6   7   2   1
```

Вредност највећег збира до сваког од поља може се израчунати на основу претходне рекурзивне дефиниције и може се представити следећом матрицом.

```
4   7 12 19 24
5  16 20 21 27
7  19 25 26 29
8  22 26 28 29
12 28 35 37 38
```

```
int maksZbir(const vector<vector<int>>& M, int n, int v, int k) {
    if (v == 0 && k == 0)
        return M[v][k];
    int odGore = 0, sLeva = 0;
    if (v > 0)
       odGore = M[v][k] + maksZbir(M, n, v-1, k);
    if (k > 0)
       sLeva = M[v][k] + maksZbir(M, n, v, k-1);
    if (v == 0) return sLeva;
    if (k == 0) return odGore;
    return max(odGore, sLeva);
}

int maksZbir(const vector<vector<int>>& M) {
  int n = M.size();
  return maksZbir(M, n, n-1, n-1);
}
```

Анализа сложености. У рекурзивној имплементацији долази до понављања истих рекурзивних позива (на пример, ако направимо кораке `dole-desno` и `desno-dole`, наћи ћемо се на истом пољу). Стога је претходна имплементација изразито неефикасна (сложеност најгорег случаја је експоненцијална).

Могуће је дати и дуалну рекурзивну дефиницију, којом се израчунава оптимални пут од датог поља (v, k) до крајњег поља $(n-1, n-1)$. Излаз из рекурзије је када је $(v, k) = (n-1, n-1)$, док у рекурзивним корацима анализирамо два поља до којих можемо доћи са текућег (то су оно десно и доле од њега) и за њих рекурзивно позивамо функцију.

```
int maksZbir(const vector<vector<int>>& M, int n, int v, int k) {
  if (v >= n-1 && k >= n-1)
```

```
    return M[v][k];
  int dole, desno;
  if (v < n - 1)
    dole = M[v][k] + maksZbir(M, n, v+1, k);
  if (k < n - 1)
    desno = M[v][k] + maksZbir(M, n, v, k+1);
  if (v >= n-1) return desno;
  if (k >= n-1) return dole;
  return max(dole, desno);
}

int maksZbir(const vector<vector<int>>& M) {
  return maksZbir(M, M.size(), 0, 0);
}
```

Мемоизација

Решење директном рекурзијом је неефикасно и потребно је извршити оптимизацију техником динамичког програмирања. Рекурзивну функцију је могуће проширити коришћењем мемоизације. Резултат за свако поље ћемо памтити у матрици.

У језику C++ матрицу можемо представити помоћу вектора који садржи векторе и који се динамички алоцира током рада програма, када је позната димензија учитане матрице. Ово решење је елегантно и флексибилно, али се мора напоменути да се одређено време губи на динамичку алокацију, тако да се мало бржи програм може добити ако би матрица била унапред статички алоцирана (по цену, већег губитка меморије у случајевима када је димензија учитане матрице мања од максималне димензије допуштене текстом задатка).

Анализа сложености. За сваки пар бројева (v, k), такав да је $0 \leq v, k < n$, израчунавање се врши само једном, а затим се израчуната вредност очитава из матрице. Меморијска и временска сложеност овог решења су стога $O(n^2)$.

```
int maksZbir(const vector<vector<int>>& M, int n, int v, int k,
             vector<vector<int>>& memo) {
  if (memo[v][k] != -1)
    return memo[v][k];

  if (v == 0 && k == 0)
    return memo[v][k] = M[v][k];
  int odGore = 0, sLeva = 0;
  if (v > 0)
    odGore = M[v][k] + maksZbir(M, n, v-1, k, memo);
  if (k > 0)
    sLeva = M[v][k] + maksZbir(M, n, v, k-1, memo);
  if (v == 0) return memo[v][k] = sLeva;
  if (k == 0) return memo[v][k] = odGore;
  return memo[v][k] = max(odGore, sLeva);
}

int maksZbir(const vector<vector<int>>& M) {
  int n = M.size();
  vector<vector<int>> memo(n, vector<int>(n, -1));
  return maksZbir(M, n, n-1, n-1, memo);
}
```

Динамичко програмирање навише

Уместо мемоизације можемо употребити и динамичко програмирање навише. Резултат рекурзивне функције која израчунава максималну вредност збира од поља $(0, 0)$ до поља (v, k), памтићемо у помоћној матрици на њеном пољу (v, k). Пошто елемент у матрици може да зависи од елемента изнад и лево од себе, матрицу можемо попуњавати врсту по врсту (а могуће би било попуњавати је и колону по колону). Вредност на пољу

$(0, 0)$ преписаћемо из полазне матрице, остале вредности у првој врсти ћемо добити увећавањем вредности лево од њих одговарајућим елементом полазне матрице, а остале вредности у првој колони ћемо добити увећавањем вредности изнад њих одговарајућим елементом полазне матрице. Остале вредности у матрици ћемо добити проналажењем већег од елемента изнад и лево од њих и увећавањем тог већег броја за одговарајућу вредност у полазној матрици.

```
int maksZbir(const vector<vector<int>>& M) {
  int n = M.size();
  vector<vector<int>> dp(n);
  for (int i = 0; i < n; i++)
    dp[i].resize(n);
  dp[0][0] = M[0][0];
  for (int v = 1; v < n; v++)
    dp[v][0] = dp[v-1][0] + M[v][0];
  for (int k = 1; k < n; k++)
    dp[0][k] = dp[0][k-1] + M[0][k];
  for (int v = 1; v < n; v++)
    for (int k = 1; k < n; k++)
      dp[v][k] = max(dp[v-1][k] + M[v][k], dp[v][k-1] + M[v][k]);

  return dp[n-1][n-1];
}
```

Меморијска оптимизација

Можемо приметити да елементи у свакој врсти матрице када се користи динамичко програмирање навише зависе само од вредности у претходној врсти (а не од вредности у врстама пре ње). То нам омогућава да направимо додатну оптимизацију тако што ћемо уместо матрице за динамичко програмирање користити само један помоћни низ (у ком ћемо памтити елементе претходне врсте, а затим их један по један мењати елементима текуће врсте).

Анализа сложености. Временска сложеност овог решења остаје $O(n^2)$, али меморијска сложеност се смањује на $O(n)$.

```
int maksZbir(const vector<vector<int>>& M) {
  int n = M.size();
  vector<int> dp(n);
  dp[0] = M[0][0];
  for (int k = 1; k < n; k++)
    dp[k] = dp[k-1] + M[0][k];
  for (int v = 1; v < n; v++) {
    dp[0] += M[v][0];
    for (int k = 1; k < n; k++)
      dp[k] = max(dp[k] + M[v][k], dp[k-1] + M[v][k]);
  }

  return dp[n-1];
}
```

Задатак: Максимални пут кроз матрицу

У табели димензија $n \times n$ поља су попуњена цифрама од 0 до 9. Играч који се налази у горњем левом углу табеле може да у једном кораку пређе у суседно десно поље или суседно доње поље. Циљ му је да стигне до доњег десног поља тако да збир вредности на пређеним пољима буде максималан. Написати програм којим се одређује максимални збир коју може остварити играч при кретању од горњег левог до доњег десног угла и инструкције кретања: `desno`, `dole`, којима се обезбеђује кретање преко поља која дају максимални збир (ако има више могућих путева, исписати било који).

Улаз: У првој линији стандардног улаза се уноси број редова табеле n ($1 \leq n \leq 30$), а у следећих n редова по n цифара од 0 до 9.

Излаз: У првој линији стандардног излаза приказати тражену вредност максималног збира, а у наредним линијама исписати инструкције за кретање: `desno`, `dole` - у сваком реду по једну, којима се обезбеђује кретање преко поља која дају максимални збир.

Пример

Улаз	*Излаз*
5 4 3 5 7 5 1 9 4 1 3 2 3 5 1 2 1 3 1 2 0 4 6 7 2 1	38 desno dole dole dole dole desno desno desno

Решење

Овај задатак представља проширење задатка Максимални збир на путу кроз матрицу. Након одређивања оптималног пута, потребно је реконструисати и сам пут. Иако је понекад потребно складиштити и додатне информације да би се од вредности решења могло реконструисати решење, овде то није случај - решење је у потпуности могуће реконструисати на основу матрице изграђене у склопу динамичког програмирања навише (или у склопу мемоизације). Ипак, нагласимо да нам је потребна цела матрица и да није могуће извршити оптимизацију у којој се чува само текућа врста матрице.

Током реконструкције решења крећемо уназад, од завршног поља па све до почетног и на основу вредности у матрици закључујемо са ког претходног поља се стигло на текуће. Анализирамо поље лево и поље изнад текућег и гледамо на ком од њих пише већа вредност (ако су вредности једнаке, свеједно је које ћемо поље одабрати). Када знамо са ког смо поља стигли на текуће, знамо и на основу које инструкције робот прави тај корак (ако смо дошли са поља лево, инструкција је `desno`, а ако смо дошли са поља изнад, инструкција је `dole`). Потребно је само да посебно обратимо пажњу на поља у првој врсти и првој колони, јер код њих постоји само једна могућност. Крај реконструкције наступа када дођемо до поља $(0, 0)$. Приметимо да на овај начин, крећући се од завршног ка полазном пољу одређујемо низ инструкција унатраг, у обратном редоследу. Да бисмо добили инструкције у редоследу који је захтеван, можемо употребити рекурзивну функцију, тако што прво рекурзивно испишемо све инструкције за долазак на претходно поље и тек након тога испишемо инструкцију за прелазак са претходног на текуће поље.

```
// odredjujemo matricu koja na polju (v, k) sadrzi duzinu
// najkraceg puta od (0, 0) do (v, k)
vector<vector<int>> maksZbirDP(const vector<vector<int>>& M) {
  int n = M.size();
  vector<vector<int>> dp(n);
  for (int i = 0; i < n; i++)
    dp[i].resize(n);
  dp[0][0] = M[0][0];
  for (int v = 1; v < n; v++)
    dp[v][0] = dp[v-1][0] + M[v][0];
  for (int k = 1; k < n; k++)
    dp[0][k] = dp[0][k-1] + M[0][k];
  for (int v = 1; v < n; v++)
    for (int k = 1; k < n; k++)
      dp[v][k] = max(dp[v-1][k] + M[v][k], dp[v][k-1] + M[v][k]);
  return dp;
}

// ispisuje instrukcije kretanja od polja (0, 0) do polja (v, k)
void pisiPut(int v, int k, const vector<vector<int>>& dp) {
  // ako smo vec na polju (0, 0), nema potrebe da se pomeramo
  if (v == 0 && k == 0)
    return;
```

```
  if (v == 0) {
    // na polja u prvoj vrsti mozemo stici samo sa polja levo od njih,
    // pomerajuci se desno
    pisiPut(v, k-1, dp);
    cout << "desno" << endl;
  } else if (k == 0) {
    // na polja u prvoj koloni mozemo stici samo sa polja iznad njih,
    // pomerajuci se na dole
    pisiPut(v-1, k, dp);
    cout << "dole" << endl;
  } else
    // u suprotnom analiziramo da li nam je povoljnije da stignemo sa
    // polja iznad ili sa polja levo
    if (dp[v-1][k] > dp[v][k-1]) {
      pisiPut(v-1, k, dp);
      cout << "dole" << endl;
    } else {
      pisiPut(v, k-1, dp);
      cout << "desno" << endl;
    }
}
```

Уместо рекурзије можемо употребити и стек. Током пута уназад инструкције за прелазак са претходног на текуће поље ћемо постављати на помоћни стек, а затим, када стигнемо до поља $(0, 0)$, скидаћемо једну по једну вредност са стека и исписиваћемо је.

```
// ispisuje instrukcije kretanja
void pisiPut(const vector<vector<int>>& dp, int n) {
  // instrukcije dobijamo u obratnom redosledu, pa ih obrcemo
  // koriscenjem pomocnog steka
  stack<string> put;

  // krecemo od kraja i idemo ka pocetku
  int v = n-1, k = n-1;
  while (v > 0 || k > 0) {
    if (v == 0) {
      // na polja u prvoj vrsti mozemo stici samo sa polja levo od njih,
      // pomerajuci se desno
      k--;
      put.push("desno");
    } else if (k == 0) {
      // na polja u prvoj koloni mozemo stici samo sa polja iznad njih,
      // pomerajuci se na dole
      v--;
      put.push("dole");
    } else
      // u suprotnom analiziramo da li nam je povoljnije da stignemo sa
      // polja iznad ili sa polja levo
      if (dp[v-1][k] > dp[v][k-1]) {
        v--;
        put.push("dole");
      } else {
        k--;
        put.push("desno");
      }
  }

  // sadrzaj steka ispisujemo u obratnom redosledu
  while (!put.empty()) {
```

```
    cout << put.top() << endl;
    put.pop();
  }
}
```

Задатак: Исплата са најмање новчића

Дате су вредности n врста новчића. Написати програм који одређује минималан број новчића потребних за исплату датог износа S, при чему се може се користити и више новчића исте врсте и сваке врсте новчића има произвољно много. Вредности новчића и износ за исплату дати су у истој валути.

Улаз: Прва линија стандардног улаза задржи природан број S ($S \leq 1000$). Друга линија садржи природан број n ($n \leq 100$), у следећих n линија налазе се природни бројеви који представљају вредности за сваку врсту новчића, свака вредност у посебној линији.

Излаз: На стандардном излазу приказати у једној линији минималан број новчића потребан sa исплату износа S.

Пример 1

Улаз	*Излаз*
7	3
3	
1	
3	
2	

Објашњење: исплата за износ 7 са најмање новчића је $3, 3, 1$.

Пример 2

Улаз

```
12
3
1
9
6
```

Излаз

```
2
```

Објашњење: исплата за износ 12 са најмање новчића је $6, 6$.

Решење

Анализа последње врсте новчића

Износ 0 се може наплатити са 0 новчића. У супротном, ако је низ расположивих новчића празан, износ није могуће наплатити. У супротном испитујемо могућност да је у износ укључен новчић последње врсте. Ако није, покушавамо да наплатимо износ без коришћења последње врсте новчића, тј. са префиксном низа новчића без последњег елемента. Ако јесте, тада износ мора бити већи или једнак од новчића последње врсте и тада преостали износ покушавамо да наплатимо поново коришћењем свих врста новчића. Ако са са $f(n, s)$ најмањи број новчића да се наплати износ s помоћу новчића који припадају префиксу дужине n полазног низа, тада важи

$$
\begin{aligned}
&f(n, 0) = 0, \\
&f(0, s) = +\infty, \quad \text{за } s > 0 \\
&f(n, s) = f(n-1, s), \quad \text{за } s < v_{n-1} \\
&f(n, s) = \min\left(f(n-1, s), 1 + f(n, s - v_{n-1})\right), \quad \text{за } s \geq v_{n-1}
\end{aligned}
$$

На основу овога је веома једноставно дефинисати рекурзивну функцију која израчунава тражени најмањи број новчића.

```
const int MAX_S = 2000;
const int MAX_V = 100;
const int INF = MAX_S + 1;

// najmanji broj novčića potreban da se naplati iznos S
// kada su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(int v[], int n, int s) {
  // iznos 0 se plaća sa 0 novčića
  if (s == 0)
    return 0;
  // ako nema novčića pozitivan iznos nije moguće platiti
  if (n == 0)
    return INF;
  // broj novčića ako se ne uzme ni jedan novčić poslednje vrste
  int br = minBrojNovcica(v, n-1, s);
  // ako je iznos veći od novčića poslednje vrste
  if (s >= v[n-1])
    // gledamo da li je bolje ako se uzme jedan novčić poslednje vrste
    br = min(br, minBrojNovcica(v, n, s - v[n-1]) + 1);
  // vraćamo rezultat
  return br;
}
```

Јасно је да у претходној функцији може доћи до понављања истих рекурзивних позива, што се може решити техником динамичког програмирања. Пошто функција има два променљива параметра, могуће је употребити матрицу за мемоизацију.

```
const int MAX_V = 100;
const int MAX_S = 2000;
const int INF = MAX_S + 1;

// matrica koju koristimo za memoizaciju
int memo[MAX_V][MAX_S + 1];

int minBrojNovcica(int v[], int n, int s){
  // ako smo već računali vrednost za (n, s), vraćamo upamćeni rezultat
  if (memo[n][s] != 0)
    return memo[n][s];
  // iznos 0 se naplaćuje sa 0 novčića
  if (s == 0)
    return memo[n][s] = 0;
  // ako nema novčića pozitivan iznos nije moguće platiti
  if (n == 0)
    return memo[n][s] = INF;
  // broj novčića ako se ne uzme ni jedan novčić poslednje vrste
  int br = minBrojNovcica(v, n-1, s);
  // ako je iznos veći od novčića poslednje vrste
  if (s >= v[n-1])
    // gledamo da li je bolje ako se uzme jedan novčić poslednje vrste
    br = min(br, minBrojNovcica(v, n, s - v[n-1]) + 1);
  // vraćamo rezultat
  return memo[n][s] = br;
}
```

Приликом динамичког програмирања навише, матрицу можемо попуњавати врсту по врсту и тако смањити меморијску сложеност.

Анализа сложености. Временска сложеност овог решења је $O(S \cdot N)$, где је S износ, а N број врста новчића, док је меморијска сложеност $O(S)$.

```
const int MAX_V = 100;
```

```
const int MAX_S = 2000;
const int INF = MAX_S + 1;


// najmanji broj novčića potreban da se naplati iznos S
// kada su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(int v[], int N, int S) {
  // najmanji broj novčića potrebnih da se plati svaki iznos od 0 do S
  int dp[MAX_S + 1];

  // analizu pokrećemo sa 0 vrsta novčića

  // iznos 0 se plaća sa 0 novčića
  dp[0] = 0;
  // ostale iznose nije moguće platiti
  for (int s = 1; s <= S; s++)
    dp[s] = INF;

  // uključujemo jednu po jednu vrstu novčića
  for (int n = 1; n <= N; n++) {
    // ažuriramo vrednosti svih iznosa
    for (int s = 0; s <= S; s++) {
      // ako je moguće uključiti novčić tekuće vrste
      if (s >= v[n-1])
        // ažuriramo minimum ako je to potrebno
        dp[s] = min(dp[s], dp[s - v[n-1]] + 1);
    }
  }

  // vraćamo najmanji broj novčića za iznos S
  return dp[S];
}
```

Анализа свих могућности за последњи новчић

Друго могуће решење разматра све могућности за последњи употребљени новчић. То може бити било који новчић који је мањи од текућег износа који треба наплатити, након чега се преостали износ и даље наплаћује помоћу свих могућих новчића. Ако са $f(s)$ обележимо најмањи број новчића потребних да се наплати износ s, при чему се могу користити сви расположиви новчићи, добијамо следећу рекурентну везу.

$$f(0) = 0,$$
$$f(s) = \min_{v_i \leq s} (1 + f(s - v_i))$$

Ако је износ позитиван, али мањи од свих новчића које имамо на располагању тада је минимум једнак $+\infty$. И у овом случају веома једноставно можемо дефинисати рекурзивну функцију.

```
const int MAX_S = 2000;
const int MAX_V = 100;
const int INF = MAX_S + 1;

// najmanji broj novčića potreban da se naplati iznos s
// kada su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(int v[], int n, int s) {
  // iznos 0 se naplaćuje sa 0 novčića
  if (s == 0)
    return 0;
  // minimalni broj novčića da se naplati iznos s (pretpostavljamo
```

```
  // da iznos nije moguće naplatiti)
  int br = INF;
  // razmatramo sve mogućnosti za poslednji novčić
  for (int i = 0; i < n; i++)
    // proveravamo da li je iznos s moguće naplatiti novčićem i
    if (v[i] <= s)
      // određujemo rekurzivno broj novčića za preostali iznos i
      // ažuriramo minimum ako je to potrebno
      br = min(br, minBrojNovcica(v, n, s-v[i]) + 1);
  // vraćamo rezultat
  return br;
}
```

Пошто долази до понављања истих рекурзивних позива потребно је да употребимо динамичко програмирање. Пошто је само један параметар променљив, за мемоизацију је довољно само да користимо низ.

```
const int MAX_S = 2000;
const int MAX_V = 100;
const int INF = MAX_S + 1;

// niz koji koristimo za memoizaciju - inicijalizovan podrazumevano na 0
int memo[MAX_S + 1];

// najmanji broj novčića potreban da se naplati iznos S
// kada su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(int v[], int n, int S) {
  // ako smo već izračunali broj novčića za iznos S, vraćamo upamćen rezultat
  if (memo[S] != 0)
    return memo[S];
  // iznos 0 se naplaćuje sa 0 novčića
  if (S == 0)
    return memo[S] = 0;
  // minimalni broj novčića da se naplati iznos s (pretpostavljamo
  // da iznos nije moguće naplatiti)
  int br = INF;
  // razmatramo sve mogućnosti za poslednji novčić
  for (int i = 0; i < n; i++)
    // proveravamo da li je iznos s moguće naplatiti novčićem i
    if (v[i] <= S)
      // određujemo rekurzivno broj novčića za preostali iznos i
      // ažuriramo minimum ako je to potrebno
      br = min(br, minBrojNovcica(v, n, S-v[i]) + 1);
  // vraćamo rezultat, pamteći ga pritom u nizu memo
  return memo[S] = br;
}
```

Приликом динамичког програмирања навише низ попуњавамо слева надесно.

```
const int MAX_S = 2000;
const int MAX_V = 100;
const int INF = MAX_S + 1;

// najmanji broj novčića potreban da se naplati iznos S
// kada su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(int v[], int n, int S) {
  int dp[MAX_S + 1];
  // iznos 0 se naplaćuje sa 0 novčića
  dp[0] = 0;
  // računamo minimalni broj novčića za sve ostale iznose
  for (int s = 1; s <= S; s++) {
```

```
    // minimalni broj novčića da se naplati iznos s (pretpostavljamo
    // da iznos nije moguće naplatiti)
    dp[s] = INF;
    // razmatramo sve mogućnosti za poslednji novčić
    for (int i = 0; i < n; i++)
      // proveravamo da li je iznos s moguće naplatiti novčićem i
      if (v[i] <= s)
        // ažuriramo minimum ako je to potrebno
        dp[s] = min(dp[s], dp[s-v[i]] + 1);
  }
  // vraćamo rezultat za iznos S
  return dp[S];
}
```

Задатак: Ранац 0-1

Програмер се сели из једне компаније у другу и жели да понесе предмете из своје старе канцеларије, међутим, на располагању само има један велики ранац у који можда не могу да стану сви предмети. Ако је позната маса и вредност сваког од предмета и ако је позната носивост ранца, напиши програм који одређује максималну вредност скупа предмета које програмер може да пренесе у ранцу.

Улаз: Са стандардног улаза се уноси носивост ранца (цео број између 1 и 150), затим број предмета n (цео број између 1 и 30), затим у наредних n редова масе и цене предмета (масе су цели бројеви између 1 и 10, а цене реални бројеви између $1,0$ и $100,0$, заокружени на две децимале).

Излаз: На стандардни излаз исписати највећу вредност предмета који се могу пренети у ранцу, заокружену на две децимале.

Пример 1

Улаз	*Излаз*
5 3 1 6.00 2 10.00 3 12.00	22.00

Објашњење

Највећа вредност се добије ако се пренесу предмети масе 2 и масе 3 (вредност је тада $10,0 + 12,0 = 22,0$).

Пример 2

Улаз

```
15
5
12 4.00
2 2.00
1 2.00
1 1.00
4 10.00
```

Излаз

```
15.00
```

Објашњење

Највећа вредност се добије ако се пренесу предмети маса 2, 1, 1 и 4 килограма (вредност је тада $2,0 + 2,0 + 1,0 + 10,0 = 15,0$).

Решење

Груба сила

Напомена. Иако неке варијанте проблема ранца допуштају одређена грамзива решења, у овој варијанти у којој се предмети не могу делити већ се узимају у целини (тзв. 0-1 варијанта проблема ранца), потребно је направити алгоритам заснован на неком облику (оптимизоване) исцрпне претраге. Покушајте да конструишете контра-примере на основу којих бисте се уверили да грамзиви алгоритми који би у ранац прво стављао највреднији предмет или који би прво стављао предмет који је највреднији по јединици масе не доводе увек до оптималног решења.

Решење грубом силом подразумева да се испитају сви могући подскупови предмета и да се пронађе онај који може да стане у ранац а има највећу вредност. Сличан облик претраге подскупова ограниченог збира вршили смо у задатку Број поднизова датог збира.

Набрајање подскупова може да се изврши рекурзивном функцијом, чији је параметар дужина низа предмета n.

- Ако је низ предмета празан (ако је $n = 0$), тада је максимална вредност која се може спаковати једнака нули.
- Ако низ предмета није празан (ако је $n > 0$), тада анализирамо могућност да његов последњи елемент изоставимо или да га спакујемо у ранац. У првом случају рекурзивну функцију позивамо за исту вредност капацитета ранца и дужину низа $n-1$. Други случај је могућ само ако је маса последњег предмета мања од капацитета ранца. У том случају на резултат рекурзивног позива где је носивост ранца умањена за масу тог предмета и где је прослеђена дужина низа $n-1$ додајемо цену последњег предмета. Највећу вредност добијамо тако што израчунамо максимум два анализирана случаја (када последњи предмет није и када јесте стављен у ранац).

Ако са $f(n, W)$ обележимо максималну вредност која се може добити тако што се неки од првих n предмета спакују у ранац носивости W, важе следеће везе:

$$\begin{aligned} f(0, W) &= 0 \\ f(n, W) &= \max\left(f(n-1, W), f(n-1, W-w_{n-1})\right) + v_{n-1}, \quad \text{за } w_{n-1} \leq W \\ f(n, W) &= f(n-1, W), \quad \text{за } w_{n-1} > W \end{aligned}$$

Приметимо да рекурзивним позивима се тражи оптимум за скуп без последњег предмета, што је у реду, јер је за глобални оптимум неопходно и да су предмети из тог подскупа одабрани оптимално (задовољен је услов оптималне подструктуре).

Анализа сложености. У наивној рекурзивној имплементацији долази до преклапања идентичних рекурзивних позива, па је та имплементација веома неефикасна (сложеност најгорег случаја јој је експоненцијална).

```
double maxCena(const vector<int>& mase, const vector<double>& cene,
               int nosivost, int n) {
  vector<double> dp(nosivost + 1);
  dp[0] = 0.0;
  for (int N = 1; N <= n; N++)
    for (int M = nosivost; M >= 0; M--)
      if (mase[N-1] <= M)
        dp[M] = max(dp[M], dp[M - mase[N-1]] + cene[N-1]);
  return dp[nosivost];
}
```

Решење проблема поновљених рекурзивних позива, наравно, долази у облику динамичког програмирања (било мемоизације, било динамичког програмирања навише). Пошто имамо два променљива параметра (број предмета n и носивост ранца W), алоцирамо такву матрицу да свакој врсти одговара један префикс низа предмета, а свакој колони једна носивост. Матрицу можемо попуњавати врсту по врсту.

Пример. Прикажимо рад алгоритма на примеру када је носивост ранца 15 килограма и када имамо 5 предмета чије су масе и цене редом $(12, 4.00)$, $(2, 2.00)$, $(1, 2.00)$, $(1, 1.00)$ и $(4, 10.00)$.

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
   -----------------------------------------------
```

```
0 | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
1 | 0  0  0  0  0  0  0  0  0  0  0  0  4  4  4  4
2 | 0  0  2  2  2  2  2  2  2  2  2  2  4  4  6  6
3 | 0  2  2  4  4  4  4  4  4  4  4  4  4  6  6  8
4 | 0  2  3  4  5  5  5  5  5  5  5  5  5  6  7  8
5 | 0  2  3  2 10 12 13 14 15 15 15 15 15 15 15 15
```

Вредност 15 у доњем десном углу указује на то да ако имамо ранац носивости 15 килограма и свих 5 предмета на располагању, тада је највећа вредност коју можемо понети 15. Даље, на пример, вредности у врсти 2 указују на максималне вредности које можемо понети ако узимамо само неке од прва два предмета (то су онај масе 12 и вредности 4,0 и онај масе 2 и вредности 2,0) Ако је носивост мања од 2, ниједан од ова два предмета не може бити спакован. Ако је носивост између 2 и 11 тада може бити спакован само предмет вредности 2,0. Ако је носивост 12 или 13, тада нам се више исплати да спакујемо предмет масе 12 и вредност 4,0. Када носивост пређе 14, тада може запаковати оба предмета, па је укупна вредност 6,0.

Анализа сложености. Овим смо добили алгоритам чија је и временска и меморијска сложеност $O(n \cdot W)$ где је n број предмета, а W носивост ранца.

Приметимо да је веома важно било да је носивост ранца и да су масе предмета целобројне. Такође, обратимо пажњу на то да иако делује да смо овај проблем решили у полиномијалној сложености, то заправо није случај. Наиме, сложеност није изражена само у терминима величине улаза, већ у термину вредности на улазу (вредности носивости ранца). За овакве алгоритме се каже да су псеудо-полиномијални. Величина улаза везаног за број W одговара броју цифара броја W (нпр. броју бинарних цифара употребљених у запису), а време извршавања алгоритма експоненцијално расте у односу на тај број. За веће вредности W добили бисмо веома неефикасан алгоритам (и што се тиче утрошене меморије и што се тиче времена извршавања).

```
double maxCena(const vector<int>& mase, const vector<double>& cene,
               int nosivost, int n) {
  vector<vector<double>> dp(nosivost + 1);
  for (int M = 0; M <= nosivost; M++) {
    dp[M].resize(n+1);
    dp[M][0] = 0.0;
  }

  for (int N = 1; N <= n; N++) {
    for (int M = 0; M <= nosivost; M++) {
      dp[M][N] = dp[M][N-1];
      if (mase[N-1] <= M)
        dp[M][N] = max(dp[M][N-1], dp[M - mase[N-1]][N-1] + cene[N-1]);
    }
  }
  return dp[nosivost][n];
}
```

Можемо приметити да елементи сваке врсте зависе само од претходне, тако да не морамо чувати целу матрицу, већ само текућу врсту. Ажурирање врсте тада вршимо с њеног десног краја.

Анализа сложености. Овом оптимизацијом се меморијска сложеност смањује на $O(W)$, а временска сложеност остаје $O(n \cdot W)$. Приметимо да временска, али и меморијска сложеност остаје експоненцијална у односу на величину улаза (број битова потребних за запис вредности W).

```
double maxCena(const vector<int>& mase, const vector<double>& cene,
               int nosivost, int n) {
  if (n == 0)
     return 0.0;
  double cenaBez = maxCena(mase, cene, nosivost, n-1);
  if (mase[n-1] > nosivost)
     return cenaBez;
  double cenaSa = maxCena(mase, cene, nosivost - mase[n-1], n-1) + cene[n-1];
  return max(cenaBez, cenaSa);
}
```

Задатак: Едит растојање

Едит-растојање између две ниске се дефинише у терминима операција уметања, брисања и измена слова прве речи којима се може добити друга реч. Свака од ове три операције има своју цену. Дефинисати програм који израчунава најмању цену операција којима се од прве ниске може добити друга. На пример, ако је цена сваке операције јединична, тада се ниска `zdravo` може претворити у `bravo!` најефикасније операцијом измене слова `z` у `b`, брисања слова `d` и уметања карактера `!`.

Улаз: Са стандардног улаза се учитавају две ниске дужине највише 100 карактера, а затим цене операције уметања, брисања и измене (природни бројеви од 1 до 10, сваки у посебном реду).

Излаз: На стандардни излаз исписати тражену вредност едит-растојања.

Пример 1

Улаз	*Излаз*

`zdravo`
`bravo!`
`1`
`1`
`1`
	`3`

Пример 2

Улаз	*Излаз*

`kitten`
`sitting`
`1`
`2`
`3`
	`7`

Решење

Рекурзивно решење

Изведимо прво индуктивно-рекурзивну конструкцију.

- Ако је прва ниска празна, најефикаснији начин да се од ње добије друга ниска је да се уметне један по један карактер друге ниске, тако да је минимална цена једнака производу цене операције уметања и броја карактера друге ниске.
- Ако је друга ниска празна, најефикаснији начин да се од прве ниске добије празна је да се један по један њен карактер избрише, тако да је минимална цена једнака производу цене операције брисања и броја карактера прве ниске.
- Индуктивна хипотеза ће бити да умемо да решимо проблем за било која два префикса прве и друге ниске. Ако су последња слова прве и друге ниске једнака, онда је потребно претворити префикс без последњег слова прве ниске у префикс без последњег слова друге ниске. Ако нису, онда имамо три могућности. Једна је да изменимо један од та два карактера у онај други и онда да, као у претходном случају, преведемо префиксе без последњих карактера један у други. Друга могућност је да обришемо последњи карактер прве ниске и пробамо да претворимо тако њен добијени префикс у другу ниску. Трећа могућност је да прву ниску трансформишемо у префикс друге ниске без последњег карактера и да затим додамо последњи карактер друге ниске.

На основу овога лако можемо дефинисати рекурзивну функцију која израчунава едит-растојање. Да нам се ниске не би мењале током рекурзије (што може бити споро), ефикасније је да ниске прослеђујемо у неизмењеном облику и да само прослеђујемо бројеве карактера њихових префикса који се тренутно разматрају.

Анализа сложености. У директној рекурзивној имплементацији долази до великог броја поновљених рекурзивних позива, што доводи до веома лоше (експоненцијалне) сложености.

```
int editRastojanje(const string& s1, const string& s2, int n1, int n2) {
  if (n1 == 0)
    return n2 * cenaUmetanja;
  if (n2 == 0)
    return n1 * cenaBrisanja;
  if (s1[n1-1] == s2[n2-1])
    return editRastojanje(s1, s2, n1-1, n2-1);
  int r1 = editRastojanje(s1, s2, n1-1, n2) + cenaUmetanja;
  int r2 = editRastojanje(s1, s2, n1, n2-1) + cenaBrisanja;
  int r3 = editRastojanje(s1, s2, n1-1, n2-1) + cenaIzmene;
  return min({r1, r2, r3});
}

int editRastojanje(const string& s1, const string& s2) {
```

```
  int n1 = s1.size(), n2 = s2.size();
  return editRastojanje(s1, s2, n1, n2);
}
```

Динамичко програмирање навише

Решење директном рекурзијом је, наравно, изразито неефикасно због преклапајућих рекурзивних позива. Алгоритам динамичког програмирања навише за овај проблем познат је под именом Вагнер-Фишеров алгоритам. Резултате за префиксе дужине i и j памтићемо у матрици на пољу (i, j). Дакле, ако су дужине ниски n_1 и n_2, потребна нам је матрица димензије $(n_1 + 1) \times (n_2 + 1)$, а коначан резултат ће се налазити на месту (n_1, n_2). Ако матрицу попуњавамо врсту по врсту, слева надесно, приликом израчунавања елемента на позицији (i, j), биће израчунати сви елементи матрице од којег он зависи (а то су $(i - 1, j - 1)$, $(i - 1, j)$ и $(i, j - 1)$).

Пример. Под претпоставком да су цене јединичне, за ниске `zdravo` и `bravo!` добија се следећа матрица.

```
     b r a v o !
   0 1 2 3 4 5 6
   -------------
 0|0 1 2 3 4 5 6
z1|1 1 2 3 4 5 6
d2|2 2 2 3 4 5 6
r3|3 3 2 3 4 5 6
a4|4 4 3 2 3 4 5
v5|5 5 4 3 2 3 4
o6|6 6 5 4 3 2 3
```

Анализа сложености. Пошто се током рада алгоритма попуњава матрица димензије $(n_1 + 1) \times (n_2 + 1)$, а свако поље матрице се попуњава у времену $O(1)$, укупна временска и меморијска сложеност алгоритма је $O(n_1 \cdot n_2)$.

```
int editRastojanje(const string& s1, const string& s2,
                   int cenaUmetanja, int cenaBrisanja, int cenaIzmene) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> dp(n1+1);
  for (int i = 0; i <= n1; i++)
    dp[i].resize(n2+1);

  dp[0][0] = 0;
  for (int i = 0; i <= n1; i++)
    dp[i][0] = i * cenaBrisanja;
  for (int j = 0; j <= n2; j++)
    dp[0][j] = j * cenaUmetanja;
  for (int i = 1; i <= n1; i++)
    for (int j = 1; j <= n2; j++) {
      if (s1[i-1] == s2[j-1])
        dp[i][j] = dp[i-1][j-1];
      else {
        int r1 = dp[i-1][j] + cenaUmetanja;
        int r2 = dp[i][j-1] + cenaBrisanja;
        int r3 = dp[i-1][j-1] + cenaIzmene;
        dp[i][j] = min({r1, r2, r3});
      }
    }

  return dp[n1][n2];
}
```

Меморијска оптимизација

На основу поставке задатка, није потребно одредити саме измене, већ само растојање (на пример, ако се врши провера да ли су две ниске блиске приликом претраге у којој се допушта да је корисник направио и неколико словних грешака). Пошто елементи текућег реда зависе само од претходног, можемо извршити меморијску оптимизацију и истовремено чувати само један ред. Током ажурирања елемента на позицији j његов део на позицијама строго мањим од j ће чувати елементе текућег реда i, део од позиције j надаље ће чувати елементе претходног реда $i-1$. Променљива `prethodni` ће чувати вредност са поља $(i-1, j-1)$, а променљива `tekuci` ће чувати вредност са поља $(i-1, j)$.

Анализа сложености. Временска сложеност након ове оптимизације је $O(n_1 \cdot n_2)$, док се меморијска сложеност може спустити на $O(\min(n_1, n_2))$ (тако што се одабере да ли ће се попуњавати врста по врста или колона по колона).

```
int editRastojanje(const string& s1, const string& s2,
                   int cenaUmetanja, int cenaBrisanja, int cenaIzmene) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1);
  dp[0] = 0;
  for (int j = 0; j <= n2; j++)
    dp[j] = j * cenaUmetanja;
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    dp[0] = i * cenaBrisanja;
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1])
        dp[j] = prethodni;
      else {
        int r1 = tekuci + cenaUmetanja;
        int r2 = dp[j-1] + cenaBrisanja;
        int r3 = prethodni + cenaIzmene;
        dp[j] = min({r1, r2, r3});
      }
      prethodni = tekuci;
    }
  }
  return dp[n2];
}
```

Реконструкција оптималног пута

На основу попуњене матрице лако је реконструисати и сам низ корака који прву ниску трансформише у другу. Крећемо од доњег десног угла матрице и крећемо се уназад. У сваком кораку проверавамо како смо дошли на текућу позицију и у складу са тим корак убацујемо у низ (вектор). На крају, када стигнемо до горњег левог угла, низ корака исписујемо уназад.

Пример. У текућем примеру на поље (6, 6) смо стигли са поља (6, 5) што значи да је последњи корак уметање карактера `!`. На поље (6, 5) смо стигли са поља (5, 4) при чему није вршен никаква измена. Слично, на поље (5, 4) смо стигли са (4, 3), на поље (4, 3) смо стигли са (3, 2), а на поље (3, 2) смо стигли са (2, 1). На поље (2, 1) смо могли стићи са (1, 1) при чему је обрисан каракер `d`, а на поље (1, 1) смо стигли са (0, 0) тако што је карактер `z` промењен у `b`. Дакле, један низ могућих корака је

```
zdravo
         измена z у b
bdravo
         брисање d
bravo
         уметање !
bravo!
```

Приметимо да смо на поље (2, 1) могли стићи и са поља (1, 0) операцијом измене `d` у `b`. На поље (1, 0) смо стигли са (0, 0) операцијом брисања слова `z`. На тај начин добијамо следећи низ корака.

```
zdravo
         брисање z
dravo
         измена d у b
bravo
         уметање !
bravo!
```

Имплементација функције којом се врши реконструкција (једног) решења може бити следећа.

```
void ispisiIzmene(const vector<vector<int>>& dp,
                  const string& s1, const string& s2,
                  int cenaUmetanja, int cenaBrisanja, int cenaIzmene) {
  vector<string> izmene;
  int n1 = s1.size(), n2 = s2.size();
  while (n1 > 0 || n2 > 0) {
    cout << n1 << " " << n2 << endl;
    if (n1 > 0 && n2 > 0 && dp[n1][n2] == dp[n1-1][n2-1] && s1[n1-1] == s2[n2-1]) {
      n1--; n2--;
    } else if (n1 > 0 && n2 > 0 && dp[n1][n2] == dp[n1-1][n2-1] + cenaIzmene) {
      izmene.push_back(string("Izmena: ") + s1[n1-1] + " -> " + s2[n2-1]);
      n1--; n2--;
    } else if (n2 > 0 && dp[n1][n2] == dp[n1][n2-1] + cenaUmetanja) {
      izmene.push_back(string("Umetanje: ") + s2[n2-1]);
      n2--;
    } else if (n1 > 0 && dp[n1][n2] == dp[n1-1][n2] + cenaBrisanja) {
      izmene.push_back(string("Brisanje: ") + s1[n1-1]);
      n1--;
    }
  }
  for (auto it = izmene.rbegin(); it != izmene.rend(); it++)
    cout << *it << endl;
}
```

Задатак: Најдужи заједнички подниз две ниске

Напиши програм који израчунава дужину највећег заједничког подниза две ниске. Подниз чине карактери ниске који не морају бити узастопни, али се јављају у истом редоследу као у оригиналној ниски. На пример за ниске `abacbc` и `babbca` најдужа заједничка подниска је `babc`.

Улаз: Две линије стандардног улаза садрже две ниске које се састоје од малих слова енглеске азбуке и дугачке су највише 1000 карактера.

Излаз: На стандардни излаз исписати само тражену дужину.

Пример

Улаз	*Излаз*
`xmjyauz` `mzjawxu`	4

Решење

Рекурзивно решење

Ако је било која од две ниске празна, тада је једини њен подниз празан, па је дужина најдужег заједничког подниза једнака нули.

Ако су обе ниске непразне, тада можемо упоредити њихова последња слова.

- Ако су она једнака, она могу бити укључена у најдужи заједнички подниз две ниске и проблем се рекурзивно своди на проналажење најдужег заједничког подниза префикса тих ниски добијених искључива-

њем последњих слова. Нагласимо да није грешка експлицитно анализирати и случајеве када неко од та два последња слова није укључено у најдужи заједнички подниз (тиме смо сигурнији да ћемо добити коректан алгоритам), али се може доказати да за тим нема потребе.

- У супротном, није могуће да оба последња слова буду укључена у заједнички подниз. Зато разматрамо најдужи заједнички подниз прве ниске и префикса друге ниске без њеног последњег слова и заједнички подниз друге ниске и префикса прве ниске без њеног последњег слова. Дужи од два подниза биће најдужи заједнички подниз те две ниске. Нагласимо и да у овом случају није неопходно експлицитно анализирати најдужи заједнички подниз та два префикса (јер он не може бити дужи од најдужих заједничких поднизова добијених када се неки од тих префикса прошири додатним словом).

Пошто рекурзија тече по префиксима ниски, једини променљиви параметри током рекурзије могу бити дужине тих префикса. Ако са $f(n_1, n_2)$ означимо дужину најдужег заједничког подниза префикса ниске s дужине n_1 и префикса ниске t дужине n_2, тада важи следећа рекурентна веза:

$$
\begin{aligned}
f(0, n_2) &= 0 \\
f(n_1, 0) &= 0 \\
f(n_1, n_2) &= f(n_1 - 1, n_2 - 1) + 1, \quad \text{за } s_{n_1 - 1} = t_{n_2 - 1} \\
f(n_1, n_2) &= \max\left(f(n_1, n_2 - 1), f(n_1 - 1, n_2)\right), \quad \text{за } s_{n_1 - 1} \neq t_{n_2 - 1}
\end{aligned}
$$

```
int najduziZajednickiPodniz(const string& s1, int n1,
                            const string& s2, int n2) {
    if (n1 == 0 || n2 == 0)
       return 0;
    int rez = max(najduziZajednickiPodniz(s1, n1, s2, n2-1),
                  najduziZajednickiPodniz(s1, n1-1, s2, n2));
    if (s1[n1-1] == s2[n2-1])
       rez = max(rez, najduziZajednickiPodniz(s1, n1-1, s2, n2-1) + 1);
    return rez;
}

int najduziZajednickiPodniz(const string& s1, const string& s2) {
  return najduziZajednickiPodniz(s1, s1.size(), s2, s2.size());
}
```

Мемоизација

У директном рекурзивном решењу има много преклапајућих рекурзивних позива. Стога је ефикасност могуће поправити техником динамичког програмирања. Један могући приступ је да употребимо мемоизацију. Вредност дужине најдужег подниза за сваки пар дужина префикса можемо памтити у матрици.

```
int najduziZajednickiPodniz(const string& s1, int n1,
                            const string& s2, int n2,
                            vector<vector<int>>& memo) {
  if (memo[n1][n2] != -1)
    return memo[n1][n2];

  if (n1 == 0 || n2 == 0)
    return memo[n1][n2] = 0;
  int rez = max(najduziZajednickiPodniz(s1, n1, s2, n2-1, memo),
                najduziZajednickiPodniz(s1, n1-1, s2, n2, memo));
  if (s1[n1-1] == s2[n2-1])
    rez = max(rez, najduziZajednickiPodniz(s1, n1-1, s2, n2-1, memo) + 1);
  return memo[n1][n2] = rez;
}

int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> memo(n1+1);
```

```
  for (int i = 0; i <= n1; i++)
    memo[i].resize(n2 + 1, -1);

  return najduziZajednickiPodniz(s1, n1, s2, n2, memo);
}
```

Динамичко програмирање навише

Проблем прекалпајућих рекурзивних позива се може решити ако се употреби динамичко програмирање навише. Дужине најдужих поднизова префикса можемо чувати у матрици. Елемент матрице на позицији (n_1, n_2) зависи само од елемената на позицијама $(n_1 - 1, n_2)$, $(n_1, n_2 - 1)$ и $(n_1 - 1, n_2 - 1)$, тако да матрицу пожемо да попуњавамо било врсту по врсту, било колону по колону.

Пример. Прикажимо матрицу за пример две ниске `xmjyauz` и `mzjawxu`.

```
     mzjawxu
    01234567
   ---------
  0|00000000
x 1|00000011
m 2|01111111
j 3|01122222
y 4|01122222
a 5|01123333
u 6|01123334
z 7|01223334
```

```
int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> dp(n1+1);
  for (int i = 0; i <= n1; i++)
    dp[i].resize(n2 + 1);

  for (int i = 0; i <= n1; i++)
    dp[i][0] = 0;
  for (int j = 0; j <= n2; j++)
    dp[0][j] = 0;

  for (int i = 1; i <= n1; i++)
    for (int j = 1; j <= n2; j++) {
      dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
      if (s1[i-1] == s2[j-1])
        dp[i][j] = max(dp[i][j], dp[i-1][j-1] + 1);
    }

  return dp[n1][n2];
}
```

Меморијска оптимизација

Можемо приметити да се приликом попуњавања матрице врсту по врсту садржај сваке наредне врсте попуњава само на основу претходне врсте. Стога није потребно истовремено памтити целу матрицу, већ је довољно памтити само једну, текућу врсту. Ажурирање врсте морамо вршити с лева надесно, јер сваки елемент у текућој врсти зависи од елемента који му претходи у тој врсти. Приметимо да нам је у неком тренутку потребно да знамо претходни елемент текуће врсте, а понекад претходни елемент претходне врсте, тако да приликом ажурирања врсте морамо да у помоћној променљивој памтимо стару вредност претходног елемента врста (јер се ажурирањем претходног елемента његова стара вредност губи, а она нам може затребати у случају да су одговарајући карактери у нискама једнаки).

```
int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
```

```
  vector<int> dp(n2 + 1, 0);
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1])
        dp[j] = prethodni + 1;
      else
        dp[j] = max(dp[j-1], dp[j]);
      prethodni = tekuci;
    }
  }
  return dp[n2];
}
```

Задатак: Најдужи растући подниз

Напиши програм који одређује најдужи строго растући подниз (не обавезно узастопних елемената) унутар датог низа.

Улаз: Са стандардног улаза се учитава број елемената низа n ($1 \leq n \leq 50000$), а затим елементи низа (цели бројеви, сваки у посебном реду).

Излаз: На стандардни излаз исписати дужину најдужег растућег подниза.

Пример

Улаз	*Излаз*
10 3 2 6 9 5 4 3 7 2 8	4

Објашњење

Један растући подниз дужине 4 је `2 6 7 8`.

Решење

Приказаћемо два решења заснована на динамичком програмирању, која су различите ефикасности.

Најдужи растући подниз који се завршава на датој позицији у низу

У првој групи решења разматраћемо позицију по позицију у низу и одредићемо најдужи растући подниз чији је последњи елемент на свакој од њих. Најдужи растући подниз се онда добити као најдужи од свих тих поднизова (јер се он сигурно завршава на некој позицији у низу). Приликом одређивања дужине најдужег растућег подниза који се завршава на позицији $i \geq 0$, претпоставићемо да за сваку претходну позицију (ако их има) умемо да одредимо дужину најдужег растућег подниза који се на њој завршава. Низ који се завршава на позицији i сигурно садржи елемент a_i, а може продужити све оне низове који се завршавају на некој позицији $0 \leq j < i$ ако је $a_j < a_i$. Да би низ који се завршава на позицији i био што дужи, његов префикс који се завршава на позицији j мора бити што дужи (а дужину најдужег растућег низа за сваку позицију j можемо одредити рекурзивно). Зато од свих низова који се завршавају на позицијама j, таквим да је $a_j < a_i$ одређујемо најдужи и продужавамо га елементом a_i (ако таквих елемената нема, тада је најдужи низ који се завршава на позицији a_i једночлан).

Анализа сложености. У директној рекурзивној имплементацији долази од преклапања рекурзивних позива, што доводи до веома неефикасног решења, експоненцијалне сложености.

```
// pronalazi duzinu najduzeg strogo rastuceg podniza niza a koji se
// zavrsava elementom na poziciji i
int najduziRastuciPodniz(const vector<int>& a, int i) {
  int maksI = 1;
  for (int j = 0; j < i; j++)
    if (a[j] < a[i]) {
      int maksJ = najduziRastuciPodniz(a, j);
      if (maksJ + 1 > maksI)
        maksI = maksJ + 1;
    }
  return maksI;
}

// pronalazi duzinu najduzeg strogo rastuceg podniza niza a
int najduziRastuciPodniz(const vector<int>& a) {
  int maks = 0;
  for (int i = 0; i < a.size(); i++) {
    int maksI = najduziRastuciPodniz(a, i);
    if (maksI > maks)
      maks = maksI;
  }
  return maks;
}
```

Мемоизација

Неефикасност која настаје услед понављања идентичних рекурзивних позива решавамо динамичким програмирањем. За мемоизацију довољно је да памтимо низ дужина најдужих растућих низова који се завршавају на свакој позицији у низу. Пошто су све те дужине веће или једнаке од 1 (сваки елемент сам за себе чини растући низ), низ у који меморишемо решења можемо иницијализовати нулама (што значи да је тражена дужина још непозната).

```
// pronalazi duzinu najduzeg strogo rastuceg podniza niza a koji se
// zavrsava elementom na poziciji i
int najduziRastuciPodniz(const vector<int>& a, int i,
                         vector<int>& memo) {
  if (memo[i] != 0)
    return memo[i];
  int maksI = 1;
  for (int j = 0; j < i; j++) {
    if (a[j] < a[i]) {
      int maksJ = najduziRastuciPodniz(a, j, memo);
      if (maksJ + 1 > maksI)
        maksI = maksJ + 1;
    }
  }
  return memo[i] = maksI;
}

// pronalazi duzinu najduzeg strogo rastuceg podniza niza a
int najduziRastuciPodniz(const vector<int>& a) {
  vector<int> memo(a.size(), 0);
  int maks = 0;
  for (int i = 0; i < a.size(); i++) {
    int maksI = najduziRastuciPodniz(a, i, memo);
    if (maksI > maks)
      maks = maksI;
  }
  return maks;
}
```

Динамичко програмирање навише

Једноставно можемо формулисати и динамичко програмирање навише, тако што низ попуњавамо слева надесно. Након попуњавања низа одређујемо његов максимум. Нагласимо да сваки наредни елемент низа потенцијално зависи од великог броја претходних тако да није могуће редуковати меморијску сложеност тиме што би се памтили само неки елементи низа (морамо увек знати дужине свих претходних елемената).

Пример. Прикажимо на примеру како ће се тај низ попуњавати.

```
i  0 1 2 3 4 5 6 7 8 9
ai 3 2 6 9 5 4 3 7 8 2
dp 1 1 2 3 2 2 2 3 4 1
```

На пример, када израчунавамо елемент на позицији 5 анализирамо низове који се завршавају елементима мањим од вредности 4 која се налази на позицији 5. То су вредности 3 на позицији 0 и 2 на позицији 1. У оба случаја максимална дужина подниза који се завршава на тој позицији је 1, па се било који од тих низова продужава елементом 4 и добија се растући низ дужине 2.

Анализа сложености. Решење које се добија на овај начин је меморијске сложености $O(n)$ и временске сложености $O(n^2)$.

```
// pronalazi duzinu najduzeg strogo rastuceg podniza niza a
int najduzi_rastuci_podniz(const vector<int>& a) {
  int n = a.size();
  vector<int> dp(n);
  for (int i = 0; i < n; i++) {
    dp[i] = 1;
    for (int j = 0; j < i; j++)
      if (a[i] > a[j] && dp[j] + 1 > dp[i])
        dp[i] = dp[j] + 1;
  }

  int max = dp[0];
  for (int i = 0; i < n; i++)
    if (dp[i] > max)
      max = dp[i];
  return max;
}
```

Најмањи елемент којим се завршава растући подниз дате дужине

Изменом индуктивно-рекурзивне конструкције можемо добити и много ефикасније решење. Кључна идеја је да претпоставимо да уз дужину d_{max} најдужег растућег подниза до сада обрађеног дела низа можемо за сваку дужину подниза $1 \leq d \leq d_{max}$ да одредимо најмањи елемент којим се завршава неки растући подниз дужине d.

Пример. Нека је, на пример, дат низ $a = [3, 2, 6, 9, 5, 4, 3, 7, 2, 8]$. Током проласка кроз елементе низа a одржаваћемо низ DP, у коме је на свакој позицији $d - 1 (d \geq 1)$ најмањи елемент низа a, којим се завршава неки растући подниз од a, дужине d.

Ево како се мења низ DP:

```
1 2 3 4 5 6 7 8 9 10
- - - - - - - - - -
3 - - - - - - - - -         3
2 - - - - - - - - -         2
2 6 - - - - - - - -         6
2 6 9 - - - - - - -         9
2 5 9 - - - - - - -         5
2 4 9 - - - - - - -         4
2 3 9 - - - - - - -         3
2 3 7 - - - - - - -         7
2 3 7 - - - - - - -         2
```

```
2 3 7 8 - - - - - -         8
```

Размислимо сада о поступку трансформисања низа DP.

Ако се досадашњи укупно најдужи подниз завршавао елементом који је мањи од текућег, на крај низа DP дописујемо текући елемент јер смо нашли подниз који је дужи за један од претходно најдужег. Текући елемент је једини, па тиме и најмањи којим се завршава растући подниз нове дужине. То се у примеру дешава приликом обраде елемента 3, елемента 6, елемента 9 и елемента 8.

Размотримо и ситуацију у којој обрађујемо елемент 5. До тада смо видели елементе 3, 2, 6 и 9. Елемент 2 на првој позицији у табели означава да је најмањи елемент којим се може завршити једночлани растући низ једнак 2. Елемент 6 на другој позицији у табели означава да је најмањи елемент којим се може завршити двочлани растући низ једнак 6 (у питању је низ 2 6 или низ 3 6). Елемент 9 на трећој позицији у табели означава да је најмањи елемент којим се може завршити трочлани растући низ једнак 9 (у питању је низ 2 6 9 или 3 6 9). Пошто је 5 мањи од 9 ниједан од ових трочланих низова није могуће проширити елементом 5, па четворочланих растућих низова нема. Поставља се питање да ли се можда трочлани низови могу завршити елементом 5, но ни то није могуће. Наиме, пошто је у табели двочланим низовима придружена вредност 6, то значи да се сви двочлани растући низови завршавају бар са 6, па није могуће 9 заменити са 5. Са друге стране, пошто је 5 веће од 2, завршни елемент двочланих низова 6 је могуће заменити са 5 и тиме добити мању завршну вредност двочланих низова (то су у овом случају низови 3 5 и 2 5). Дакле, у табели вредност 6 треба заменити вредношћу 5. Вредност 2 лево од 6 нема смисла заменити са 5, јер би се тиме завршна вредност једночланих низова увећала, а ми у табели памтимо најмање завршне вредности.

Приметимо да низ DP у сваком тренутку мора да буде строго растући. Заиста, ако је a_i најмањи завршетак строго растућег низа дужине d, онда се претпоследњим елементом a_j тог растућег низа (који је мањи од a_i) завршава један растући низ дужине $d-1$, а $DP_{d-2} \leq a_j < a_i = DP_{d-1}$.

На основу анализе овог примера можемо да закључимо да је приликом анализе сваког текућег елемента потребно пронаћи прву позицију d у табели на којој се налази елемент који је већи или једнак од текућег и позицију d уместо тога уписати текући елемент. Ако су сви елементи мањи од текућег (ако је $d = d_{max}$), онда се текући елемент додаје на крај низа (и у том случају заправо радимо исто - уписујемо елемент на позицију d). Остали елементи у табели остају непромењени. Заиста на свим позицијама у табели лево од позиције d уписани су елементи строго мањи од текућег и њиховом заменом са текућим се не би смањила вредност завршног елемената тих низова. За елементе десно од позиције d, иако су већи од текућег, ажурирање није могуће. У свим низовима дужине $d' > d$ неки префикс се морао завршавати елементом на позицији d или елементом већим од њега, а пошто је он био већи или једнак од текућег, заменог последњег елемента текућим не бисмо добили више растући низ.

Кључни добитак настаје када се примети да, пошто су елементи у табели сортирани, позицију првог елемента који је већи или једнак од текућег можемо остварити бинарном претрагом. Отуда следи ефикасна имплементација (у низу `dp` вредност најмањег завршног елемента за низове дужине d памтимо на позицији $d-1$).

Анализа сложености. Временска сложеност такве имплементације је $O(n \log(n))$, док је меморијска сложеност $O(n)$.

Бинарна претрага може бити извршена библиотечком функцијом.

```
// pronalazi duzinu najduzeg strogo rastuceg podniza niza a
int najduziRastuciPodniz(const vector<int>& a) {
  int n = a.size();
  vector<int> dp(n);
  int max = 0;
  for (int i = 0; i < n; i++) {
    auto it = lower_bound(dp.begin(), next(dp.begin(), max), a[i]);
    *it = a[i];
    int d = distance(dp.begin(), it);
    if (d + 1 > max)
      max = d + 1;
  }
  return max;
}
```

Још једна могућност је да се проналазак првог елемента већег или једанког датом коришћењем бинарне претраге имплементира ручно.

```
int prviVeciIliJednak(const vector<int>& a, int l, int d, int x) {
  while (l < d) {
    int s = l + (d - l) / 2;
    if (a[s] < x)
      l = s + 1;
    else
      d = s;
  }
  return l;
}

int najduziRastuciPodniz(const vector<int>& a) {
  int n = a.size();
  vector<int> dp(n+1);
  int max = 0;
  for (int i = 0; i < n; i++) {
    int k = prviVeciIliJednak(dp, 0, max, a[i]);
    dp[k] = a[i];
    if (k + 1 > max)
      max = k + 1;
  }
  return max;
}
```

Свођење на проблем најдужег заједничког подниза

Постоји веома једноставно свођење овог проблема на проблем проналажења најдужег заједничког подниза два низа. Решење тог проблема већ приказано је у задатку Најдужи заједнички подниз две ниске. Наиме, дужина најдужег растућег подниза датог низа једнака је дужини најдужег заједничког подниза тог низа и низа који се добија неопадајућим сортирањем и уклањањем дупликата тог низа.

```
int najduziZajednickiPodniz(const vector<int>& s1, const vector<int>& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1, 0);
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1]) {
        dp[j] = prethodni + 1;
      } else {
        dp[j] = max(dp[j-1], dp[j]);
      }
      prethodni = tekuci;
    }
  }
  return dp[n2];
}

int najduziRastuciPodniz(const vector<int>& a) {
  // sortirana kopija niza a
  vector<int> b = a;
  sort(begin(b), end(b));
  // uklanjamo duplikate iz vektora b
  b.erase(unique(begin(b), end(b)), end(b));
  return najduziZajednickiPodniz(a, b);
}
```

Задатак: Оптимално множење матрица

Множење матрица димензије $D_1 \times D_2$ и димензије $D_2 \times D_3$ даје матрицу димензије $D_1 \times D_3$ и да би се оно спровело потребно је $D_1 \cdot D_2 \cdot D_3$ множења бројева. Када је потребно измножити дужи низ матрица, онда ефикасност зависи од начина како се те матрице групишу (множење је асоцијативна операција и допуштено је било које груписање множења). Напиши програм који одређује најмањи број множења бројева потребан да би се измножио низ матрица датих димензија.

Улаз: Са стандардног улаза се прво учитава број $3 \leq n \leq 100$, а затим и низ димензија $D_0, D_1, D_2, \ldots D_{n-1}$. Матрице које се множе су димензија $D_0 \times D_1$, $D_1 \times D_2$, ..., $D_{n-2} \times D_{n-1}$.

Излаз: На стандардни излаз исписати тражени најмањи број множења.

Пример

Улаз	*Излаз*
4 5 4 6 2	88

Објашњење: Множимо матрицу A димензије 5×4, матрицу B димензије 4×6 и матрицу C димензије 6×2. За рачунање производа $(AB)C$ потребно је $5 \cdot 4 \cdot 6 + 5 \cdot 6 \cdot 2 = 180$ операција, а за рачунање производа $A(BC)$ потребно је $4 \cdot 6 \cdot 2 + 5 \cdot 4 \cdot 2 = 88$ операција множења бројева.

Решење

Рекурзивно решење

Кренимо од рекурзивног решења. Како год да групишемо матрице неко множење је то које се последње извршава. То може бити множење прве матрице и производа свих осталих, множење производа прве две матрице и производа осталих матрица, множење производа прве три матрице и производа осталих матрица итд. све до множења производа свих матрица пре последње последњом матрицом. Основна идеја је да експлицитно изанализирамо све те могућности и да одаберемо најбољу од њих. За сваки фиксирани избор позиције последњег множења потребно је одредити како множити све матрице лево и све матрице десно од те позиције. Кључни увид је да је да би глобални избор био оптималан и та два потпроблема потребно решити на оптималан начин (јер у случају да не одаберемо оптимални распоред заграда у неком потпроблему, боље глобално решење можемо добити заменом тог распореда оптималним). Дакле, потпроблеме можемо решавати рекурзивним позивима.

Нека $f(l, d)$ означава минималан број множења потребан да се измноже матрице димензија $D_l, \ldots, D_d$. Потребно је одредити $f(0, n-1)$. На основу претходне дискусије, важе следеће рекурентне везе.

$$f(l,d) = 0, \quad \text{за } d - l + 1 \leq 2$$
$$f(l,d) = \min_{l<i<d} (f(l,i) + f(i,d) + D_l \cdot D_i \cdot D_d), \quad \text{за } d - l + 1 \geq 3$$

Заиста, ако је $d - l + 1 \leq 2$, то значи да се ради само о једној матрици и није потребно вршити било каква множења.

У супротном постоји више од једне матрице. Позиција последњег множења може бити свака позиција строго већа од l и строго мања од d. Када је последње множење на позицији i значи да се на крају множи производ матрица димензија $D_l, \ldots, D_i$ и производ матрица димензија $D_i, \ldots, D_d$. Рекурзивно одређујемо минималан број множења за сваки од тих производа. Први производ даје матрицу димензије $D_l \times D_i$, други даје матрицу димензије $D_i \times D_d$, и на крају се врши множење те две матрице, за шта је потребно додатних $D_l \times D_i \times D_d$ операција.

```
long long minBrojMnozenja(const vector<int>& dimenzije, int l, int d) {
  int n = d - l + 1;
  if (n <= 2)
    return 0;
  long long min = numeric_limits<long long>::max();
```

```
  for (int i = l+1; i <= d-1; i++) {
    long long broj = minBrojMnozenja(dimenzije, l, i) +
                     minBrojMnozenja(dimenzije, i, d) +
                     dimenzije[l] * dimenzije[i] * dimenzije[d];
    if (broj < min)
      min = broj;
  }
  return min;
}

long long minBrojMnozenja(const vector<int>& dimenzije) {
  return minBrojMnozenja(dimenzije, 0, dimenzije.size() - 1);
}
```

Мемоизација

Директно рекурзивно решење доводи до великог броја идентичних рекурзивних позива и неупоредиво боље решење се добија динамичким програмирањем. Најједноставније решење је додати мемоизацију рекурзивној функцији. Пошто функција има два променљива целобројна параметра (то су l и d), чије се вредности крећу у интервалу $[0, n-1]$, мемоизацију можемо извршити помоћу матрице димензије $n \times n$. Коначно решење се налази на позицији $(0, n-1)$.

```
long long minBrojMnozenja(const vector<int>& dimenzije, int l, int d,
                          vector<vector<long long>>& memo) {
  if (memo[l][d] != -1)
    return memo[l][d];
  int n = d - l + 1;
  if (n == 2)
    return 0;
  long long min = numeric_limits<long long>::max();
  for (int i = l+1; i <= d-1; i++) {
    long long broj = minBrojMnozenja(dimenzije, l, i, memo) +
                     minBrojMnozenja(dimenzije, i, d, memo) +
                     dimenzije[l] * dimenzije[i] * dimenzije[d];
    if (broj < min)
      min = broj;
  }
  return memo[l][d] = min;
}

long long minBrojMnozenja(const vector<int>& dimenzije) {
  int n = dimenzije.size();
  vector<vector<long long>> memo(n);
  for (int i = 0; i < n; i++)
    memo[i].resize(n, -1);
  return minBrojMnozenja(dimenzije, 0, dimenzije.size() - 1, memo);
}
```

Динамичко програмирање навише

Динамичко програмирање навише је донекле компликованије, због компликованијих зависности између елемената матрице. Прво, пошто важи да је $l \leq d$, релевантан нам је само део матрице изнад њене главне дијагонале. Попуњавање није могуће ни по врстама, ни по колонама. Може се уочити да сваки елемент зависи само од елемената који се налазе испод дијагонале на којој се тај елемент налази. Зато је елементе могуће израчунавати редом, по дијагоналама. За $d - l + 1 \leq 2$, важи да је $f(l, d) = 0$, па елементе на главној дијагонали и дијагонали инзад ње постављамо на нулу, а затим рачунамо елементе на дијагоналама изнад њих. Сваку дијагоналу карактерише константни размак између индекса l и d тј. исти број елемената у интервалу $[l, d]$ тј. исти број матрица које се множе.

Пример. Прикажимо на једном примеру како се гради и попуњава матрица. Нека су матрице димензија 4,

3, 5, 1, 2.

```
     0  1  2  3  4
------------------
0 |  0  0 60 27 35
1 |  0  0  0 15 21
2 |  0  0  0  0 10
3 |  0  0  0  0  0
4 |  0  0  0  0  0
```

- Вредност на пољу $(0, 2)$ подразумева да се множе матрице димензија 4×3 и 3×5, за шта је потребно $4 \cdot 3 \cdot 5 = 60$ операција множења.
- Вредност на пољу $(1, 3)$ подразумева да се множе матрице димензија 3×5 и 5×1, за шта је потребно $3 \cdot 5 \cdot 1 = 15$ операција множења.
- Вредност на пољу $(2, 4)$ подразумева да се множе матрице димензија 5×1 и 1×2, за шта је потребно $5 \cdot 1 \cdot 2 = 10$ операција множења.
- Вредност на пољу $(0, 3)$, подразумева да се множе матрице димензије 4×3, 3×5 и 5×1.

  Једна, боља, могућност је да се прво помноже матрице димензије 3×5 и 5×1 за шта је потребно 15 операција множења, а да се онда прва матрица димензије 4×3 помножи добијеном матрицом димензије 3×1, за шта је потребно додатних $4 \cdot 3 \cdot 1 = 12$ операција множења.

  Друга могућност је да се прво помноже прве две матрице, за шта је потребно 60 операција множења, а да се затим добијена матрица димензије 4×5 помножи матрицом димензије 5×1, за шта је потребно додатних $4 \cdot 5 \cdot 1 = 20$ операција множења.
- Вредност на пољу $(1, 4)$, подразумева да се множе матрице димензије 3×5, 5×1 и 1×2.

  Једна могућност је да се прво помноже матрице димензије 5×1 и 1×2 за шта је потребно 10 операција множења, а да се онда прва матрица димензије 3×5 помножи добијеном матрицом димензије 5×2, за шта је потребно додатних $3 \cdot 5 \cdot 2 = 30$ операција множења.

  Друга, боља, могућност је да се прво помноже прве две матрице, за шта је потребно 15 операција множења, а да се затим добијена матрица димензије 3×1 помножи матрицом димензије 1×2, за шта је потребно додатних $3 \cdot 1 \cdot 2 = 6$ операција множења.
- Вредност на пољу $(0, 4)$, подразумева да се множе матрице димензије 4×3, 3×5, 5×1 и 1×2.

  Једна могућност је да се прва матрица димензије 4×3 помножи производом осталих матрица, за који знамо да се може израчунати помоћу 21 операције множења. Након тога је потребно помножити матрицу димензије 4×3 и добијену матрицу димензије 3×2, за шта је потребно додатних $4 \cdot 3 \cdot 2 = 24$ операције множења.

  Друга могућност је да се производ матрица димензија 4×3 и 3×5, који се израчунава помоћу 60 операција множења помножи производом матрица димензија 5×1 и 1×2, који се израчунава помоћу 10 операција множења. Производ добијених матрица димензија 4×5 и 5×2 захтева још $4 \cdot 5 \cdot 2 = 40$ операција множења.

  Трећа, најбоља, могућност је да се производ прве три матрице, који се израчунава помоћу 27 операција множења помножи последњом матрицом димензије 1×2. Пошто је производ прве три матрице димензије 4×1, ово захтева додатних $4 \cdot 1 \cdot 2 = 8$ операција множења.

Анализом зависности између елемената матрице може се установити да није могуће једноставно смањивање меморијске сложености, као што је то био случај у неким раније приказаним проблемима.

```
long long minBrojMnozenja(const vector<int>& dimenzije) {
  int n = dimenzije.size();
  vector<vector<long long>> dp(n);
  for (int i = 0; i < n; i++)
    dp[i].resize(n, 0);

  for (int k = 3; k <= n; k++)
    for (int l = 0, d = l + k - 1; d < n; l++, d++) {
```

```
      dp[l][d] = numeric_limits<long long>::max();
      for (int i = l+1; i <= d-1; i++) {
        long broj = dp[l][i] + dp[i][d] +
          dimenzije[l] * dimenzije[i] * dimenzije[d];
        if (broj < dp[l][d])
          dp[l][d] = broj;
      }
    }

  return dp[0][n-1];
}
```

Реконструкција оптималног редоследа множења

Да би се реконструисало решење, можемо у посебној матрици памтити позиције минималних елемената.

```
void odstampaj(const vector<vector<int>>& poz, int l, int d) {
  int n = d - l + 1;
  if (n <= 2)
    cout << "A" << l;
  else {
    cout << "(";
    odstampaj(poz, l, poz[l][d]);
    cout << "*";
    odstampaj(poz, poz[l][d], d);
    cout << ")";
  }
}

long long minBrojMnozenja(const vector<int>& dimenzije) {
  int n = dimenzije.size();
  vector<vector<long long>> dp(n);
  for (int i = 0; i < n; i++)
    dp[i].resize(n, 0);

  vector<vector<int>> poz(n);
  for (int i = 0; i < n; i++)
    poz[i].resize(n);


  for (int k = 3; k <= n; k++)
    for (int l = 0, d = l + k - 1; d < n; l++, d++) {
      dp[l][d] = numeric_limits<long long>::max();
      for (int i = l+1; i <= d-1; i++) {
        int broj = dp[l][i] + dp[i][d] +
          dimenzije[l] * dimenzije[i] * dimenzije[d];
        if (broj < dp[l][d]) {
          dp[l][d] = broj;
          poz[l][d] = i;
        }
      }
    }
  odstampaj(poz, 0, n-1);
  cout << endl;
  return dp[0][n-1];
}
```

Задатак: Најдужи подниз палиндром

Написати програм којим се за дату ниску s одређује дужина најдужег подниза ниске s који је палиндром. Подниз не мора да садржи само узастопне карактере ниске, али они морају да се јављају у истом редоследу (подниз се добија брисањем 0 или више карактера).

Улаз: Са стандардног улаза се учитава ниска s састављена само од малих слова енглеске абецеде, чија је дужина највише 5000 карактера.

Излаз: На стандардни излаз исписати само тражену дужину најдужег палиндромског подниза.

Пример

Улаз	*Излаз*
`najduzipalindrom`	`5`

Решење

Рекурзивно решење

Кренимо од рекурзивног решења.

- Празна ниска има само празан подниз, па је дужина најдужег палиндромског подниза једнака нули. Ниска дужине 1 је сама свој палиндромски подниз, па је дужина њеног најдужег палиндромског подниза једнака 1.
- Ако ниска има бар два карактера, онда рамзатрамо да ли су њен први и последњи карактер једнаки. Ако јесу, онда они могу бити део најдужег палиндромског подниза и проблем се своди на проналажење најдужег палиндромског подниза дела ниске без првог и последњег карактера. У супротном они не могу истовремено бити део најдужег палиндромског подниза и потребно је елиминисати бар један од њих. Проблем, дакле, сводимо на проналажење најдужег палиндромског подниза суфикса ниске без првог карактера и на проналажење најдужег палиндромског подниза префикса ниске без последњег карактера. Дужи од та два палиндромска подниза је тражени палиндромски подниз целе ниске.

Овим је практично дефинисана рекурзивна процедура којом се решава проблем. У сваком рекурзивном позиву врши се анализа неког сегмента (низа узастопних карактера полазне ниске), па је сваки рекурзивни позив одређен са два броја који представљају границе тог сегмента. Ако са $f(l, d)$ означимо дужину најдужег палиндромског подниза дела ниске $s[l, d]$, тада важе следеће рекурентне везе.

$$
\begin{aligned}
f(l,d) &= 0, \quad \text{за } l > d \\
f(l,d) &= 1, \quad \text{за } l = d \\
f(l,d) &= 2 + f(l+1, d-1), \quad \text{за } l < d \text{ и } s_l = s_d \\
f(l,d) &= \max\left(f(l+1,d), f(l,d-1)\right), \quad \text{за } l < d \text{ и } s_l \neq s_d
\end{aligned}
$$

На основу овога, функцију је веома једноставно имплементирати.

```
int najduziPalindrom(const string& s, int l, int d) {
  if (l > d)
    return 0;
  if (l == d)
    return 1;
  if (s[l] == s[d])
    return 2 + najduziPalindrom(s, l+1, d-1);
  return max(najduziPalindrom(s, l, d-1),
             najduziPalindrom(s, l+1, d));
}

int najduziPalindrom(const string& s) {
  return najduziPalindrom(s, 0, s.length() - 1);
}
```

Анализа сложености. У овој имплементацији долази до преклапања рекурзивних позива, па је она веома неефикасна (сложеност најгорег случаја је експоненцијална).

Мемоизација

У претходној функцији долази до преклапања рекурзивних позива, па је пожељно употребити мемоизацију. За мемоизацију користимо матрицу (практично, њен горњи троугао у којем је $l < d$).

```
int najduziPalindrom(const string& s, int l, int d,
                     vector<vector<int>>& memo) {
  if (memo[l][d] != -1)
    return memo[l][d];
  if (l > d)
    return memo[l][d] = 0;
  if (l == d)
    return memo[l][d] = 1;
  if (s[l] == s[d])
    return memo[l][d] = 2 + najduziPalindrom(s, l+1, d-1, memo);
  return memo[l][d] = max(najduziPalindrom(s, l, d-1, memo),
                          najduziPalindrom(s, l+1, d, memo));
}

int najduziPalindrom(const string& s) {
  vector<vector<int>> memo(s.length(), vector<int>(s.length(), -1));
  return najduziPalindrom(s, 0, s.length() - 1, memo);
}
```

Динамичко програмирање навише

До ефикасног решења можемо доћи и динамичким програмирањем одоздо навише. Елемент на позицији (l, d) матрице зависи од елемената на позицијама $(l + 1, d)$, $(l, d - 1)$ и $(l + 1, d - 1)$, док се коначно решење налази у горњем левом углу матрице, тј. на пољу $(0, n - 1)$. Због оваквих зависности матрицу не можемо попуњавати ни врсту по врсту, ни колону по колону, већ дијагоналу по дијагоналу. На дијагоналу испод главне уписујемо све нуле, на главну дијагоналу све јединице, а затим попуњавамо једну по једну дијагоналу изнад главне, све док не дођемо до елемента у горњем левом углу.

Пример. Прикажимо како изгледа попуњена матрица на примеру ниске `abaccba`.

```
    abaccba
    0123456
    -------
a 0|1133346
b 1|0111244
a 2| 011224
c 3|  01222
c 4|   0111
b 5|    011
a 6|     01
```

Коначно решење 6 одговара поднизу `abccba`.

```
int najduziPalindrom(const string& s) {
  int n = s.length();
  vector<vector<int>> dp(n);
  for (int i = 0; i < n; i++) {
    dp[i].resize(n, 0);
    dp[i][i] = 1;
  }
  for (int k = 1; k < n; k++)
    for (int l = 0; l + k < n; l++) {
      int d = l + k;
      if (s[l] == s[d])
        dp[l][d] = dp[l+1][d-1] + 2;
      else
        dp[l][d] = max(dp[l+1][d], dp[l][d-1]);
```

```
    }

  return dp[0][n - 1];
}
```

Анализа сложености. Ово решење има и меморијску и временску сложеност $O(n^2)$.

Меморијска оптимизација

Примећујемо да елементи сваке дијагонале зависе само од елемената претходне две дијагонале. Могуће је да чувамо само две дијагонале - текућу и претходну. Током ажурирања текуће дијагонале њене постојеће елементе истовремено преписујемо у претходну. Када су карактери једнаки, тада у привремену промењиву бележимо одговарајући елемент претходне дијагонале, на његово место уписујемо одговарајући елемент текуће дијагонале, а онда на место тог елемента уписујемо вредност привремене промењиве увећану за два. Када су карактери различити одговарајући елемент текуће дијагонале уписујемо на одговарајуће место у претходној дијагонали, а на његово место уписујемо максимум те и наредне вредности текуће дијагонале.

```
int najduziPalindrom(const string& s) {
  int n = s.length();
  // elementi dve prethodne dijagonale
  vector<int> dpp(n, 0);
  vector<int> dp(n, 1);
  for (int k = 1; k < n; k++) {
    for (int l = 0; l + k < n; l++) {
      int d = l + k;
      if (s[l] == s[d]) {
        int tmp = dp[l];
        dp[l] = dpp[l+1] + 2;
        dpp[l] = tmp;
      }
      else {
        dpp[l] = dp[l];
        dp[l] = max(dp[l], dp[l+1]);
      }
    }
    dpp[n-k] = dp[n-k];
  }

  return dp[0];
}
```

Анализа сложености. Меморијска сложеност овог решења је $O(n)$, док временска сложеност остаје $O(n^2)$.

Свођење на проблем најдужег заједничког подниза две ниске

Рецимо још и да је решење овог задатка могуће добити и свођењем на проблем одређивања најдужег заједничког подниза две ниске. Тај је проблем описан у задатку Најдужи заједнички подниз две ниске. Наиме, најдужи палиндромски подниз једнак је најдужем заједничком поднизу оригиналне ниске и ниске која се добија њеним обртањем.

```
int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1, 0);
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1]) {
        dp[j] = prethodni + 1;
      } else {
        dp[j] = max(dp[j-1], dp[j]);
```

```
      }
      prethodni = tekuci;
    }
  }
  return dp[n2];
}

int najduziPalindrom(const string& s) {
  string sObratno = s;
  reverse(begin(sObratno), end(sObratno));
  return najduziZajednickiPodniz(s, sObratno);
}
```

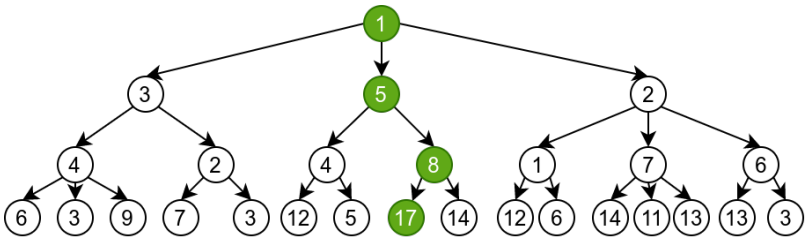
Анализа сложености. Сложеност ове редукције је иста као и сложеност директног решења (временски $O(n^2)$, а просторно $O(n)$).

Глава 9

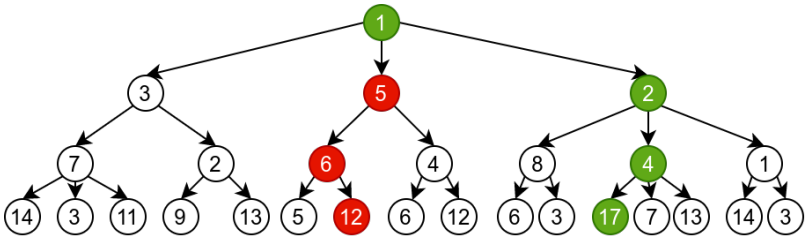
Грамзиви алгоритми

Алгоритми засновани на претрази до решења долазе кроз низ корака где у сваком кораку анализирају неколико могућности. Алгоритми код којих се уместо анализе различитих могућности у сваком кораку узима неко *локално оптимално* решење називају се **похлепни** или **грамзиви алгоритми** (енгл. greedy algorithms). Такве алгоритме обично има смисла примењивати само код проблема код којих постоји гаранција да ће такви избори на крају довести до *глобално оптималног* решења.

На слици су приказана два дрвета претраге таква да у случају првог дрвета похлепни алгоритам који у сваком кораку бира наследника са највећом могућом вредношћу доводи до оптималног решења (максималне могуће вредности), док у случају другог дрвета похлепни алгоритам доводи до неоптималног решења (вредности која није оптимална).



Слика 9.1: Дрво претраге код којег грамзиви алгоритам доводи до оптималног решења (максималне вредности)



Слика 9.2: Дрво претраге код којег грамзиви алгоритам не доводи до оптималног решења (максималне вредности)

Похлепни алгоритми нам, дакле, пружају јасну стратегију (бирај што бољу од свих расположивих могућности) како да у сваком кораку изаберемо једну од више понуђених могућности тако да на крају дођемо до жељеног оптималног решења. У наставку ћемо појам грамзивих алгоритама мало проширити и разматраћемо алгоритме који у сваком кораку бирају само једну могућност на основу неке прецизно дефинисане стратегије, тако да ти избори гарантују да ће се на крају доћи до жељеног (исправног тј. оптималног) решења проблема.

Похлепни алгоритми не врше испитивање различитих случајева нити исцрпну претрагу и стога су по правилу веома ефикасни (у сваком кораку је извршено максимално могуће одсецање). Такође, обично се веома јед-

ноставно имплементирају. Са друге стране, као и код свих других алгоритама у којима се користи одсецање, потребно је унапред доказати да се похлепним алгоритмом добија коректно тј. оптимално решење, што у неким случајевима може бити веома изазовно. Само налажење исправног похлепног алгоритма (тј. стратегије) може представљати озбиљан изазов и често није тривијално одредити да ли за неки проблем постоји или не постоји похлепно решење.

Код неких проблема похлепни алгоритми не доводе увек до оптималног решења, али се може доказати да ће решења која се добијају бити квалитетна и неће се пуно разликовати од оптималних, што може оправдати употребу похлепних алгоритама (јер они могу бити много ефикаснији од исцрпних алгоритама који гарантују оптималност). У том случају похлепни алгоритми су *хеуристике* (технике које не гарантују да ће увек довести до оптималног решења, али који доводе до довољно добрих решења).

Алгоритми засновани на претрази или на динамичком програмирању обично у сваком кораку разматрају више могућности (којима се добија више потпроблема) и након разматрања свих могућности бирају ону најбољу. Дакле, избор се врши тек након решавања потпроблема. За разлику од тога грамзиви алгоритми унапред знају која могућност ће водити до оптималног решења и избор врше одмах, након чега решавају само један потпроблем. У случају оптмизационих проблема и у случају грамзивих алгоритма потребно је да важи својство **оптималне подструктуре** тј. да се оптимално решење полазног проблема добија помоћу оптималног решења потпроблема.

Да би се доказала коректност похлепног алгоритма, обично је потребно доказати неколико ствари. Иако ћемо некада грамзивим алгоритмима решавати проблеме у којима се захтева да се испита да се провери да ли постоји неко решење које задовољава дате услове и да се пронађе било које такво решење, најчешће ћемо разматрати проблеме у којима се захтева да се у групи решења која задовољавају неке дате услове (исправних решења) пронађе оно оптимално (у случају када постоји више таквих оптималних решења обично је довољно да се пронађе било које).

1. Прво је потребно доказати да похлепна стратегија даје решење које је исправно тј. решење које задовољава све услове задатка.

2. Након тога је потребно доказати и да је решење добијено похлепном стратегијом оптимално. Ти докази су по правилу тежи и постоји неколико техника како се они изводе. Обично се крене од неког решења за које претпостављамо да је оптимално и које не мора бити идентично ономе које смо добили похлепном стратегијом. Оно не може бити горе од решења нађеног на основу похлепне стратегије (јер она враћа једно коректно решење, па оптимум може бити само евентуално бољи од тог решења), а потребно је доказати да не може бити боље.

    - Једна техника да се оптималност докаже је то да се покаже да се оптимално решење може мало по мало, применом трансформације појединачних корака може претворити у решење добијено на основу наше стратегије. Обично је довољно доказати да се први корак оптималног решења може заменити првим кораком који грамзива стратегија сугерише, тако да се коректност и квалитет решења тиме не нарушавају и коректност даље следи на основу индуктивног аргумента. Ову технику називаћемо **техника размене** (енгл. exchange).

    - Једна техника да се оптималност докаже је то да се докаже да је решење добијено на основу похлепне стратегије увек по неком критеријуму испред претпостављеног оптималног решења. Ову технику називаћемо **похлепно решење је увек испред** (енгл. greedy stays ahead).

    - Једна техника да се оптималност докаже је да се одреди теоријска граница вредности оптимума и да се онда докаже да похлепни алгоритам даје решење чија је вредност управо једнака оптимуму. Ову технику називаћемо **техника границе** (енгл. structural bound).

Задатак: Реч у реч прецртавањем слова

Дате су две речи записане малим словима. Написати програм којим се проверава да ли се друга реч може добити прецртавањем слова (не обавезно суседних) у првој речи.

Улаз: У првој линији стандардног улаза налази се прва реч, а у другој линији друга реч.

Излаз: У првој линији стандардног излаза приказати реч `da` ако се друга реч може добити прецртавањем неких слова прве речи, иначе приказати реч `ne`.

Пример 1		**Пример 2**	
Улаз	*Излаз*	*Улаз*	*Излаз*
`anica`	`da`	`anica`	`ne`
`ana`		`cana`	

Решење

При прецртавању слова, њихов редослед се не мења што значи да речи редослед слова у другој речи мора бити исти као редослед прецртаних слова у првој речи. Свако слово друге речи мора да се јави у првој. Ако постоји неко исправно прецртавање, онда се исправно прецртавање може добити и тако што се у првој речи задржи прво појављивање првог слова друге речи (сва слова испред њега се прецртају) и остала слова друге речи потраже иза тог првог појављивања.

Пример. На пример, ако се у речи `bcababaca` тражи реч `abac`, тада се прецртају слова `bc`, задржи се прво `a` и затим се унутар дела `babaca` потражи реч `bac`.

Наиме, ако би постојало решење у којем је прво појављивање првог слова друге речи у првој речи прецртано, а задржано је неко његово касније појављивање, пошто се сва остала задржана слова друге речи у првој речи налазе иза тог каснијег појављивања првог слова, могли бисмо извршити размену и прецртати то задржано касније појављивање, а задржати прво појављивање и тако добити исправно прецртавање у којем је задржано баш прво појављивање.

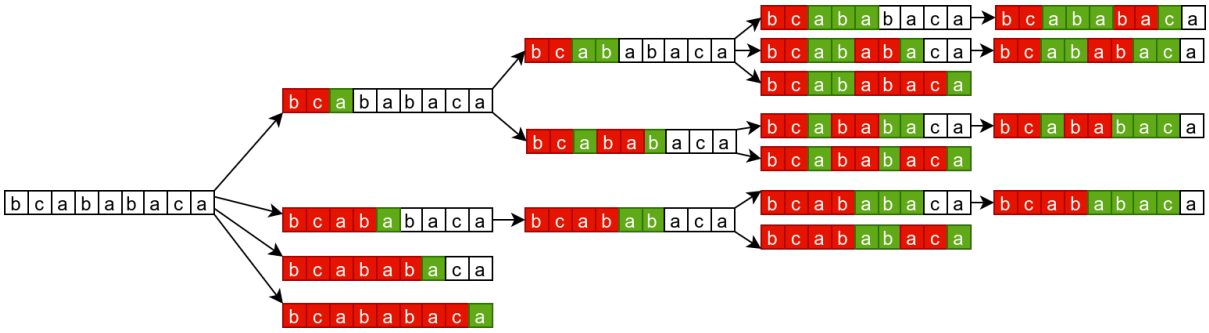
Пример. Једно могуће прецртавање слова које од речи `bcababaca` даје реч `abac` је `BCABabacA` (прецртана слова су приказана великим словима). Видимо да је прецртано прво појављивање слова `A`, док је непрецртано неко његово касније појављивање. Исправно прецртавање можемо добити тако што оставимо прво `a` које је било прецртано, а прецртамо `a` које није било прецртано. Тако добијамо прецртавање `BCaBAbacA`.

Сличан је случај и са даљим словима (увек у првој речи бирамо прво непрецртано појављивање текућег слова друге речи, јер ако реч можемо добити прецртавањем слова, можемо је добити и коришћењем те стратегије).

Пример. Размене се могу наставити. Од прецртавања `BCaBAbacA`, разменом можемо добити прецртавање `BCabABacA` и затим `BCabaBAcA`. Ово прецртавање је такво да је увек задржано прво појављивање текућег слова друге речи.

Стога се решење заснива на једноставном грамзивом алгоритму (прецртавамо прво појављивање текућег слова друге речи у првој на које наиђемо и не разматрамо остале варијанте), чија се коректност заснива на томе што се било које друго исправно решење може разменама преправити исправно решење добијено грамзивом стратегијом. Дакле, ако постоји исправно прецртавање постоји и исправно прецртавање које ће дати грамзива стратегија, што значи да ако грамзива стратегија не успе да пронађе исправно прецртавање, тада исправно прецртавање не постоји (друга реч није подниз прве).

На слици је приказано дрво исцрпне претраге приликом тражења речи `abac` унутар речи `bcababaca`. Грамзиви алгоритам избегава анализу целог дрвета тако што се креће само дуж његове горње ивице.



Нагласимо и да је наша грамзива стратегија таква да у случају када постоји више могућности, бира ону која оставља највише могућности да се преостали потпроблем успешно реши (избором првог појављивања првог слова друге речи унутар прве речи, остатак друге речи се надаље тражи у најдужем могућем преосталом делу ниске, што повећава вероватноћу да се пронађе успешно прецртавање). Ово је и општи савет за креирање грамзивих стратегија.

Довољно је пролазити у петљи кроз прву реч, карактер по карактер, и проверавати да ли је текући карактер једнак карактеру који је на реду у другој речи (на почетку, на реду је први карактер са индексом 0). Ако су одговарајући карактери једнаки, прелази се на наредни карактер у обе речи, а иначе само у првој. Петљу

треба прекинути када прођемо кроз све карактере бар једне речи. Ако смо прошли кроз све карактере друге речи, закључујемо да се друга реч може добити прецртавањем слова прве, иначе не.

Анализа сложености. Два показивача `i` и `j` се померају, први по првој, а други по другој речи само у једном смеру, па је сложеност овог алгоритма јасно $O(m+n)$, где су m и n дужине речи.

Решење можемо реализовати помоћу два бројача и петље `while`.

```
// provera da li je niska s2 podniz (ne obavezno uzastopnih karaktera) niske s1
bool jePodniz(const string& s1, const string& s2) {
  // redom prolazimo kroz slova obe reci dok ne dodjemo do kraja jedne
  // od njih
  int i = 0, j = 0;
  while (i < s1.size() && j < s2.size()) {
    // ako je tekuce slovo prve reci jednako tekucem slovu druge reci
    // onda ga zadrzavamo, a u suprotnom ga precrtavamo
    if (s1[i] == s2[j])
      // ako smo slovo zadrzali, prelazimo na naredno slovo druge reci
      j++;
    // prelazimo na naredno slovo prve reci
    i++;
  }

  // druga rec je podniz prve ako i samo ako smo stigli do kraja druge reci
  // (sva slova druge reci smo pronasli u prvoj)
  return j == s2.size();
}
```

Други поглед на исти овај поступак је да се за сваки карактер друге речи провери да ли постоји у првој речи и ако постоји, пронађе његово прво појављивање, при чему се претрага за првим карактером врши од почетка прве речи, а за сваким наредним од позиције иза оне на којој је претходни карактер нађен. Ако се неки карактер не може пронаћи, тада другу реч није могуће добити од прве. Ако се сви карактери друге речи успешно пронађу, онда је другу реч могуће добити од прве. Претрагу карактера можемо вршити или алгоритмом линеарне претраге.

```
// provera da li je niska s2 podniz (ne obavezno uzastopnih karaktera) niske s1
bool jePodniz(const string& s1, const string& s2) {
  // tekuca pozicija u prvoj reci
  int i = 0, j;
  // trazimo jedno po jedno slovo druge reci u prvoj
  for (j = 0; j < s2.size(); j++) {
    // pretragu pokrecemo od pozicije i
    while (i < s1.size()) {
      // ako smo nasli slovo, prekidamo pretragu
      if (s2[j] == s1[i])
        break;
      i++;
    }
    // ako smo dosli do kraja prve reci, nismo nasli slovo i prekidamo
    // pretragu
    if (i == s1.size())
      break;
  }
  // druga rec je podniz prve ako i samo ako smo stigli do kraja druge reci
  // (sva slova druge reci smo pronasli u prvoj)
  return j == s2.size();
}
```

Линеарну претрагу у делу ниске можемо вршити и библиотечким функцијама. У језику C++ може се употребити метода `find` која враћа специјалну вредност `string::npos` ако карактер није нађен. Постоји варијанта

ове методе која при карактер који се тражи и позицију од које креће претрага.

```
// provera da li je niska s2 podniz (ne obavezno uzastopnih karaktera) niske s1
bool jePodniz(const string& s1, const string& s2) {
  // trazimo jedno po jedno slovo druge rece u prvoj, krenuvsi od pozicije i
  int i, j;
  for (i = 0, j = 0; j < s2.size(); i++, j++) {
    // trazimo tekuce slovo drugoj reci u prvoj, krenuvsi od pozicije i
    i = s1.find(s2[j], i);
    // ako ga nismo nasli, tada prekidamo pretragu
    if (i == string::npos)
      break;
    // prelazimo na sledecu poziciju u obe reci
  }

  // druga rec je podniz prve ako i samo ako smo stigli do kraja druge reci
  // (sva slova druge reci smo pronasli u prvoj)
  return j == s2.size();
}
```

Задатак: Жаба на камењу

Камење је постављено дуж позитивног дела x-осе и за сваки камен је позната његова координата x. Жаба креће да скаче са првог камена који се налази у координатном почетку и жели да у што мање скокова дође до последњег камена. У сваком скоку она може да прескочи највише растојање r (а може да скочи и мање, ако је то потребно). Написати програм који одређује да ли жаба може стићи до последњег камена и ако може у колико најмање скокова то може учинити.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 50000$), а затим у наредном реду n позитивних целих бројева број (у питању је растуће сортиран низ бројева који представља координате камења). У последњем реду се налази позитиван цео број r.

Излаз: На стандардни излаз исписати најмањи број скокова потребан да жаба стигне до последњег камена или -1 ако то није могуће.

Пример

Улаз	*Излаз*
5 0 3 8 14 16 10	2

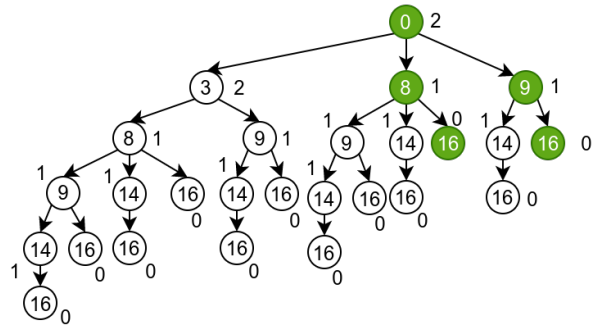
Решење

Груба сила - динамичко програмирање

Један начин да се задатак реши је грубом силом, тј. испробавањем свих могућности. Иако се на овај начин добија алгоритам чија је коректност прилично очигледна (јер су све могућности експлицитно испитане), у овом задатку такво решење је непотребно неефикасно, јер, како ћемо видети, постоји веома једноставна грамзива стратегија која увек доводи до оптималног решења (најмањег броја скокова).

Основна идеја алгоритма грубе силе је да се покуша скок са почетног камена на све каменове на које се може доскочити и да се затим рекурзивно израчуна минимални број скокова са камена на који смо доскочили до крајњег камена. Ако елиминишемо каменове на које смо доскочили са којих не постоји пут до крајњег камена и од преосталих каменова пронађемо минимални број скокова до крајењег камена, минимални број скокова од почетног камена до крајњег је број који је за један већи од тог броја.

Пример. На слици је приказано дрво исцрпне претраге за низ каменова $0, 3, 8, 9, 14, 16$ и дужину скока $r = 10$. Са камена 0, жаба може да скочи на каменове 3, 8 и 9, са камена 3 на каменове 8 и 9, са камена 8 на каменове 9, 14 и 16, са камена 9 на каменове 14 и 16 и са камена 14 на камен 16. Уз сваки чвор је обележен и најмањи број скокова потребних да жаба дође до завршног камена 16. Означене су и две најкраће путање ($0 - 8 - 16$ и $0 - 9 - 16$).



Слика 9.3: Дрво исцрпне претраге

Пошто једноставна рекурзивна имплементација доводи до понављених рекурзивних позива (може се очекивати да ћемо током претраге често долазити на неки камен и рачунати минимални број скокова од тог камена до краја), програм треба оптимизовати динамичким програмирање.

Једно решење је да се употреби мемоизација.

Анализа сложености. Ако би се са сваког камена могло доскочити до сваког другог, овим бисмо добили решење сложености $O(n^2)$ (уз коришћење помоћног низа дужине n).

```
// minimalni broj skokova od kamena i do poslednjeg kamena n-1
int brojSkokova(const vector<int>& kamenje, int i, int r, vector<int>& memo) {
  int n = kamenje.size();

  // zaba je vec na poslednjem kamenu
  if (i == n - 1)
    return 0;

  // ako smo vec resavali ovaj problem, ocitavamo resenje
  if (memo[i] != 0)
    return memo[i];

  // minimalni broj skokova (n ima ulogu +beskonacno)
  int min = n;
  // analiziramo jedan po jedan kamen na koji zaba moze da doskoci sa kamena i
  for (int j = i+1; j < n && kamenje[j] <= kamenje[i] + r; j++) {
    // rekurzivno odredjujemo najmanji broj skokova od j do n-1
    int broj = brojSkokova(kamenje, j, r, memo);
    // ako se od j moze stici do n-1, azuriramo minimum ako je potrebno
    if (broj != n && broj + 1 < min)
      min = broj + 1;
  }
  // memorisemo i vracamo minimalni broj skokova od i do n-1
  // ako nije moguce stici, vrednost ce ostati n
  return memo[i] = min;
}

int brojSkokova(const vector<int>& kamenje, int r) {
  int n = kamenje.size();
  vector<int> memo(n, 0);
  // odredjujemo minimalni broj skokova od kamena 0 do n-1
  int broj = brojSkokova(kamenje, 0, r, memo);
  // funkcija vraca n ako nije moguce doci do n-1, a treba da vrati -1
  return broj == n ? -1 : broj;
}
```

Уместо мемоизације можемо употребити и динамичко програмирање навише. Низ у коме чувамо минимални

број скокова до крајњег камена треба попуњавати од крајњег камена налево, у опадајућем редоследу координата.

Анализа сложености. Сложеност овог решења је такође $O(n^2)$ (уз коришћење помоћног низа дужине n).

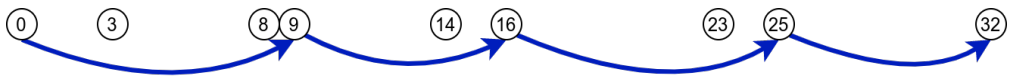
```
int brojSkokova(const vector<int>& kamenje, int r) {
  int n = kamenje.size();
  vector<int> dp(n);
  // zaba se vec nalazi na poslednjem kamenu
  dp[n-1] = 0;
  // racunamo minimalni broj skokova za svaki kamen, unazad
  for (int i = n-2; i >= 0; i--) {
    // n ima ulogu +beskonacno
    dp[i] = n;
    // analiziramo jedan po jedan kamen na koji zaba moze da
    // doskoci sa kamena i
    for (int j = i+1; j < n && kamenje[j] <= kamenje[i] + r; j++)
      // ako se od j moze stici do n-1, azuriramo minimum
      // ako je potrebno
      if (dp[j] != n && dp[j] + 1 < dp[i])
        dp[i] = dp[j] + 1;
  }

  // dp[0] je minimalni broj skokova od kamena 0 do n-1
  // dp[0] je jednako n ako nije moguce doci do n-1, a treba da vrati -1
  return dp[0] == n ? -1 : dp[0];
}
```

Оптимална стратегија - грамзиви алгоритам

Иако у сваком кораку жаба може имати избор између неколико каменова на које може да скочи, интуитивно нам је сасвим јасно да она ништа не губи тиме што скочи што даље. Стога је јасно да жаба у сваком кораку треба да скочи на што даљи камен који је на растојању највише r. Ако ниједан камен који је на растојању највише r не постоји, тада жаба не може стићи до краја. Након што жаба направи први скок, поступак се понавља на исти начин, све док не стигне до краја или док се не дође до ситуације у којој жаба не може да скочи даље.

У сваком кораку тражимо најдаљи камен на који жаба може да скочи и то тако што тражимо први камен на који жаба не може да доскочи (то можемо урадити линеарном, али и бинарном претрагом). Када нађемо камен на који жаба не може да доскочи жаба скаче на камен испред њега (ако жаба може да скочи на сваки камен, тада скаче на последњи, а ако не може да скочи ни на један камен, тада констатујемо да до последњег камена не може да доскочи).

Пример. Размотримо пример када је низ камења $0, 3, 8, 9, 14, 16, 23, 25, 32$ и када је дужина скока $r = 10$. Са почетног камена 0 жаба скаче на камен 9 (то је најдаљи камен на који може да доскочи са камена 0). Након тога скаче на камен 16 (то је најдаљи камен на који може да доскочи са камена 9), након тога на камен 25 (то је најдаљи камен на који може да доскочи са камена 16) и на крају на камен 32 (то је најдаљи камен на који може да доскочи са камена 25).



Слика 9.4: Решење добијено грамзивом стратегијом

Овај алгоритам је типичан похлепни (грамзиви) алгоритам, јер се у сваком кораку узима што је више могуће и тако да се долази и до глобалног оптимума (најмањег броја скокова). Остаје питање како доказати да је ова стратегија коректна.

Доказ коректности. Прво доказујемо да претходни алгоритам увек даје коректно решење тј. решење у складу са условом задатка (жаба креће са првог камена, долази на последњи и у сваком кораку скаче само

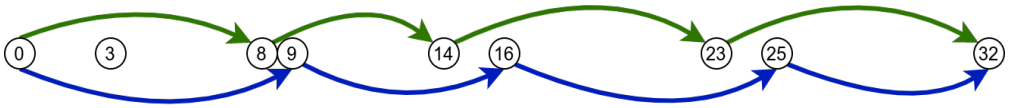
напред, не више од r метара). Заиста, имплементација је одређена тако да је сваки скок који жаба прави у претходном алгоритму скок на камен који је удаљен највише r (то се у алгоритму експлицитно проверава), па смо сигурни да је низ скокова, ако се пронађе, исправан.

Додатно је потребно да докажемо да је и у случају када функција враћа -1, тај одговор коректан тј. да тада заиста није могуће да жаба дође до краја. То се дешава када постоје два камена са координатама x_i и x_{i+1} такви да је $x_{i+1} > x_i + r$. Ако би постојало неко исправно решење, жаба би морала да скочи са неког камена пре i (или са камена i) на неки камен након x_{i+1} (или на камен x_{i+1}), но то је немогуће јер је због сортираности низа растојање сваког таквог пара каменова строго веће од r.

Друго што је потребно доказати је да је решење које се добија стратегијом оптимално. У овом случају то значи да похлепна стратегија даје најмањи могући број скокова.

Доказаћемо да се камен на ком се жаба налази након i корака примене стратегије никада не налази строго иза камена на ком се жаба налази након i корака у оптималном решењу (жаба скаче по стратегији је у сваком кораку или испред или на истом камену у односу на све жабе које достижу циљ у најмањем броју корака). Ово је типичан доказ техником *похлепно решење је увек испред*.

Пример. Размотримо пример када је низ камења $0, 3, 8, 9, 14, 16, 23, 25, 32$ и када је дужина скока $r = 10$. Решење на основу стратегије је $0, 9, 16, 25, 32$. Једно другачије оптимално решење је $0, 8, 14, 23, 32$. Заиста, ни у једном кораку решење на основу стратегије не заостаје за оптималним решењем (а у многим корацима предњачи). Важи $0 = 0$, $9 \geq 8$, $16 \geq 14$, $25 \geq 23$ и $32 = 32$.



Слика 9.5: Решење добијено грамзивом стратегијом приказано доле, предњачи у односу на било које друго оптимално решење приказано горе

Формално, нека је оптимална вредност броја скокова k и и нека је $x_0^*, x_1^*, \ldots, x_{k-1}^*$ сортиран низ x-координата камења на које жаба стаје у једном таквом оптималном решењу. Нека је $x_0, x_1, \ldots, x_{m-1}$ решење добијено на основу наше стратегије. Пошто ниједно решење не може бити боље од оптималног, важи да је $k \leq m$. Индукцијом доказујемо да за свако $i < k$ важи $x_i \geq x_i^*$.

- Базу чини случај $i = 0$ и тада је $x_0 = x_0^*$, јер се жаба у оба случаја налази на почетном камену.
- Под претпоставком да важи $x_i \geq x_i^*$ доказујемо да важи $x_{i+1} \geq x_{i+1}^*$. Ако је $x_i \geq x_{i+1}^*$ (тј. ако је у i-том кораку грамзивом стратегијом жаба већ на камену на који долази у кораку $i + 1$ у оптималном решењу или негде испред тог камена), пошто жаба скаче само напред, важи да је $x_{i+1} > x_i \geq x_{i+1}^*$. У супротном је $x_i < x_{i+1}^*$ (након i корака грамзиве стратегије жаба се налази иза камена на ком се налази након $i + 1$ корака у отпималном решењу). Пошто је оптимално решење коректно, знамо да је $x_{i+1}^* \leq x_i^* + r$. Пошто на основу индуктивне хипотезе знамо да важи $x_i \geq x_i^*$ важи и $x_{i+1}^* \leq x_i + r$. Дакле, жаба са камена x_i може сигурно да доскочи на камен x_{i+1}^* (он се налази испред камена x_i, на растојању мањем од r), а можда може и даље. Пошто грамзива стратегија узима увек најдаљи скок, важи да је $x_{i+1} \geq x_{i+1}^*$.

На основу доказаног важи и да је $x_{k-1} \geq x_{k-1}^*$, међутим, пошто је x_{k-1}^* координата последњег камена, то мора бити и x_{k-1} (па је $m = k$). Дакле, и оптимална стратегија стиже до последњег камена у k корака, па оптимално решење није боље од похлепног.

```
int brojSkokova(const vector<int>& kamenje, int r) {
  // strategija: u svakom koraku skoci sto dalje

  int n = kamenje.size();
  int broj = 0;   // broj skokova
  int kamen = 0;  // kamen na kom se zaba trenutno nalazi
  while (kamen < n - 1) {
    // odredjujemo najdalji kamen na koji mozemo stici od tekuceg
    int noviKamen = kamen;
    // dok god mozes da skocis dalje, skoci dalje
    while (noviKamen + 1 < n &&
```

```
          kamenje[noviKamen + 1] - kamenje[kamen] <= r)
      noviKamen++;
    // zaba ne moze da skoci ni na jedan kamen
    if (noviKamen == kamen)
      return -1;
    // zaba skace na najdalji moguci kamen i uvecavamo broj skokova
    kamen = noviKamen;
    broj++;
  }
  // vracamo broj skokova dobijen ovom strategijom
  return broj;
}
```

Анализа сложености. У имплементацији се користе два показивача (`kamen` и `novi_kamen`), који се кроз низ каменова крећу само унапред, тако да је укупан број корака алгоритма сигурно ограничен двоструком дужином низа и сложеност овог решења је $O(n)$.

Задатак: Шаховске екипе

Шаховска екипа A је позвала на припреме шаховску екипу B. Свака екипа има исти број играча и за сваког играча је познат рејтинг. Екипа домаћина има могућност да одабере парове који ће играти у првом колу (пар чини по један играч из сваке екипе). Ако сваки играч домаћина побеђује госта који има мањи или једнак рејтинг, а губи од госта који има строго већи рејтинг, напиши програм који одређује који је највећи број победа које екипа домаћина (екипа A) може да оствари у првом колу.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 50000$), а затим у наредом реду рејтинзи играча екипе домаћина (природни бројеви) раздвојени размацима, а у наредом реду рејтинзи играча екипе гостију (природни бројеви) раздвојени размацима.

Излаз: На стандардни излаз исписати само један број који представља највећи могући број победа домаћина.

Пример

Улаз	*Излаз*
4 2120 1985 2205 1842 2045 2100 1990 1980	3

Објашњење

Домаћин може да оствари највише три победе. На пример, играч 2205 може да добије играча 2100, играч 2120 може да добије играча 2045, а играч 1985 може да добије играча 1980.

Решење

Задатак можемо решити помоћу неколико различитих грамзивих алгоритама. Преформулишимо мало задатак ради једноставније дискусије. Циљ је да одредимо највећи број парова домаћих и гостујућих играча у којима домаћи играч нема мањи рејтинг од свог пара. Зато ћемо приликом распоређивања обраћати пажњу само на оне партије у којима домаћини побеђују, док ћемо остале партије занемарити (потпуно нам је небитно како су тачно распоређени играчи у партијама у којима домаћини губе).

Најбољи домаћин против најбољег госта којег може да победи

Једна могућност је да се парови формирају тако што се упари k најбољих домаћих играча са k најлошијих гостујућих играча (док остале играче упаримо на произвољан начин). Та стратегија би била коректна, али њена имплементација није тривијална, јер није јасно колико максимално може да буде k тако да у свих k парова домаћини добијају.

Слична стратегија, која се једноставно имплементира је следећа. Ако домаћин може да оствари бар једну победу, њу може да донесе најбољи домаћи играч. Наиме, ако би он изгубио свој меч, увек бисмо неког од играча који је добио меч могли заменити њиме (јер је он бољи од свих играча домаћина) и добити исти број победа. Поставља се питање са којим гостујућем играчем он треба да игра. Циљ нам је да након формирања тог пара преостану што лошији гостујући играчи, да би слабији играчи тима домаћина имали шансе да остваре

победе. Јасно је да у скуп парова у којима домаћини добијају не можемо да укључимо госте који су бољи од тог најбољег домаћег играча (јер њих нико од играча домаћина не може да победи). Добра стратегија је да од преосталих гостујућих играча изаберемо најбољег. Након упаривања најбољег домаћег играча и госта са којим ће он да игра и елиминисања свих гостију бољих од њега, проблем је сведен на проблем истог облика, али мање димензије (смањен је број преосталих домаћих и број гостујућих играча које покушавамо да упаримо тако да домаћини побеђују). Излаз из овог рекурзивног поступка представља случај када су сви домаћи играчи успешно упарени или када међу преосталим гостујућим играчима нема лошијих од најбољег међу преосталим домаћинима.

Доказ коректности. Докажимо и формално коректност овакве грамзиве стратегије.

Решење које претходни алгоритам даје је коректно упаривање и задовољава услове задатка јер се сваки домаћин упарује са гостом која није бољи од њега (то се експлицитно проверава) и није ни упарен ни са једним другим домаћином (јер се након упаривања и домаћин и гост елиминишу из даљег разматрања). Дакле, сигурни смо да заиста постоји коректно упаривање у коме домаћин остварују број пријављен број победа.

Покажимо и да наша стратегија прави оптимални број победа за домаћу екипу. Доказ ће ићи техником размене, тј. тиме што ћемо се показати да се оптимално упаривање може трансформисати у оно добијено грамзивом стратегијом, одржавајући укупан број парова у којима домаћин побеђује (за играче домаћина који побеђују рећи ћемо да су *добитно упарени*). Посматрајмо неко оптимално упаривање. Нека је d_i најбољи домаћин који учествује у њему и нека је g_i гост са којим је он упарен.

Ако он није укупно најбољи домаћин d_s, тада најбољи домаћин сигурно није добитно упарен. Можемо домаћина d_i избацити из добитног упаривања и њему придруженог госта g_i придружити укупно најбољем домаћину d_s (то је могуће јер је $d_s \geq d_i \geq g_i$). Такво упаривање је и даље оптимално (јер се број добитних парова за домаћина није променио).

Нека је g_s гост која би био одабран стратегијом (најбољи гост који није бољи од d_s, тј. најбољи гост за кога важи $d_s \geq g_s$).

- Ако он није део тренутног добитног упаривања, онда госта g_i који тренутно игра са домаћином d_s можемо избацити и заменити гостом g_s (то је могуће јер је $d_s \geq g_s$).
- Ако јесте распоређен тако да губи од неког домаћина d_j, онда можемо направити размену тако да d_s игра са g_s, а d_j са g_i. Докажимо да је ово и даље коректно упаривање. Важи да је $d_s \geq g_s$ и $d_s \geq g_i$. Пошто је g_s најбољи гост кога d_s може да победи, важи да је $g_s \geq g_i$. Зато је $d_j \geq g_s \geq g_i$. Са ове две евентуалне размене добијамо и даље оптималан распоред који је у складу са нашом стратегијом што се тиче првог пара.

Настављајући размене по истом принципу (тј. на основу индуктивног аргумента), упаривање можемо трансформисати у оно формирано нашом стратегијом, задржавајући све време оптималност.

Приликом имплементације, скупове домаћих и гостујућих играча можемо чувати у низовима уређеним у нерастућем редоследу рејтинга и алгоритам можемо реализовати техником два показивача (продискутоваћемо касније и остале могуће варијанте). Низ домаћина обилазимо редом, елемент по елемент, а низ гостију раздвајамо на оне које су елиминисани (оне који су до сада упарени и оне које нису упарени, али су бољи од текућег домаћег играча) и преостале. Одржавамо место почетка низа гостију који још нису обрађени и приликом тражења пара за текућег домаћег играча низ гостију обилазимо од те позиције. Сваког госта или елиминишемо, јер је бољи од текућег домаћег играча или га упарујемо са текућим домаћим играчем и онда их обојицу елиминишемо. Нагласимо да се у имплементацији не морамо враћати на елиминисане госте, јер ако је неки гост бољи од текућег домаћина (најбољег међу преосталим), биће бољи и од свих наредних (преосталих) домаћина.

Анализа сложености. Пошто се оба показивача крећу само у једном смеру и сложеност фазе упаривања је линеарна. Укупним алгоритмом, дакле, доминира сложеност сортирања, па је укупна сложеност $O(n \log n)$.

Могуће је формулисати и грамзиву стратегију која ће бити дуална овој управо описаној. У тој грамзивој стратегији обрађујемо госте у неопадајућем редоследу рејтинга (од лошијих ка бољима) и сваком госту додељујемо најлошијег домаћина који може да га победи. Аналогно претходној, једноставно се доказује да је и та грамзива стратегија такође коректна.

```
int maksBrojPobeda(vector<int>& domaci, vector<int>& gosti) {
  // Ako domacini mogu da ostvare nekih k pobeda, onda tih k pobeda moze
  // da ostvari njihovih k najjacih igraca. Zato domacine obradjujemo u
```

```
  // nerastucem redosledu rejtinga i za svakog redom odredjujemo gosta kojeg
  // moze da pobedi. Za svakog domacina odredjujemo najboljeg gosta kojeg
  // moze da pobedi jer time losijim igracima domace ekipe ostavljamo prostor
  // da pobede nekoga.

  // broj igraca
  int n = domaci.size();
  // zelimo da i domacine i goste obilazimo u nerastucem redosledu rejtinga
  // (od boljih ka losijim)
  sort(begin(domaci), end(domaci), greater<int>());
  sort(begin(gosti), end(gosti), greater<int>());
  // broj pobeda domacih igraca
  int brojPobeda = 0;
  // tekuci indeks domaceg i gostujuceg igraca
  int d = 0, g = 0;
  while (true) {
    // trazimo najboljeg gosta kojeg moze da pobedi domacin na poziciji d
    // goste koji su jaci od trenutno najaceg domacina ne moze da pobedi niko
    // od preostalih domacina, pa ih eliminisemo iz daljeg razmatranja
    while (g < n && domaci[d] < gosti[g])
      g++;
    // ako najaci nerasporedjeni domacin ne moze da pobedi nijednog
    // gosta, ne mozemo povecati broj pobeda
    if (g >= n) break;
    // u suprotnom smo nasli gosta g kojeg uparujemo sa domacinom d
    brojPobeda++;
    // obojicu eliminisemo iz daljeg uparivanja
    g++, d++;
  }
  return brojPobeda;
}
```

Неефикасна имплементација

Скренимо пажњу и на важност ефикасне имплементације. Размотримо решење засновано на дуалној стратегији у којој упарујемо лоше госте. Као што смо видели, у ефикасној имплементацији би приликом преласка на сваког новог госта домаћина требало тражити само међу онима који у ранијим корацима нису елиминисани (било тако што су упарени или тако што је установљено да не могу да победе неког од слабијих гостију). Ако претрагу домаћина сваки пут почињемо из почетка (водећи рачуна о томе да раније упарене домаћине не упарујемо поново, тако што у посебном низу региструјемо оне домаћине које смо већ упарили) добићемо неефикасан алгоритам.

Анализа сложености. Пошто се за сваког од n гостију изнова претражује низ од n домаћина, сложеност најгорег случаја ове имплементације је $O(n^2)$. Нагласимо да није проблем у грамзивој стратегији, већ у њеној лошој имплементацији.

```
int maksBrojPobeda(vector<int>& domaci, vector<int>& gosti) {
  // broj igraca
  int n = domaci.size();
  // Ako je moguce pobediti nekih k gostiju, onda tih k gostiju mogu
  // biti k najlosijih gostiju. Zato goste obradjujemo u rastucem
  // redosledu rejtinga (od losijih ka boljima) i za svakog redom
  // odredjujemo domacina koji moze da pobedi tog gosta, a nije ranije
  // uparen.
  sort(begin(domaci), end(domaci));
  sort(begin(gosti), end(gosti));
  // broj pobeda domacih igraca
  int brojPobeda = 0;
  // da li je domacin trenutno uparen
  vector<bool> zauzet(n, false);
```

```
  // obilazimo sve goste
  for (int g = 0; g < n; g++)
    // trazimo najslabijeg neuparenog domacina koji moze pobediti gosta g
    for (int d = 0; d < n; d++)
      if (!zauzet[d] && domaci[d] >= gosti[g]) {
        brojPobeda++;
        zauzet[d] = true;
        break;
      }

  return brojPobeda;
}
```

Распоређивање сваког домаћина са најлошијим или најбољим гостом

Још једна варијанта наше победничке стратегије (тј. њене дуалне варијанте) обилази све домаћине и госте у неопадајућем редоследу рејтинга и ако домаћин може да победи најслабијег тренутно нераспоређеног госта, онда га упарујемо са њим, а у супротном га упарујемо са најјачим гостом (јер он не може победити никога од преосталих гостију, а мора бити упарен са неким, па је најбоље упарити га са најјачим гостом за ког је сада извесно да нико не може да га победи). На тај начин не добијамо само број победа, већ и ефективно упаривање свих играча у ком се постиже тај максимални број победа.

Ту стратегију можемо имплементирати тако што чувамо скуп преосталих домаћина и гостију, проналазимо најмањи тј. највећи елемент у скупу и уклањамо их. Ако се скуп имплементира преко низа (вектора, листе), добијамо веома неефикасан алгоритам.

Анализа сложености. Овај алгоритам је сложености $O(n^2)$ (јер се и проналажење минимума и максимума и уклањање елемента са дате позиције врши у линеарној сложености).

```
int maksBrojPobeda(vector<int>& domaci, vector<int>& gosti) {
  // broj pobeda domacina
  int brojPobeda = 0;

  // sve dok ima preostalih domacina
  while (domaci.size() > 0) {
    // odredjujemo i uklanjamo najlosijeg domacina
    int najlosijidom = *min_element(begin(domaci), end(domaci));
    domaci.erase(find(begin(domaci), end(domaci), najlosijidom));

    // odredjujemo najlosijeg gosta
    int najlosijigost = *min_element(begin(gosti), end(gosti));
    if (najlosijidom >= najlosijigost) {
      // uparujemo najlosijeg domacina i najlosijeg gosta
      // domacin je bolji i pobedjuje
      brojPobeda++;
      // uklanjamo najlosijeg gosta
      gosti.erase(find(begin(gosti), end(gosti), najlosijigost));
    } else {
      // uparujemo najlosijeg domacina sa najboljim gostom
      int najboljigost = *max_element(begin(gosti), end(gosti));
      // uklanjamo najboljeg gosta
      gosti.erase(find(begin(gosti), end(gosti), najboljigost));
    }
  }

  return brojPobeda;
}
```

Програм постаје много ефикаснији ако употребимо библиотечке колекције које нам пружају ефикаснију имплементацију скупа (која дозвољава ефикасно тражење и уклањање минималног и максималног елемента). У језику C++ можемо употребити мултискупове (јер можда постоји више играча са истим рејтингом) које на

располагању имамо кроз колекцију `multiset`. Пошто је мултискуп уређен итератор `begin()` указује на најмањи елемент, а итератор `prev(end())` на највећи елемент. Уклањање елемента можемо извршити помоћу метода `erase`.

Анализа сложености. Под претпоставком да се операције са мултискупом врше у логаритамској сложености у односу број елемената у мултискупу, овај алгоритам ће бити сложености $O(n \log n)$.

```
int maksBrojPobeda(multiset<int>& domaci, multiset<int>& gosti) {
  int brojPobeda = 0;

  // sve dok ne rasporedimo sve domace igrace
  while (domaci.size() > 0) {
    // rasporedjujemo najboljeg domacina
    int najmanjidom = *domaci.begin();
    domaci.erase(domaci.begin());
    // sa najlosijim gostom ako moze da ga pobedi ili sa najboljim
    // gostom ako ne moze
    int najmanjigost = *gosti.begin();
    if (najmanjidom >= najmanjigost) {
      brojPobeda++;
      gosti.erase(gosti.begin());
    } else {
      gosti.erase(prev(gosti.end()));
    }
  }

  return brojPobeda;
}
```

Ипак најефикасније решење добијамо ако мултискупове представимо сортираним низом, а брисање не вршимо ефективно, већ само чувамо показивач на тренутно необрађеног домаћина, док у скупу гостију необрађене госте чувамо између два показивача.

```
int maksBrojPobeda(vector<int>& domaci, vector<int>& gosti) {
  // broj igraca
  int n = domaci.size();
  // broj pobeda domacina
  int brojPobeda = 0;
  // sortiramo oba tima u neopadajucem redosledu rejtinga (od najslabijih do najboljih)
  sort(begin(domaci), end(domaci));
  sort(begin(gosti), end(gosti));
  // pozicija najlosijeg i najboljeeg nerasporedjenog gosta
  // svi gosti iz intervala [0, gLos) i (gDobar, n) su vec upareni
  int gLos = 0, gDobar = n-1;
  // rasporedjujemo sve domace igrace
  for (int d = 0; d < n; d++) {
    // rasporedjujemo najboljeg domacina sa najlosijim gostom ako moze
    // da ga pobedi ili sa najboljim gostom ako ne moze
    if (domaci[d] >= gosti[gLos]) {
      // uparujemo domacina d i gosta gLos
      brojPobeda++;
      gLos++;
    } else {
      // uparujemo domacina d i gosta gDobar
      gDobar--;
    }
  }
  return brojPobeda;
}
```

Задатак: Зли учитељ

Зли учитељ Нави одлучио је да се освети својим ученицима за ометање часа. Следећи домаћи задатак који им буде задао радиће се у паровима. Учитељ за сваког ученика зна колико сати му је потребно да уради свој део задатка, а време потребно пару ученика да ураде задатак је збир времена та два ученика. Учитељ жели да одреди колико највише сати може да зада за израду домаћег задатка (зато што жели да прикрије своје зле намере и да делује фер) тако да бар један пар ученика не може да стигне да уради задатак, без обзира на то како се ученици поделе у парове.

Улаз: Са стандардног улаза се учитава паран број ученика n ($1 \leq n \leq 10^6$). Затим се учитава n природних бројева који представљају бројеве дана потребних за израду задатка сваког ученика.

Излаз: На стандардни излаз исписати један природан број који представља број дана који учитељ треба да постави као рок за израду задатка.

Пример 1

Улаз	*Излаз*
6 5 7 2 6 4 6	10

Пример 2

Улаз	*Излаз*
6 2 4 1 16 7 11	16

Решење

Потребно је одредити упаривање ученика такво да је највеће време израде домаћег задатка најмање могуће. Другим речима, у низу бројева, потребно је пронаћи оно упаривање које даје најмањи могући (у односу на сва упаривања) максимални збир пара елемената (у односу на све парове). Учитељ може ученицима да остави рок који је за један мањи од тог минималног максималног збира и биће сигуран да неће сви ученици моћи да на време заврше домаћи (и то ће бити најдужи такав рок).

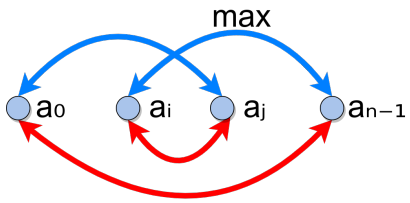
Минимални максимални збир пара можемо одредити грамзивим алгоритмом који упарује најспоријег и најбржег ученика и тиме своди проблем на проблем мање димензије (који се рекурзивно решава на исти начин). Када се исти принцип примени на преостале ученике, закључујемо да је један начин да добијемо оптимално упаривање да се други најбржи упари са другим најспоријим, трећи најбржи са трећим најспоријим и тако даље. Овим смо у потпуности одредили упаривање које може дати минималну максималну разлику и њену вредност одређујемо испитивањем збирова времена свих парова ученика.

Имплементација се може направити итеративно. Прво сортирамо времена потребна да се уради домаћи неопадајуће. Након тога за свако $0 \leq i < \frac{n}{2}$ одређујемо збир $a_i + a_{n-1-i}$ и проналазимо максимум тих збирова.

```
int minimalniMaksimalniZbir(const vector<int>& a) {
  // pravimo sortiranu kopiju svih vremena
  auto as = a;
  sort(begin(as), end(as));
  // broj vremena
  int n = as.size();
  // minimalni maksimalni zbir se dobija u uparivanju u kojem je prvi
  // ucenik uparen sa poslednjim, drugi sa pretposlednjim itd.
  // odredjujemo maksimalni zbir nekog para u tom uparivanju
  int m = 0;
  for (int i = 0; i < n / 2; i++)
    m = max(m, as[i] + as[n - 1 - i]);

  return m;
}
```

Доказ коректности. Нека је низ $a_0, \ldots, a_{n-1}$ неопадајуће сортиран низ времена потребних да се уради домаћи. Тврдимо да постоји упаривање у ком се постиже минимални максимални збир, а у коме су најбржи и најспорији ученик упарени.

Размотримо упаривање у коме је најспорији ученик a_{n-1} упарен са неким учеником a_i, али који није најбржи ($i > 0$). Тада је најбржи ученик a_0 упарен са неким учеником a_j, али који није најспорији ($j < n - 1$). Ако сада направимо размену тако да упаримо ученике a_0 и a_{n-1} (најбржег и најспоријег) и ученике a_i и a_j, максимум се може само смањити. Остали парови остају непромењени, па је потребно посматрати само упаривања ова 4 ученика.



Слика 9.6: Стање пре размене и након размене. Збир $a_i + a_{n-1}$ је већи или једнак од $a_0 + a_j$, $a_0 + a_{n-1}$ и $a_i + a_j$, тако да се након размене максимум не може повећати

Важи да је $\max(a_0 + a_j, a_i + a_{n-1}) = a_i + a_{n-1}$, јер је $a_0 \leq a_i$ и $a_j \leq a_{n-1}$. Пошто је $a_0 \leq a_i$, важи да је $a_0 + a_{n-1} \leq a_i + a_{n-1}$, а пошто је $a_j \leq a_{n-1}$, важи да је $a_i + a_j \leq a_i + a_{n-1}$. Дакле, оба броја $a_0 + a_{n-1}$ и $a_i + a_j$ су мања или једнака од вредности $a_i + a_{n-1} = \max(a_0 + a_j, a_i + a_{n-1})$, па важи $\max(a_0 + a_{n-1}, a_i + a_j) \leq \max(a_0 + a_j, a_i + a_{n-1})$.

Дакле, ако се у неком оптималном упаривању изврши размена тако да се први и последњи елемент упаре, упаривање остаје оптимално (максимум се не може смањити, јер је полазно упаривање оптимално, а на основу доказаног не може се ни повећати). Према томе, постоји оптимално упаривање у коме су најбржи и најспорији ученик упарени.

Анализа сложености. Након сортирања које се врши у времену $O(n \log n)$, упаривање се одређује у времену $O(n)$.

Задатак: Разломљени ранац

У једној продавници се продају слаткиши (бомбонице, чоколадице, кексићи) "на меру". Постоји n врста слаткиша и знамо да i-тог слаткиша има w_i грама, по укупној цени од v_i динара. Продавница је у оквиру своје промоције организовала наградну игру у којој је наградила једну своју муштерију тако да на поклон може да узме све слаткише који стају у ранац носивости W грама. Написати програм који одређује највећу вредност слаткиша које срећни добитник може да узме.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 10^5$). У n наредних редова се налазе по два цела броја w_i и v_i (цели бројеви између 1 и 100). У последњем реду налази се носивост ранца W (цео број између 1 и 10^9).

Излаз: На стандардни излаз исписати највећу вредност коју срећни добитник може да покупи, заокружену на две децимале.

Пример 1

Улаз	*Излаз*

`3`
`10 60`
`20 100`
`30 120`
`50`
	`240.00`

Објашњење

Максимална вредност се постиже ако се узме 10 килограма слатикша чија је укупна цена 60 динара, 20 килограма слаткиша чија је укупна цена 100 динара и 20 килограма слаткиша чија је укупна цена 120 динара (пошто се узима две трећине масе, на њему се добија вредност 80 динара).

Пример 2

Улаз

```
3
10 60
20 100
30 120
80
```

Излаз

```
280.00
```

Решење

Пошто срећни добитник жели да покупи што већу вредност слаткиша, јасно је да треба да почне узимање оних слаткиша који су највреднији тј. чија је цена по граму највећа. Ако том врстом слаткиша може да испуни цео ранац (тј. да не покупи целокупну расположиву количину тог слаткиша), најбоље му је да то и уради. У супротном, покупиће целокупну количину тог слаткиша, а затим ће преосталу носивост ранца попунити преосталим слаткишима, по истом принципу (овде се може уочити индуктивно-рекурзивна конструкција).

Јасно је да претходни грамзиви алгоритам даје увек **коректно** решење, јер се ни за један предмет не узима већа количина од доступне и укупна маса узетих слаткиша не превазилази масу ранца.

Дужни смо још да докажемо да наша грамзива стратегија доводи до оптималног решења.

Доказ коректности. Користићемо метод размене. Претпоставимо да смо сортирали слаткише тако да важи

$$\frac{v_0}{w_0} \geq \frac{v_1}{w_1} \geq \ldots \geq \frac{v_{n-1}}{w_{n-1}}.$$

Свако решење је одређено узетом масом (у грамима) сваког од слаткиша. Нека је решење на основу стратегије одређено низом маса $(s_0, s_1, \ldots, s_{n-1})$, при чему за свако $0 \leq i < n$ важи $0 \leq s_i \leq w_i$ и $\sum_{i=0}^{n-1} s_i \leq W$, где је W укупна носивост ранца. Претпоставимо да је оптимално решење одређено низом маса $(o_0, o_1, \ldots, o_{n-1})$, при чему за свако $0 \leq i < n$ важи $o_i \leq w_i$ и $\sum_{i=0}^{n-1} o_i \leq W$.

Пошто је o оптимално решење, мора да важи да је $\sum_{i=0}^{n-1} o_i = \sum_{i=0}^{n-1} s_i$. Наиме, знамо да ће када се слаткиши узимају на основу стратегије укупна маса узетих слаткиша бити или једнака носивости ранца или укупној расположивој маси свих слаткиша (када је она већа или једнака од носивости ранца). Ако би се у оптималном решењу садржала мања маса слаткиша од тога, могли бисмо масу неког слаткиша да повећамо и тако да добијемо још боље решење, што је у контрадикцији са претпоставком оптималности.

Претпоставимо да стратегија даје решење s које је различито од оптималног решења o. Нека је j прва позиција на којој се низови s и o разликују (за свако $0 \leq i < j$ важи да је $s_i = o_i$). Грамзива стратегија узима редом целокупне расположиве масе свих предмета, све до последњег узетог предмета где се узима максимална маса која стаје у ранац, тако да је једина могућност да је $o_j < s_j$ и то за $j < n - 1$ (због услова $\sum_{i=0}^{n-1} o_i = \sum_{i=0}^{n-1} s_i$). Размотримо решење o' које добијамо тако што уместо масе o_j предмета j узмемо масу s_j (то сигурно можемо, јер је $s_j \leq w_j$). Тиме смо повећали укупну масу у ранцу за $s_j - o_j$ и зато укупну масу узетих предмета након позиције j морамо да смањимо за $s_j - o_j$ (то можемо јер је $\sum_{i=0}^{n-1} o_i = \sum_{i=0}^{n-1} s_i$ и $\sum_{i=0}^{j-1} o = \sum_{i=0}^{j-1} s_i$). Докажимо да ово решење не може бити лошије од оптималног. То је интуитивно јасно јер смо повећали масу скупљег предмета, на рачун смањења масе масе јефтинијих предмета, но покажимо то и формално. Вредност у ранцу се повећала за $(s_j - o_j) \cdot \frac{v_j}{w_j}$ и умањила за $\sum_{k=j+1}^{n-1} s_k \frac{v_k}{w_k}$, где је $\sum_{k=j+1}^{n-1} s_k = s_j - o_j$. Докажимо да повећање вредности не може бити мање него смањење вредности тј. да се овом променом укупна вредност у ранцу није смањила. Важи да је

$$\sum_{k=j+1}^{n-1} c_k \frac{v_k}{w_k} \leq \sum_{k=j+1}^{n-1} c_k \frac{v_{j+1}}{w_{j+1}} = \frac{v_{j+1}}{w_{j+1}} \sum_{k=j+1}^{n-1} c_k = \frac{v_{j+1}}{w_{j+1}}(s_j - o_j) \leq \frac{v_j}{w_j}(s_j - o_j).$$

Овом променом смо, дакле, направили решење o' чија је вредност једнака оптималној (јер ниједно решење не може бити боље од оптималног решења o, а управо смо доказали да решење o' није лошије од њега), а које се поклапа са стратегијским решењем s на једној позицији више него полазно оптимално решење o. Настављајући поступак на исти начин добићемо решење које је оптимално и једнако је s.

Анализа сложености. Сложеношћу доминира фаза сортирања предмета по јединичној цени, па је сложеност целог алгоритма $O(n \log n)$. Након сортирања, узимање предмета и израчунавање максималне вредности врши се у сложености $O(n)$.

```
double razlomljeniRanac(const vector<int>& cena,
                        const vector<int>& kolicina,
                        int nosivostRanca) {
  // broj vrsta slatkiša
```

```
  int n = cena.size();
  // vektor u kome čuvamo jedinične cene i količine svih slatkiša
  vector<pair<double, int>> jedinicnaCenaIKolicina(n);
  for (int i = 0; i < n; i++) {
    // jedinična cena slatkiša slatkiša broj i
    double jedinicnaCena = (double)cena[i] / (double)kolicina[i];
    jedinicnaCenaIKolicina[i] =
      make_pair(jedinicnaCena, kolicina[i]);
  }
  // sortiramo opadajuće na osnovu jedinične cene
  sort(begin(jedinicnaCenaIKolicina), end(jedinicnaCenaIKolicina),
       greater<pair<double, int>>());
  // ukupna vrednost slatkiša koja se može poneti u rancu
  double ukupnaVrednost = 0.0;
  // obrađujemo slatkiše po opadajućoj vrednosti jedinične cene
  for (int i = 0; nosivostRanca > 0 && i < n; i++) {
    // čitamo jediničnu cenu i količinu slatkiša broj i
    double jedinicnaCena = jedinicnaCenaIKolicina[i].first;
    int kolicina = jedinicnaCenaIKolicina[i].second;
    // uzimamo što više, ali smo ograničeni raspoloživom količinom
    // i preostalom nosivišću ranca
    int uzetaKolicina = min(kolicina, nosivostRanca);
    // preostala nosivost ranca
    nosivostRanca -= uzetaKolicina;
    // ažuriramo ukupnu vrednost
    ukupnaVrednost += uzetaKolicina * jedinicnaCena;
  }
  // vraćamo ukupnu vrednost uzetih slatkiša
  return ukupnaVrednost;
}
```

Задатак: Распоред активности

У једном кабинету се суботом одржава обука програмирања. Сваки наставник је написао термин у ком жели да држи наставу (познат је сат и минут почетка и сат и минут завршетка часа). Одреди како је могуће направити распоред часова тако да што више наставника буде укључено.

Улаз: Са стандардног улаза се учитава прво број n (укупан број наставника, $1 \leq n \leq 50000$), а затим у n наредних редова по четири броја раздвојена размацима који представљају сат и минут почетка тј. завршетка часа (претпоставити да је завршетак увек иза почетка).

Излаз: На стандардни излаз исписати највећи број наставника који могу да одрже своје часове.

Пример

Улаз	*Излаз*
7 8 15 9 20 10 45 11 30 11 20 12 45 9 30 12 40 10 20 11 20 12 00 13 00 11 30 13 30	3

Објашњење

Могу се одржати, на пример, часови од 8:15 до 9:20, затим час од 10:20 до 11:20 и на крају од 11:30 до 13:30.

Решење

Сваки час можемо представити паром бројева који представљају број минута од претходне поноћи до почетка и до краја часа (већ приликом учитавања сате и минуте можемо превести само у минуте).

Исцрпна претрага

Наивно решење се заснива на испитивању свих могућих подскупова скупа часова који су такви да се сви часови могу одржати (никоја два часа из тог скупа се не секу). Пресек два интервала постоји ако и само ако је каснији почетак часа после ранијег краја часа. Генерисање свих подскупова можемо вршити рекурзивно. Тај поступак је пописан у задатку Сви подскупови.

Анализа сложености. С обзиром на велики број подскупова које треба испитати ово решење је веома неефикасно (сложеност му је експоненцијална $O(2^n)$, где је n укупан број часова).

Грамзиви приступ

Ефикасно решење проблема се може добити грамзивим приступом. Постоји неколико грамзивих стратегија које је логично размотрити, међутим, неке од њих неће гарантовати оптималност пронађеног решења.

Један приступ може бити онај у коме прво распоређујемо час који први почиње. На слици је приказан контра-пример, који показује да се већ са три часа на тај начин може добити распоред који није оптималан.

Један приступ може бити онај у коме тежимо да распоредимо часове који кратко трају, са идејом да на тај начин остављамо више простора да се у слободним терминима одрже други часови. На слици је приказан контра-пример који показује да се већ са три часа на тај начин може добити распоред који није оптималан.

Слика 9.7: Пример на коме стратегија која распоређује час који први почиње и пример на коме стратегија која распоређује час који је најкраћи даје неоптимално решење

Једна грамзива стратегија која даје оптимално решење је следећа. Од свих нераспоређених часова бирамо онај који се најраније завршава и који се може одржати (не сече се са до сада одржаним часовима, тј. почиње након завршетка претходног часа). Интуитивно, таквим избором остављамо што већу могућност за распоређивање накнадних часова. Овим добијамо једну рекурзивну конструкцију.

- Ако је скуп часова празан, нема часова који се могу распоредити.
- У супротном бирамо час који се најраније завршава, одбацујемо часове који се са њим секу и рекурзивном правимо распоред за преостале часове.

Доказ коректности. Докажимо да је ова рекурзивна формулација коректна, тако што ћемо да докажемо да увек постоји исправан, оптималан распоред (распоред са највише часова) у ком учествује час c_0 који се први завршава. Претпоставимо да је O неки исправан, оптималан распоред. Ако у њему учествује час c_0, онда је O је тај тражени распоред. Ако не учествује, онда претпоставимо да је o_0 час у O који се први завршава. Сви други часови у O почињу након завршетка часа o_0. Заиста, пошто је O исправан, ниједан час у o_i се не сече са o_0, ако би неки почињао пре o_0, тада би морао и да се заврши пре o_0, што је немогуће, јер се од свих часова у O час o_0 први завршава. Час c_0 се не завршава касније него час o_0, јер је он час који се први завршава од свих часова. Дакле, час c_0 се не сече ни са једним часом у O (осим евентуално са o_0). Када заменимо c_0 и o_0, добијамо исправан, оптималан распоред (број часова се није променио) који садржи c_0.

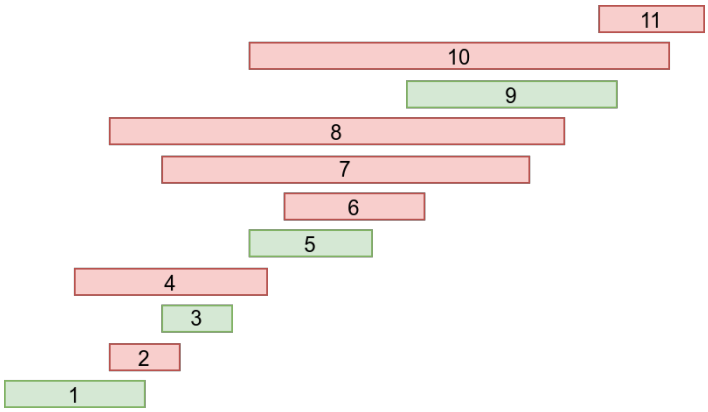
Дакле, јасно је да ће нас избор часа c_0 водити до неког оптималног решења. Такође је јасно да у том решењу не може да учествује ниједан час који се сече са c_0. Остатак часова бирамо рекурзивно, па коректност алгоритма следи на основу индуктивног аргумента (претпостављамо да рекурзивни позив коректно проналази оптимални распоред у преосталом скупу часова).

Техника која је употребљена у претходном доказу назива се техника замене тј. размене, јер се од неког произвољног оптималног распореда заменом добио распоред који наша грамзива стратегија бира.

Алгоритам се може формулисати и итеративно. Часове можемо сортирати неопадајуће на основу времена њиховог завршетка и обилазити их у том редоследу. Први час сигурно бирамо да буде одржан. Редом прола-

зимо кроз наредне часове и ако текући час почиње након последњег одабраног часа, бирамо га, а у супротном га прескачемо.

Пример. На слици су приказани часови сортирани по времену завршетка. Грамзивом стратегијом се прво одржава час 1, затим се час 2 прескаче (јер се преклапа са 1), па се затим час 3 одржава (јер се не преклапа са 1, јер је десно од њега), па се час 4 прескаче (јер се преклапа са 3), па се час 5 одржава (јер се не преклапа са 3, па самим тим ни са 1, јер је десно од њих), па се часови 6, 7 и 8 прескачу (јер се преклапају са часом 5), па се час 9 одржава (јер се не преклапа са 5, па самим тим ни са 3 ни са 1, јер је десно од њих) и на крају се прескачу часови 10 и 11 (јер се преклапају са часом 9). Максималан број часова који се могу одржати без преклапања је 4, међутим, часови 1, 3, 5 и 9 нису једино решење. Могуће је, на пример, одржати часове 1, 3, 6 и 11.



Слика 9.8: Резултат примене грамзиве стратегије

Доказ коректности. Испишимо и доказ коректности итеративне варијанте алгоритма.

Формално, претпоставимо да је $O = [o_1, o_2, \ldots, o_k]$, низ часова који представља неко исправно оптимално решење и докажимо да он садржи исти број часова као и распоред који би одабрала наша стратегија. Претпоставимо да су часови o_1 до o_k сортирани неопадајуће по редоследу њиховог завршетка. Пошто се сви ти часови могу одржати, између њих нема преклапања и сваки наредни почиње након завршетка претходног.

Нека је $S = [s_1, s_2, \ldots, s_{k'}]$ низ часова који би био одабран нашом грамзивом стратегијом. Пошто је O оптималан, S не може да садржи више часова од њега важи да је $k \leq k'$.

Докажимо прво да постоји оптималан распоред такав да за свако $1 \leq i \leq k'$ важи да је $o_i = s_i$. Тај распоред можемо добити поступним изменама почетног распореда O. Претпоставимо да постоји неко $1 \leq i \leq k'$ тако да је $o_i \neq s_i$ (у супротном је полазни распоред тај тражени). Нека је i први такав индекс тј. нека за свако $1 \leq j < i$ важи да је $o_j = s_j$. Покажимо да се заменом часа o_i часом s_i у низу O добија такође распоред који је исправан (он је свакако оптималан јер се број часова не мења). Покажимо прво да се s_i не завршава касније него o_i.

- Заиста, ако је $i = 0$, тада наша стратегија бира s_0 који се први завршава он не може да се завршава касније него o_0.
- Ако је $i > 0$, тада знамо да се o_i мора да почиње после $o_{i-1} = s_{i-1}$, међутим, наша стратегија за s_i бира онај час који почиње након s_{i-1} који се први завршава, па се s_i ни у овом случају не може завршавати касније него o_i.

Ако постоје часови у O пре часа o_i, они остају непромењени и час s_i се не преклапа са њима (јер га је стратегија бира тако да почиње након завршетка часа $s_{i-1} = o_{i-1}$). Пошто се s_i не завршава касније него o_i он се сигурно не преклапа ни са једним часом из O који иде после o_i (јер сви они почињу и завршавају се након краја часа o_i). Дакле, s_i се не преклапа ни са једним часом из O (осим евентуално са o_i, који је у склопу размене уклоњен) и распоред добијен заменом је исправан.

Наставком овог процеса замена стићи ћемо до жељеног оптималног распореда O таквог да за свако $1 \leq i \leq k'$ важи $o_i = s_i$.

Докажимо сада да није могуће да важи да је $k' < k$. Ако би важило да је $k' < k$, тада би важило да час $o_{k'+1}$ припада а не припада S. Пошто је O исправан, то би био час који би почињао после завршетка часа $o_{k'} = s_{k'}$.

Међутим, није могуће да такав час постоји, јер би он почињао и завршавао се после часа s'_k, што значи да би наша грамзива стратегија морала да га одабере, што је у контрадикцији са тим да се он не налази у S. Дакле, важи да је $k' \geq k$.

Пошто је $k' \geq k$ и $k' \leq k$, важи да је $k' = k$, па наша грамзива стратегија бира исти број часова који је у неком (па и сваком) оптималном распореду. Дакле, стратегијом се добија један исправан, оптималан распоред.

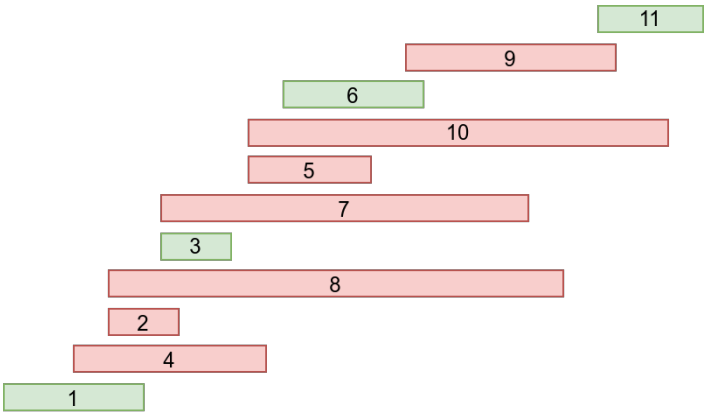
Пример. Илуструјмо технику размене на којој лежи претходни доказ, тако што ћемо објаснити како да се од распореда $1, 3, 6, 11$ који је оптималан, али није у складу са нашом стратегијом добије распоред $1, 3, 5, 9$ који јесте у складу са нашом стратегијом.

- Први час где се распореди разликују је час 5 тј. 6. Заменом часа 6, часом 5 добија се распоред $1, 3, 5, 11$, који је такође исправан. Заиста, час 6 се не сече ни са часовима 1 и 3, ни са часом 11, јер је распоред $1, 3, 6, 11$ исправан. Час 5 се не сече са часовима 1 и 3, јер га у супротном грамзива стратегија не би одабрала. Међутим, час 5 се не може завршавати после часа 6, јер се час 6 не сече са часовима 1 и 3, а од свих часова који се не секу са часовима 1 и 3, стратегија бира онај који се најраније завршава. Дакле, 5 се не завршава касније од 6, па пошто се 6 не сече са 11 и пошто је 11 потпуно десно у односу на 6, ни 5 се не сече са 11. Дакле, распоред $1, 3, 5, 11$ је исправан.
- У наредном кораку се упоређују распореди $1, 3, 5, 11$ и $1, 3, 5, 9$ (који наша стратегија препоручује). Први час где се распореди разликују је час 9 тј. 11. Заменом часа 11, часом 9 добија се распоред $1, 3, 5, 9$, који стратегија препоручује.

Дакле, произвољни оптимални распоред смо размена трансформисали у распоред који препоручује наша стратегија, што значи да је број часова које наша стратегија распореди за одржавање оптималан.

Рецимо да постоји и дуално решење у којем се бира онај час који последњи почиње и часови се обилазе уназад, по нерастућем редоследу њиховог почетка.

Пример. На слици су приказани часови сортирани по редоследу почетка. Прво се одржава час 11, који последњи почиње, затим се прескаче час 9 који се са њим преклапа, затим се одржава час 6, након чега се прескачу часови 10, 5 и 7 који се са њим преклапају, затим се одржава час 3, прескачу се часови 8, 2 и 4 који се са њим преклапају и на крају се одржава час 1.



Слика 9.9: Резултат примене дуалне грамзиве стратегије

```
// casove predstavljamo uredjenim parovima (pocetak, kraj), u minutima
typedef pair<int, int> cas;

inline int pocetakCasa(const cas& c) {
  return c.first;
}

inline int krajCasa(const cas& c) {
  return c.second;
}

cas napraviCas(int pocSat, int pocMin, int krajSat, int krajMin) {
```

```
  return make_pair(pocSat*60 + pocMin, krajSat*60 + krajMin);
}

// ucitava se niz casova
vector<cas> ucitajCasove() {
  int n;
  cin >> n;
  vector<cas> casovi(n);
  for (int i = 0; i < n; i++) {
    int pocSat, pocMin, krajSat, krajMin;
    cin >> pocSat >> pocMin >> krajSat >> krajMin;
    casovi[i] = napraviCas(pocSat, pocMin, krajSat, krajMin);
  }
  return casovi;
}

// maksimalni broj casova koji se mogu odrzati tako da nema
// preklapanja medju rasporedjenim casovima
int maksBrojCasova(vector<cas>& casovi) {
  // broj casova
  int n = casovi.size();

  // sortiramo casove na osnovu vremena zavrsetka
  sort(begin(casovi), end(casovi),
       [](const cas& a, const cas& b) {
         return krajCasa(a) < krajCasa(b);
       });

  // broj odrzanih casova
  int brojOdrzanihCasova;
  // kraj poslednjeg odrzanog casa
  int kraj;

  // rasporedjujemo prvi cas
  brojOdrzanihCasova = 1;
  kraj = krajCasa(casovi[0]);

  // analiziramo ostale casove u redosledu zavrsetka
  for (int i = 1; i < n; i++)
    // ako se tekuci cas ne preklapa sa poslednjim rasporedjenim
    if (pocetakCasa(casovi[i]) >= kraj) {
      // on se odrzava
      brojOdrzanihCasova++;
      kraj = krajCasa(casovi[i]);
    }

  return brojOdrzanihCasova;
}
```

Задатак: Распоред са најмањим бројем учионица - опис решења

За сваки од n часова познато је време почетка и завршетка. Када се у некој учионици заврши један час, истог тренутка у њој може да почне неки други час. Напиши програм који одређује минималан број учионица потребан да се сви часови одрже.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^5$), а затим у n наредних редова подаци о n часова (сат и минут почетка и сат и минут завршетка, при чему је време почетка строго мање од времена завршетка).

Излаз: На стандардни излаз исписати тражени број потребних учионица.

Пример

Улаз	*Излаз*

`6`
`8 0 8 45`
`10 0 10 45`
`9 0 9 45`
`8 30 9 15`
`10 45 11 30`
`10 30 11 15`
	`2`

Објашњење

Један могући распоред наведих 6 часова у две учионице је следећи:

```
Учионица 1          Учионица 2

8:00-8:45 Час 1     8:30-9:15 Час 4
9:00-9:45 Час 3     10:30-11:15 Час 6
10:00-10:45 Час 2
10:45-11:30 Час 5
```

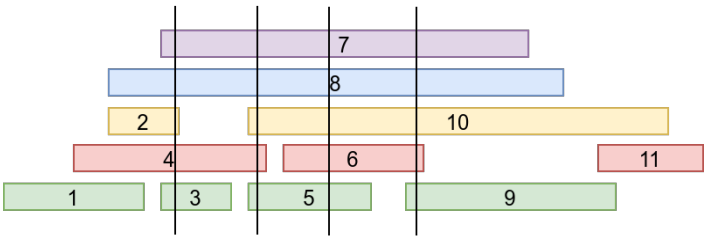
Решење

Иако донекле делује слично задатку Распоред активности, решење овог задатка захтева другачију стратегију.

Пошто се захтева да се сви часови одрже, обилазићемо их у одређеном редоследу и сваки час ћемо придруживати некој од слободних учионица. У тренутку у ком почиње неки час и у коме нема више слободних учионица које су раније употребљаване, отвараћемо нову учионицу у коју ћемо распоређивати тај час.

Знамо да је најмањи број учионица сигурно већи или једнак највећем броју часова који се истовремено одржавају у неком тренутку. У наставку ћемо доказати да је минималан број учионица увек једнак том броју и да се распоред може направити ако се часови обилазе у растућем редоследу њиховог почетка.

Пример. На слици је приказан један могући распоред часова. Учионице се налазе једна изнад друге (сви часови у истој учионици су приказани на истој висини). Вертикалним линијама су означени тренуци у којима су све учионице попуњене тј. тренуци у којима постоји 5 часова који се преклапају. Због тога је јасно да није могуће направити распоред са мање од 5 учионица.



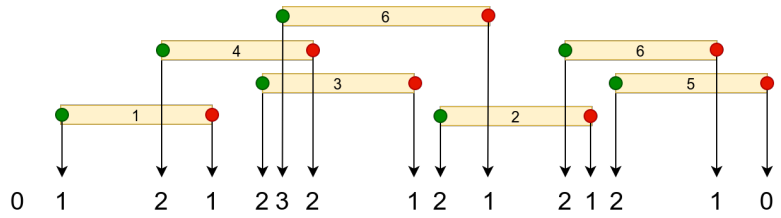
Слика 9.10: Распоред часова

Један могући редослед обраде часова је растући (неопадајући) редослед њихових почетака.

Доказ коректности. Докажимо да је наша стратегија оптимална. Стратегија је таква да је једини разлог да се нова учионица отвори то да су све раније отворене учионице већ попуњене у тренутку када почиње текући час, тј. да постоји неки час (рецимо $[s_j, f_j)$) који се преклапа са свим часовима који су распоређени и у тренутку његовог почетка се одржавају у тренутних d отворених учионица. Пошто су часови сортирани на основу времена почетка, свих тих d часова почиње пре тренутка s_j и завршава се након тренутка s_j (јер трају у тренутку s_j). То значи да у тренутку s_j сигурно постоји $d+1$ часова (тих d већ распоређених и час $[s_j, f_j)$) који се у том тренутку одржавају, па број учионица мора бити бар $d+1$. Дакле, ако се на основу стратегије резервише нова учионица, сигурни смо да је то неопходно. Због тога знамо да када наша стратегија направи распоред са неким бројем учионица, сигурни смо да није било могуће направити распоред са мањим бројем учионица, што значи да је направљени распоред оптималан.

Приметимо да је у овом доказу одређена граница квалитета решења (распоред не може бити у мање учионица него што је број часова у највећој групи часова који се одржавају у неком тренутку) и затим је показано да похлепно решење достиже ту границу, па је због тога оптимално.

Ако нас занима само број потребних учионица, а и конкретан распоред, тада можемо направити веома једноставну имплементацију која одржава број тренутно отворених учионица и обилази све значајне временске тренутке (почетке и завршетке часова) редом. На крају часа смањује се број заузетих учионица, а на почетку часа повећава се број учионица. Тражени резултат се добија као максимална вредност бројача заузетих учионица.



Слика 9.11: Обилазак свих карактеристичних тачака уз ажурирање бројача

Потребно је још обратити пажњу на специјалан случај када се неки часови завршавају у истом тренутку у ком други часови почињу. Елегантно решење је да у датом временском тренутку прво обрадимо све часове који се завршавају у том тренутку и ослободимо учионице, а затим да обрадимо часове који почињу у том тренутку (тако нове часове распоређујемо у управо ослобођене учионице, што је у складу са тим да је време трајања сваког часа полуотворени интервал $[s, f)$).

Анализа сложености. За n часова постоји $2n$ карактеристичних тренутака. Њихово сортирање се врши у времену $O(n \log n)$. Обилазак сортираног низа тренутака и ажурирање бројача се затим врши у времену $O(n)$. Сложеношћу, дакле, доминира сортирање и укупна сложеност је $O(n \log n)$.

```
struct Vreme {
  int minut;
  bool pocetak;
};

Vreme napraviVreme(int sat, int min, bool pocetak) {
  Vreme v;
  v.minut = 60*sat + min;
  v.pocetak = pocetak;
  return v;
}

// ucitava spisak casova sa standardnog ulaza
vector<Vreme> ucitajCasove() {
  int n;
  cin >> n;
  vector<Vreme> vremena;
  vremena.reserve(2*n);
  for (int i = 0; i < n; i++) {
    int pocSat, pocMin, krajSat, krajMin;
    cin >> pocSat >> pocMin >> krajSat >> krajMin;
    vremena.push_back(napraviVreme(pocSat, pocMin, true));
    vremena.push_back(napraviVreme(krajSat, krajMin, false));
  }
  return vremena;
}

// funkcija rasporedjuje casove u ucionice, vraca minimalni potrebni
// broj ucionica i svakom casu dodeljuje broj ucionice
int minUcionica(vector<Vreme>& vremena) {
```

```
  sort(begin(vremena), end(vremena),
       [](const Vreme& v1, const Vreme& v2) {
         return v1.minut < v2.minut || (v1.minut == v2.minut && !v1.pocetak && v2.pocetak);
       });
  // trenutan broj zauzetih ucionica
  int brojUcionica = 0;
  // maksimalni broj zauzetih ucionica u nekom trenutku
  int maksBroj = 0;
  for (const Vreme& v : vremena)
    if (v.pocetak)
      maksBroj = max(++brojUcionica, maksBroj);
    else
      brojUcionica--;
  return maksBroj;
}
```

Ако желимо да направимо конкретан распоред, имплементација је мало компликованија. Први корак у имплементацији је веома једноставан – учитавамо све часове у низ и сортирамо их на основу почетног времена. Кључни корак у другој фази је одређивање учионице у коју може бити смештен текући час. За све до тада отворене учионице знамо времена завршетка часова у њима. Можемо пронаћи учионицу у којој се час најраније завршава и проверити да ли је могуће да у њу распоредимо текући час. Ако јесте, њој ажурирамо време завршетка часа, а ако није, онда знамо да су све учионице заузете (јер се час који се први завршава још није завршио), па морамо отворити нову учионицу. Да бисмо ефикасно могли да нађемо учионицу у којој се час најраније завршава, све учионице можемо чувати у реду са приоритетом сортираном по времену завршетка часа у свакој од учионица. Ако тај ред није празан и ако је време завршетка часа у учионици на врху реда мање или једнако времену почетка текућег часа, време завршетка часа у тој учионици ажурирамо на време завршетка текућег часа (најлакше тако што ту учионицу избацимо из реда и поново је додамо са ажурираним временом). У супротном у ред додајемо нову учионицу којој је време завршетка часа постављено на време завршетка текућег часа (број учионице је за један већи од дотадашњег броја учионица у реду).

Број отворених учионица је увек једнак броју елемената реда. Наиме, елемент у ред додајемо само када отварамо нову учионицу јер су све до тада отворене учионице у реду заузете, док у супротном само мењамо елемент који је био на врху реда новим (часови који су се завршили а нису замењени новим часовима у истој учионици остају у реду).

Анализа сложености. Укупна сложеност алгоритма је $O(n \log n)$ – и у фази сортирања и у фази распоређивања (јер се читање и избацивање минимума, као и убацивање новог елемента у ред са приоритетом врши у времену $O(\log n)$). Када бисмо уместо реда са приоритетом користили обичан низ и у њему стално тражили минимум, сложеност најгорег случаја би порасла на $O(n^2)$.

```
struct Cas {
  // redni broj casa (njegov jedinstveni identifikator)
  int broj;
  // minut pocetka i kraja casa
  int pocetak, kraj;
  // redni broj ucionice u kojoj se cas odrzava
  int ucionica;
};

Cas napraviCas(int broj, int pocSat, int pocMin, int krajSat, int krajMin) {
  Cas c;
  c.broj = broj;
  c.pocetak = pocSat*60 + pocMin;
  c.kraj = krajSat*60 + krajMin;
  return c;
}

struct Ucionica {
  // broj ucionice (njen jedinstveni identifikator)
  int broj;
```

```
  // minut od kog je ucionica slobodna
  int slobodnaOd;
};

Ucionica napraviUcionicu(int slobodnaOd, int broj) {
  Ucionica u;
  u.broj = broj;
  u.slobodnaOd = slobodnaOd;
  return u;
}

// ucitava spisak casova sa standardnog ulaza
vector<Cas> ucitajCasove() {
  int n;
  cin >> n;
  vector<Cas> casovi(n);
  for (int i = 0; i < n; i++) {
    int pocSat, pocMin, krajSat, krajMin;
    cin >> pocSat >> pocMin >> krajSat >> krajMin;
    casovi[i] = napraviCas(i, pocSat, pocMin, krajSat, krajMin);
  }
  return casovi;
}

// funkcija rasporedjuje casove u ucionice, vraca minimalni potrebni
// broj ucionica i svakom casu dodeljuje broj ucionice
int minUcionica(vector<Cas>& casovi) {
  // sortiramo časove na osnovu vremena njihovog početka
  sort(begin(casovi), end(casovi),
       [](const Cas& c1, const Cas& c2) {
         return c1.pocetak < c2.pocetak;
       });

  // red sa prioritetom u kom su trenutno zauzete ucionce slozene po
  // vremenu zavrsetka casa
  struct PorediUcionice {
    bool operator()(const Ucionica& u1, const Ucionica& u2) {
      return u1.slobodnaOd > u2.slobodnaOd;
    }
  };
  priority_queue<Ucionica, vector<Ucionica>, PorediUcionice> redUcionica;
  for (Cas& c : casovi) {
    int brojUcionice;
    if (redUcionica.empty() || redUcionica.top().slobodnaOd > c.pocetak)
      brojUcionice = redUcionica.size() + 1;
    else {
      brojUcionice = redUcionica.top().broj;
      redUcionica.pop();
    }
    c.ucionica = brojUcionice;
    redUcionica.push(napraviUcionicu(c.kraj, brojUcionice));
  }

  // sortiramo časove na osnovu rednog broja
  sort(begin(casovi), end(casovi),
       [](const Cas& c1, const Cas& c2) {
         return c1.broj < c2.broj;
       });
```

```
  return redUcionica.size();
}
```

Задатак: Распоред са најмањим закашњењем

Радник треба да заврши n послова. За сваки од послова које треба завршити познато је трајање у минутима и рок до којег се посао мора завршити. Радник почиње да ради у тренутку 0, започете послове ради док их не заврши и не може да ради два посла истовремено. Ако се посао не заврши у предвиђеном року, муштерије ће бити незадовољне. Што је кашњење веће, веће је и незадовољство. Ако је нека муштерија баш јако незадовољна, престаће да наручује послове у овој фирми. Зато послове треба распоредити тако да, ако је то могуће, не постоји ниједна муштерија која је баш јако незадовољна (боље је имати више мало незадовољних муштерија, него једну много незадовољну). Зато се укупан квалитет распореда послова одређује на основу највећег направљеног закашњења – потребно је да оно буде што мање. Написати програм који одређује оптималан распоред, тј. највеће кашњење неког посла у том распореду.

Улаз: Са стандардног улаза се у првом реду уноси број n ($1 \leq n \leq 10^5$), затим у наредном реду трајања послова t_i (позитивни цели бројеви) и затим у наредном реду рокови за завршетак послова r_i (позитивни цели бројеви).

Излаз: На стандардни излаз исписати квалитет најбољег распореда (највеће направљено закашњење у том распореду).

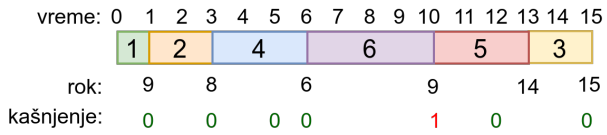
Пример

Улаз	*Излаз*

`6`
`1 2 2 3 3 4`
`9 8 15 6 14 9`
	`1`

Објашњење

На слици је приказан један распоред послова код којег је највеће кашњење једнако 1 (само се посао број 6, који треба да се заврши у 9 завршава један минут касније, у 10).

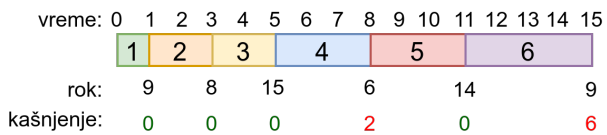


Слика 9.12: Један оптимални распоред послова

Решење

Задатак је могуће решити помоћу грамзивог алгоритма. Кључно питање је у ком редоследу треба заказивати послове.

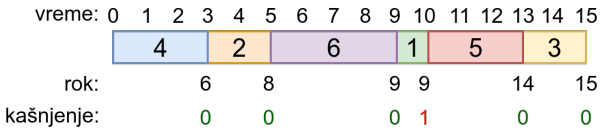
На пример, једна могућност би могла бити да се прво обављају кратки послови. Међутим, пример на слици (који одговара оном у тексту задатка) показује да ова стратегија није добра, јер се добије распоред који је много лошији од оптималног.



Слика 9.13: Стратегија која прво распоређује кратке послове даје неоптималан распоред

Слични контрапримери би се могли направити и за распореде у којима се послови распоређују тако да се дугачки послови прво обављају.

Оптимално решење се добија ако се послови заказују у редоследу рока њиховог завршетка, при чему сваки посао почиње одмах након што се претходни завршио (не прави се пауза ни један једини минут).



Слика 9.14: Распоред добијен грамзивом стратегијом – послови су заказани у неопадућем редоследу истека њихових рокова

Имплементација је једноставна. Након сортирања послова у неопадајућем редоследу истека рокова, израчунава се закашњење за сваки посао и одређује се максимално закашњење.

```
int n;
cin >> n;
vector<pair<int, int>> poslovi(n);
for (int i = 0; i < n; i++) {
  int trajanje;
  cin >> trajanje;
  poslovi[i].second = trajanje;
}
for (int i = 0; i < n; i++) {
  int rok;
  cin >> rok;
  poslovi[i].first = rok;
}
sort(begin(poslovi), end(poslovi));

int maksKasnjenje = 0;
int vreme = 0;
for (auto& posao : poslovi) {
  vreme += posao.second;
  int kasnjenje = max(vreme - posao.first, 0);
  maksKasnjenje = max(maksKasnjenje, kasnjenje);
}
cout << maksKasnjenje << endl;
```

Анализа сложености. Алгоритмом доминира фаза сортирања која се извршава у сложености $O(n \log n)$. Максимално кашњење се затим одређује једним проласком кроз низ у сложености $O(n)$.

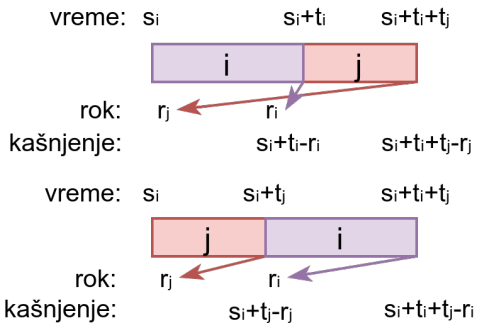
Доказ коректности. Докажимо комбинацијом технике размене и технике границе да се грамзивом стратегијом добија оптималан распоред.

Ако су два посла i и j такви да је рок завршетка првог r_i пре рока завршетка другог r_j (тј. важи $r_i < r_j$), а да је почетак првог посла s_i заказан након почетка другог посла s_j (тј. важи $s_i > s_j$), рећи ћемо да су они распоређени наопако.

Претпоставимо да је O неки оптимални распоред.

Ако у том распореду постоје два наопако распоређена посла, тада у распореду сигурно постоје и два наопако распоређена узастопна посла. Наиме, претпоставимо да су послови i и j распоређени наопако. Пошто је рок завршетка посла j испред рока завршетка посла i, приликом обиласка послова између i и j у редоследу њиховог заказаног почетка, није могуће да рок њиховог завршетка стално расте и неопходно је да се приликом преласка са једног посла на следећи рок завршетка смањи. Два узастопна посла код којих се то деси су сигурно наопако распоређена.

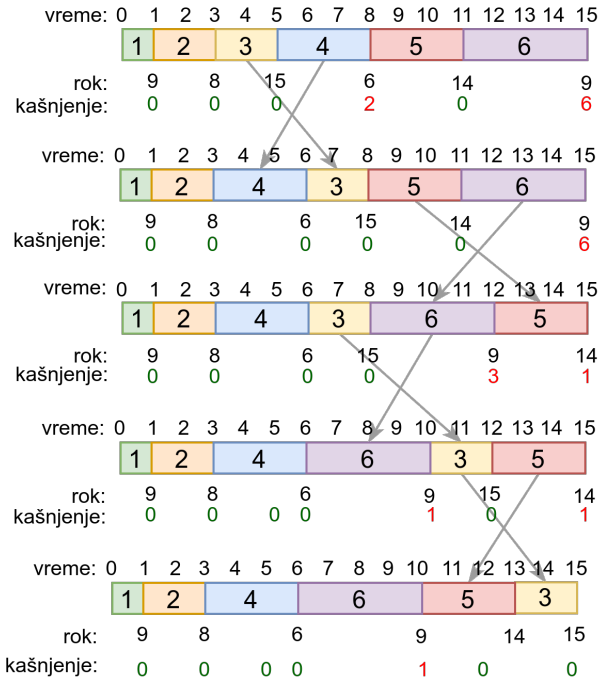
Два суседна посла се могу разменити тако да се остали послови не дирају. Докажимо да се разменом два суседна наопако распоређена посла максимално кашњење не може повећати. Пошто се остали послови не мењају разменом два суседна наопако распоређена посла, потребно је размотрити једино њихова кашњења. Нека посао i почиње у у тренутку s_i, траје t_i минута и нека му је рок завршетка r_i. Тада посао j почиње у тренутку $s_i + t_i$, нека траје t_j минута и нека му је рок завршетка r_j. Пошто су они распоређени наопако важи да је $r_i > r_j$ (и да је $s_i < s_i + t_i$).



Слика 9.15: Размена наопаких послова – посао j се померио напред, па се приближио свом року. Посао i се померио ка назад, па му се кашњење повећало, али он сигурно мање касни него што је каснио посао , јер му је рок касније.

Пре размене кашњење посла i је $\max(s_i + t_i - r_i, 0)$ док је кашњење посла j једнако $\max(s_i + t_i + t_j - r_j, 0)$. Након размене њихова кашњења постају $\max(s_i + t_j + t_i - r_i, 0)$ и $\max(s_i + t_j - r_j, 0)$. Посао j је померен унапред (за t_i минута) и његово кашњење се тиме само могло смањити (јер је $\max(s_i + t_j - r_j, 0) \leq \max(s_i + t_i + t_j - r_j, 0)$). Посао i је померен уназад, међутим, пошто је $r_i > r_j$, важи да је $\max(s_i + t_i + t_j - d_i, 0) \leq \max(s_i + t_i + t_j - r, 0)$, па посао i не касни више него што је раније каснио посао j.

Претходно описаном разменом се број наопако распоређених послова смањује за 1. Размене можемо понављати све док не стигнемо до распореда у коме нема наопако распоређених послова.



Слика 9.16: Размене које доводе до оптималног распореда – даљим размена ма би се могло стићи и до распореда у ком нема наопаких послова, али се кашњење не може смањити тако да буде мање од 1

Лако се доказује да сви распореди у којима нема наопако распоређених послова и нема пауза имају исто максимално кашњење (разменом узастопних послова са истим роком не мења се максимално кашњење), а наш похлепни алгоритам гради један баш такав распоред, то значи да ће се максимално кашњење нашег распореда поклапати са оптималним (достиже се теоријска граница) и наш распоред ће такође бити оптималан.

Напомена. Напоменимо и да наизглед једноставна модификација текста задатка даје савим другачији и рачунски много тежи проблем. Наиме, ако бисмо уместо највећег кашњења покушавали да смањимо збир свих кашњења, формулисана грамзива стратегија не би давала оптимално решење. Наиме, разменом суседних наопаких послова не може се гарантовати да ће се укупно кашњење смањити, већ само да ће се максимално кашњење смањити. Може се показати да је реформулисани проблем NP-комплетан проблем и да нема познато полиномијално решење.

Задатак: Мали поштар

Јовица зарађује џепарац тако што доноси пакете својим комшијама. Поделу креће од своје куће и потребно је да пакете разнесе у друге куће распоређене дуж улице и да се врати назад у своју кућу. За сваку кућу познато је растојање од почетка улице. Најкраћи пут би прешао ако би пакете сложио тако да их редом дели комшијама дуж улице. Пошто Јовица жели да буде у доброј физичкој форми, он током поделе пакета трчи и жели да пакете уреди тако да пређе што већи пут. Напиши програм који одређује највећи пут који може да пређе.

Улаз: Са стандардног улаза се уноси број кућа у које треба донети пакете (међу њима се налази и Јовицина кућа), а затим и растојања тих кућа од почетка улице.

Излаз: На стандардни излаз исписати највеће растојање које Јовица може прећи током поделе пакета.

Пример 1

Улаз	*Излаз*
5 7 3 6 10 2	24

Објашњење

Постоји више начина да Јовица претрчи 24 дужне јединице. На пример, ако је његова кућа на позицији 3, он може да редом обилази куће 3, 7, 2, 10, 6, 3.

Пример 2

Улаз

```
7
3 5 11 4 2 17 9
```

Излаз

```
56
```

Решење

Груба сила

Решење грубом силом подразумева да се провере сви могући редоследи обиласка n кућа, тј. свих $n!$ пермутација датих бројева и да се утврди која од њих даје највећу могућу вредност пређеног пута. Пермутације се могу обићи коришћењем техника приказаних у задатку Све пермутације.

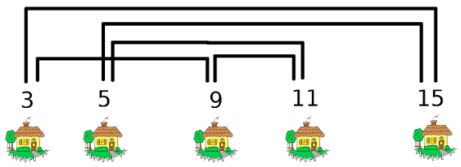
Анализа сложености. Алгоритам заснован на провери свих пермутација је изразито неефикасан, његова сложеност је $O(n!)$ и може се применити само на веома, веома мале улазе (практично, само за $n \leq 10$ програм може да реши задатак у датом временском ограничењу).

```
int zbirApsRazlika(const vector<int>& a) {
  int zbir = 0;
  for (size_t k = 1; k < a.size(); k++)
    zbir += abs(a[k] - a[k-1]);
  zbir += abs(a[a.size()-1] - a[0]);
  return zbir;
```

```
}

int najduziPut(vector<int>& a) {
  int maks = 0;
  do {
    maks = max(zbirApsRazlika(a), maks);
  } while(next_permutation(begin(a), end(a)));
  return maks;
}
```

Грамзива стратегија

Интуитивно нам је јасно да ће се пуно претрчати ако се стално трчи са једног на други крај улице. Један грамзиви приступ је да се прво обиђе крајња лева кућа, па затим крајња десна, па друга слева, па друга здесна и тако "цик-цак".



Слика 9.17: Цик-цак обилазак: креће се из 3, затим се иде у 15, па у 5, па у 11, па у 3 и на крају назад у 3

Доказ коректности. Докажимо да је ово грамзиво решење исправно.

За дати распоред $x_0, \dots x_{n-1}$ одређујемо суму апсолутних вредности разлика елемената тј. покушавамо да максимизујемо суму

$$|x_0 - x_1| + |x_1 - x_2| + ... + |x_{n-2} - x_{n-1}| + |x_{n-1} - x_0|.$$

Сваки елемент се у суми јавља тачно два пута. У зависности од међусобног односа бројева неки елементи ће бити узети са знаком $+$, а неки са знаком $-$, и то тако да се тачно n елемената узима са знаком $+$ и тачно n елемената узима са знаком $-$. Циљ нам је да елементи који се узимају са знаком $+$ буду што већи, а да ови са знаком $-$ буду што мањи. Распоред који иде цик-цак постиже да се за знаком $+$ узме $\frac{n}{2}$ већих елемената низа, а са знаком $-$ узме $\frac{n}{2}$ мањих елемената низа (ако их је непаран број, тада се средњи узима једном са знаком $-$, а једном са знаком $+$). Заиста, ако, на пример, имамо 6 елемената $a_0 \leq a_1 \leq a_2 \leq a_3 \leq a_4 \leq a_5$, цик-цак распоред даје вредност

$$|a_0 - a_5| + |a_5 - a_1| + |a_1 - a_4| + |a_4 - a_2| + |a_2 - a_3| + |a_3 - a_0|,$$

што је једнако

$$(a_5 - a_0) + (a_5 - a_1) + (a_4 - a_1) + (a_4 - a_2) + (a_3 - a_2) + (a_3 - a_0),$$

тј.

$$2 \cdot (a_5 + a_4 + a_3) - 2 \cdot (a_2 + a_1 + a_0)$$

Јасно је да се не може добити више од овога, а ово се, видели смо, увек може експлицитно достићи баш цик-цак распоредом.

У овом примеру смо коректност грамзиве стратегије доказали тако што смо израчунали теоријско ограничење функције која се оптимизује и затимо смо доказали да се грамзивом стратегијом достиже то теоријско ограничење.

Елементе низа можемо експлицитно сортирати, па затим распоредити елементе у нови низ по цик-цак редоследу и за тај нови низ израчунати збир апсолутних вредности разлика суседних елемената.

Анализа сложености. Сложеношћу алгоритма доминира фаза сортирања, чија је сложеност $O(n \log n)$. Распоређивање елемената у помоћни низ и израчунавање дужне врши се у сложености $O(n)$.

```
int najduziPut(vector<int>& a) {
  int n = a.size();
  sort(begin(a), end(a));
  vector<int> b(n);
  int i = 0, j = n-1;
  for (int k = 0; k < n; k++)
    if (k % 2 != 0)
      b[k] = a[i++];
    else
      b[k] = a[j--];

  return zbirApsRazlika(b);
}
```

Помоћни низ се може веома једноставно избећи у имплементацији.

```
int najduziPut(vector<int>& a) {
  int n = a.size();
  sort(begin(a), end(a));
  int i = 0, j = n-1;
  int zbir = 0;
  bool paran = true;
  while (i < j) {
    zbir += abs(a[j]-a[i]);
    if (paran)
      i++;
    else
      j--;
    paran = !paran;
  }
  zbir += abs(a[0] - a[i]);
  return zbir;
}
```

Ако пажљиво размотримо доказ коректности, примећујемо да распоред у коме идемо од прве до последње, па до друге, затим претпоследње куће итд., није једини који даје максимални пређени пут. Довољно је само да наизменично узимамо елементе из прве и друге половине низа (у било ком редоследу). Ово инспирише још једноставнији алгоритам за израчунавање траженог максимума (саберемо $\frac{n}{2}$ бројева са почетка, одузмемо $\frac{n}{2}$ бројева са краја низа и помножимо са 2).

```
int najduziPut(vector<int>& a) {
  int n = a.size();
  sort(begin(a), end(a));

  int zbir = 0;
  for (int i = 0; i < n/2; i++) {
    zbir += a[n-1-i];
    zbir -= a[i];
  }
  return zbir * 2;
}
```

Задатак: Исплата са посебним новчићима

У Србији се користе апоени од 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000 и 5000 динара. Написати програм који формира дати износ динара од што мањег броја апоена (новчаница и новчића) и исписује употребљен број апоена.

Улаз: Са стандардног улаза се учитава један цео број између 0 и 100 000.

Излаз: На стандардни излаз написати најмањи број апоена потребних да се исплати учитани износ новца.

Пример

Улаз	*Излаз*
243	5

Објашњење

$243 = 200 + 20 + 20 + 2 + 1$.

Решење

Један директан начин да се проблем реши је да се испитају сва могућа разлагања датог износа на збирове састављене од ових бројева и да се пронађе онај збир који има најмањи број новчића (једноставности ради, занемаримо разлику између новчића и новчаница). Решење овог типа би се могло засновати на динамичком програмирању комбинованом са одсецањем током претраге. Такво решење приказано је у задатку Исплата са најмање новчића.

Доказ коректности није тежак, али је овај приступ био прилично компликован и неефикасан. Наиме специфичности апоена омогућавају да се задатак реши много ефикасније.

```
// najmanji broj novčića potreban da se naplati iznos S
// kada su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(const vector<int>& v, int n, int S) {
  vector<int> dp(S+1);
  // iznos 0 se naplaćuje sa 0 novčića
  dp[0] = 0;
  // računamo minimalni broj novčića za sve ostale iznose
  for (int s = 1; s <= S; s++) {
    // minimalni broj novčića da se naplati iznos s (pretpostavljamo
    // da iznos nije moguće naplatiti)
    dp[s] = INF;
    // razmatramo sve mogućnosti za poslednji novčić
    for (int i = 0; i < n; i++)
      // proveravamo da li je iznos s moguće naplatiti novčićem i
      if (v[i] <= s)
        // ažuriramo minimum ako je to potrebno
        dp[s] = min(dp[s], dp[s-v[i]] + 1);
  }
  // vraćamo rezultat za iznos S
  return dp[S];
}
```

И без формалног математичког објашњења, сваки продавац у продавници и на пијаци зна да се оптимално решење добија тако што се у сваком тренутку враћа највећи апоен који је мањи или једнак од тренутног износа и након тога се исти принцип примењује на преостали износ све док се не врати цео кусур (у питању је, дакле, грамзива индуктивно-рекурзивна конструкција). Ово решење је веома ефикасно, лако се имплементира, међутим, доказ његове коректности није нимало очигледан.

Наиме, постојање новчића од 4 динара би покварило ситуацију. 8 динара би се могло добити од два новчића од 4 динара, док би грамзива стратегија употребила три новчића (од 5, 2 и 1 динар). Дакле, доказ коректности мора да укључи анализу конкретних апоена који су у оптицају и мале промене ових апоена могу да утичу на то да описани приступ даје или не даје увек оптимално решење.

Докажимо сада коректност. Једноставности ради, претпоставићемо да су у оптицају само апоени од 1, 2, 5, 10, 20 и 50 динара (за веће новчиће доказ иде по истом принципу). Методом размене доказаћемо горње границе

броја новчића од свих апоена у оптималном решењу.

- Оптимално решење не може да садржи више од једног новчића од 1 динар. Када би постојала макар два новчића од 1 динар, они би могли бити замењени једним новчићем од 2 динара, чиме би се број употребљених новчића смањио, што је у контрадикцији са претпоставком да је полазно решење оптимално. Потпуно аналогно се доказује да оптимално решење не може садржати ни више од једног новчића од 10 динара.
- Даље, оптимално решење не може садржати више од два новчића од 2 динара. Наиме, ако би садржало бар три новчића од 2 динара, они би могли бити замењени једним новчићем од 1 и једним новчићем од 5 динара, чиме би се добило мање решење, што је у контрадикцији са претпоставком да је полазно решење оптимално. Потпуно аналогно, оптимално решење не може да садржи ни више од два новчића од 20 динара.
- На крају, у оптималном решењу не може бити више од једног новчића од 5 динара. Наиме, ако би постојала бар два, она би могла бити замењена једним новчићем од 10 динара чиме би се добило мање решење, што је контрадикција.
- У решењу није могуће ни да истовремено постоје два новчића од 2 динара и новчић од 1 динара, јер би се сви они могли заменити са једним новчићем од 5 динара, што је опет контрадикција. Аналогно важи и за новчиће од 10 и 20 динара.

Узевши у обзир претходна ограничења, размотримо максималне износе са оптималним бројем новчића, који се могу добити коришћењем само одређених скупова новчића. Испоставиће се да су максимални износи увек за један мањи од првог већег апоена.

- Новчићи од 1 динар могу да направе највише износ од 1 динара (јер се смеју појавити само једном).
- Новчићи од 1 и 2 динара могу да направе највише износ од 4 динара (јер може бити највише два новчића од 2 динара и у том случају се не сме користити и новчић од 1 динар).
- Новчићи од 1, 2 и 5 динара могу да направе највише износ од 9 динара (јер не може бити више од једног новчића од 5 динара, два новчића од 2 и једног новчића од 1 динара, а ако има два новчића од 2 динара, не сме се јавити и новчић од 1 динара).
- Слично, новчићи од 1, 2, 5 и 10 динара могу да направе највише износ од 19 динара (јер се 10 динара може јавити само једном, а од 1, 2 и 5 се може направити највише 9).
- Новчићи од 1, 2, 5, 10 и 20 могу да направе највише износ од 49 динара (јер 1, 2 и 5 могу да направе највише 9, како смо објаснили, а пошто се 10 динара јавља највише једном, а 20 динара највише два пута, али не сва три таква заједно, од 10 и 20 се може направити највише 40).

Докажимо сада да се за сваки позитиван износ у оптималном решењу мора налазити највећи новчић који је мањи или једнак од тог износа (све наведене констатације се односе само на оптимална решења).

- За износ 1 јавља се само новчић 1.
- Износи између 2 и 4 динара морају да садрже новчић 2. Наиме, не може да се јави новчић од 5 динара, само од новчића од 1 динар може да се направи највише 1 динара, па за износе од 2 до 4 динара мора да се јави бар један новчић од 2 динара.
- Износи између 5 и 9 динара морају да садрже новчић од 5 динара. Наиме, не могу да садрже новчић од 10 динара, а пошто се од новчића од само 1 и 2 динара може направити највише 4 динара, за износе од 5 до 9 динара мора да се употребни новчић од 5 динара.
- Износи између 10 и 19 динара морају да садрже новчић 10. Наиме, 20 динара не може да се јави, а помоћу новчића од 1, 2 и 5 динара највише се може направити 9 динара.
- Износи између 20 и 49 динара морају да садрже бар један новчић од 20 динара. Наиме, не могу да садрже 50, а само са 1, 2, 5 и 10 се може добити највише 19.
- Износи преко 50 динара морају да садрже новчић од 50 динара. Наиме, само са новчићима од 1, 2, 5, 10 и 20 је могуће направити само 49 динара.

Дакле, успели смо да за сваки износ пронађемо новчић који оптимално решење мора да садржи, чиме онда успевамо да смањимо димензију проблема и да до решења дођемо директно, без било какве претраге и испробавања разних могућности.

```
int minBrojApoena(int iznos) {
  int brojApoena = 0;
  vector<int> apoeni{5000, 2000, 1000, 500, 200, 100, 50, 20, 10, 5, 2, 1};
  while (iznos > 0) {
    for (int apoen : apoeni)
      if (iznos >= apoen) {
        iznos -= apoen;
        brojApoena++;
        break;
      }
  }
  return brojApoena;
}
```

Задатак: Кодирање текста са што мање битова

Потребно је кодирати текст низом битова (нула и јединица), тако да је свако слово кодирано неким бинарним кодом. При том, кодирање мора да буде такво да није могуће да кôд неког слова буде префикс кода неког другог слова (јер би тада дешифровање било отежано, а можда не би ни било једнозначно). Напиши програм који одређује најмањи број битова потребних да се кодира текст.

Напомена: није неопходно користити исти број битова за кодирање сваког карактера.

Улаз: Прва линија стандардног улаза садржи број n ($1 \leq n \leq 256$) различитих карактера које треба кодирати. Након тога свака наредна линија садржи карактер и његов број појављивања у тексту.

Излаз: На стандардни излаз треба исписати тражени минималан број битова.

Пример

Улаз	*Излаз*

`5`
`a 10`
`b 28`
`c 7`
`d 14`
`e 19`
	`173`

Објашњење

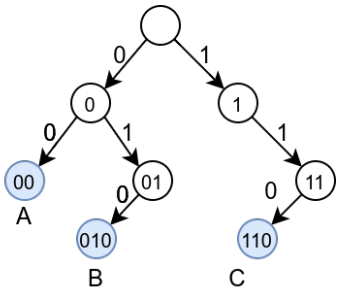
На пример, карактер `a` можемо кодирати кодом `011`, карактер `b` кодом `11`, карактер `c` кодом `010`, карактер `d` кодом `00` и карактер `e` кодом `10`.

Решење

Алгоритам који приказујемо познат је под именом *Хафманово кодирање* и представља један од класичних примера грамзивих (похлепних алгоритама).

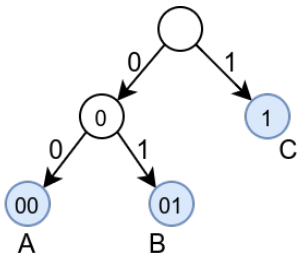
Један од услова да се обезбеди једнозначно декодирање је да ниједна кодна реч не буде префикс некој другој. Такви се кодови називају *бинарни префиксни кодови*. За дати скуп карактера и фреквенције њиховог појављивања желимо да конструишемо оптимални бинарни префиксни кôд (онај код којег је број битова $\sum_{i=0}^{n-1} f_i b_i$ потребан за запис целог текста најмањи, где је f_i број појављивања карактера c_i у тексту, а b_i број битова додељених том карактеру).

Бинарним префискним кодовима једнозначно одговарају бинарна дрвета. Кôд сваког чвора се добија обиласком дрвета – корак на лево додаје симбол 0 на код, а корак на десно симбол 1, при чему се карактерима које је потребно кодирати додељују кодови искључиво у листовима дрвета (зато није могуће да је кôд било ког карактера префикс кода неког другог карактера).



Слика 9.18: Пример бинарног префиксног кода. Слово A се кодира битовима 00, слово B битовима 010, а слово C битовима 110.

Размотримо прво структуру дрвета које одговара бинарном префиксном коду. Први важан закључак је да у оптималном префиксном коду сви унутрашњи чворови морају имати оба детета. У супротном се кôд може скратити тако што се унутрашњи чвор уклони и замени својим дететом. У претходном примеру, дрво можемо трансформисати тако што унутрашњи чвор `01` заменимо својим дететом и тако карактеру `B` доделимо код `01` уместо `010`. Слично, чвор `11`, а затим и чвор `1` можемо заменити својом децом и тако карактеру `C` можемо доделити кôд `1`.



Слика 9.19: Пример бинарног префиксног кода. Слово A се кодира битовима 00, слово B битовима 01, а слово C битом 1.

Интуиција нам говори да је пожељно карактерима који се појављују често додељивати краће кодове. Зато ће карактери који се јављају ређе имати дуже кодове.

Доказ коректности. Ово није тешко формално доказати. Претпоставимо да карактер c_i има фреквенцију појављивања f_i и да се кодира са b_i битова, а да карактер c_j има фреквенцију појављивања f_j и да се кодира са b_j битова, при чему је $f_i \geq f_j$, док је $b_i \leq b_j$. Ако се замене кодови та два карактера, укупан потребан број битова за кодирање текста се не може повећати. Пошто се осталим карактерима кодови не мењају, број битова зависи само од ова два карактера. У првом случају он је једнак $f_i b_i + f_j b_j$, а у другом је једнак $f_i b_j + f_j b_i$. Ако посматрамо разлику $(f_i b_j + f_j b_i) - (f_i b_i + f_j b_j)$ добијамо израз $f_i(b_j - b_i) + f_j(b_i - b_j)$ тј. $(f_i - f_j)(b_j - b_i)$. Пошто су оба чиниоца ненегативна и полазна разлика је ненегативна, па се укупан потребан број битова овом трансформацијом није могао повећати (ако је разлика позитивна, онда се број битова смањио).

Наредно важно тврђење је то да постоји оптимално дрво у којем се два карактера који се најређе јављају, налазе као два суседна листа (сина истог оца) и то најудаљенија од корена.

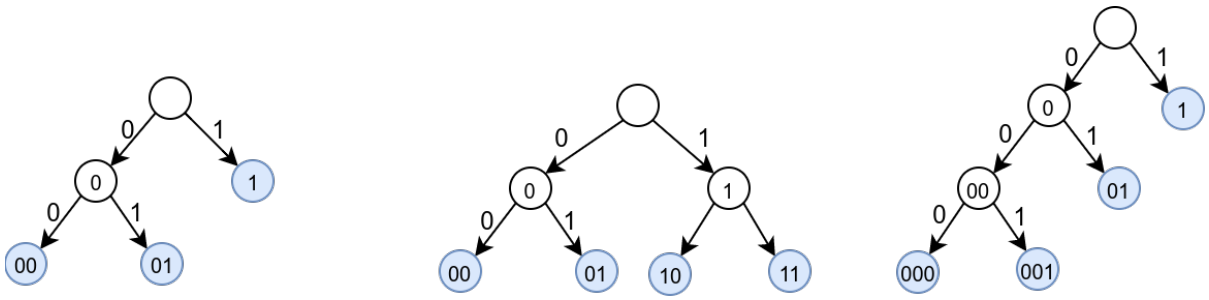
Доказ коректности. Нека су c_1 и c_2 два карактера са најмањим фреквенцијама. Размотримо неко оптимално дрво и његова два листа најудаљенија од корена. Нека су њима придружени карактери c_1' и c_2'. Можемо заменити c_1 и c_1' и заменити c_2 и c_2'. Структура дрвета остаје идентична, па се након размене и даље добија исправан бинарни префиксни кôд. Пошто су након ове размене кодови замењени тако да су чешћи карактери добили краће кодове, на основу претходно доказаног тврђења укупан број битова за кодирање текста се није могао повећати, па је добијени кôд и даље оптималан.

Пример. Покушајмо да кроз неколико примера стекнемо мало интуицију о проблему и његовом решењу.

Ако азбука има један карактер или два карактера, ситуација је потпуно јасна – ти карактери се могу кодирати са по једним битом.

Ако постоје три карактера, онда није могуће да постоје два карактера који се кодирају са по једним битом, јер би тада кôд трећег карактера садржао као префикс кôд неког од та два карактера (он би морао да почне било нулом, било јединицом). Дакле, у тој ситуацији један карактер мора бити кодиран једним битом, а друга два карактера са по два бита. Одлука који карактер треба да буде кодиран са једним битом је једноставна – то мора да буде карактер који се најчешће јавља.

Ако постоје четири карактера, тада постоји могућност да сви буду кодирани са по два бита (2-2-2-2) или да један од њих буде кодиран са једним битом, један са два бита и два са по три бита (1-2-3-3). Анализом могућих облика дрвета показује се да би сва друга кодирања била неповољнија.



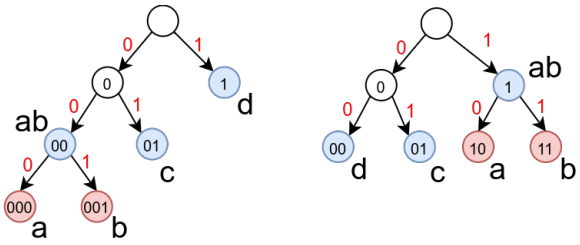
Слика 9.20: Једнина могућа структура оптималног дрвета са три листа и једине две могуће структуре оптималног дрвета са четири листа, ако се занемари могућност симетричног пресликавања, тј. размене левог и десног детета

Поново је јасно да карактери који се често појављују треба да буду кодирани са мање, а они који се ретко појављују треба да буду кодирани са више битова, међутим, нејасно је од чега зависи да ли је боље употребити кодирање 2-2-2-2 или 1-2-3-3. Размотримо два примера.

```
frekvencije                    frekvencije
a b c d                        a b c d
1 2 5 6                        3 4 5 6

kodiranje 2-2-2-2              kodiranje 2-2-2-2
1·2 + 2·2 + 5·2 + 6·2 = 28     3·2 + 4·2 + 5·2 + 6·2 = 36

kodiranje 1-2-3-3              kodiranje 1-2-3-3
1·3 + 2·3 + 5·2 + 6·1 = 25     3·3 + 4·3 + 5·2 + 6·1 = 37
```

У првом случају је повољније кодирање 1-2-3-3, а у другом 2-2-2-2. У оба случаја два најређа карактера `a` и `b` се кодирају са истим бројем битова (два или три), карактер `c` се кодира са два бита, док се најчешћи карактер `d` кодира са два или са једним битом. Кодирање 1-2-3-3 је повољније ако је $1 \cdot f_d + 3 \cdot (f_a + f_b) \leq 2 \cdot f_d + 2 \cdot (f_a + f_b)$, а кодирање 2-2-2-2 у супротном (јер се у првом случају `a` и `b` кодирају са три бита, а `d` са једним, док се у другом случају `a`, `b` и `d` кодирају са по два бита). Дакле, веома битан податак нам је збир фреквенција два карактера који се најређе јављају (збир фреквенција $f_a + f_b$ карактера `a` и `b`) и тај збир треба поредити са фреквенцијама других карактера (пошто се ти карактери увек кодирају са истим бројем битова, битан нам је само збир њихових фреквенција, а не њихов појединачан однос).

Када бисмо из оптималног дрвета са четири листа уклонили два листа која представљају најређе карактере, добили бисмо дрво оптималне структуре са три листа. Узевши у обзир симетрије тј. могућност размене левог и десног детета, оно увек има јединствени облик. Његови листови ће бити карактери `c` и `d` и трећи чвор испод којег ће бити постављени касније карактери `a` и `b` (чвор родитељ листова `a` и `b` у крајњем дрвету). Јасно је да, како год да су друга два чвора распоређена, карактеру `c` морају бити придружена два бита (ако би му се придружио 1 бит, карактеру `d` би била придружена 2, па би се њиховом разменом добио бољи код). Дакле, или ће карактеру `d` бити придружен 1 бит, а родитељу карактера `a` и `b` 2 бита или ће карактеру `d` бити придружена 2 бита, а родитељу карактера `a` и `b` 1 бит. Ако је кодирање 1-2-3-3 повољније, тада важи и да је $1 \cdot f_d + 2 \cdot (f_a + f_b) \leq 2 \cdot f_d + 1 \cdot (f_a + f_b)$ (одузимањем вредности $f_a + f_b$ у почетној неједнакости), а то је неједнакост чијом се анализом одређује како треба распоредити чвор `d` и родитељски чвор чворова `a` и `b` у дрвету са три листа — у првом случају би се карактер `d` кодирао са једним битом, а родитељски чвор, када бисмо га третирали као лист и када би му се придружио неки карактер, би се кодирао са два бита, док би се у другом случају карактер `d` кодирао са два бита, а родитељски чвор са једним битом.



Слика 9.21: Питање положаја листова a и b своди се на питање оптималног распоређивања листова c, d и "листа" ab, са фреквенцијом $f_a + f_b$

Дакле, основа Хафмановог алгоритма лежи у идеји да се уместо два карактера са најмањим фреквенцијама f_i и f_j посматра њихов родитељски чвор, као нови карактер чија је фреквенција $f_i + f_j$ и да се оптимално распоређивање тог, редукованог, скупа чворова, након чега се два најређа карактера додају испод свог родитељског чвора.

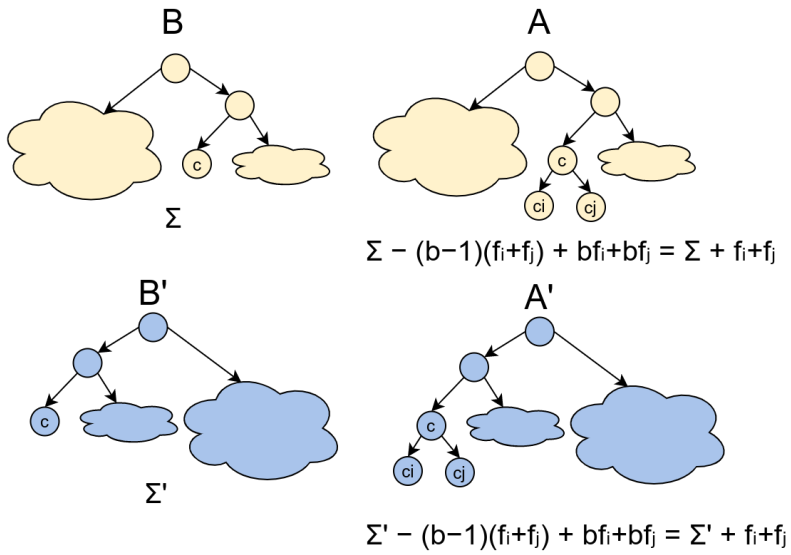
Опишимо сада Хафманову индуктивно-рекурзивну конструкцију на основу које конструишемо оптимални префиксни кôд. Индукција тече по броју карактера у азбуци за коју конструишемо кôд.

- Базу индукције чини случај када азбука има један или два карактера. Њима можемо доделити једнобитне кодове и то је сигурно оптимално.

- Редукцију на мању димензију проблема ћемо остварити тако што ћемо променити азбуку тако да два најређа карактера c_i и c_j са фреквенцијама f_i и f_j заменити новим карактером c са фреквенцијом $f_i + f_j$. Рекурзивно конструишемо дрво B (оптимални префиксни кôд) за промењену азбуку. У њему се карактер c јавља као неки лист. Дрво A (кôд) за полазну азбуку ћемо добити тако што ћемо испод листа c додати листове c_i и c_j.

Коректност поступка лежи на тврђењу да овако конструисано дрво A представља оптимални префиксни кôд за полазну азбуку.

Доказ коректности. Заиста, ако дрво A не би било оптимално за полазну азбуку, постојало би боље дрво за њу (рецимо A'). На основу претходно доказаног можемо претпоставити да се карактери c_i и c_j у њему јављају као два суседна листа (сина истог оца). Уклонимо та два листа и њиховом оцу доделимо карактер c. Тако добијамо дрво B' које одређује префиксни кôд за кодирање редуковане азбуке. Претпоставимо да кôд одређен дрветом B за кодирање текста укупно захтева Σ битова, а да кôд одређен дрветом B' за кодирање текста укупно захтева Σ' битова. Укупан број битова у дрвету A и B се разликује само за битове којима се кодирају карактери c_i, c_j и нови карактер c. У дрвету A то је $f_i b + f_j b$ битова, где је b број битова потребан за кодирање карактера c_i и c_j, док је у дрвету B то $(f_i + f_j)(b-1)$, јер је карактеру c придружена фреквенција $f_i + f_j$, док је висина чвора c за 1 мања од висине чворова c_i и c_j. Дакле, дрво A одређује кôд који за кодирање текста захтева $\Sigma - (f_i + f_j)(b-1) + (f_i b + f_j b) = \Sigma + f_i + f_j$ битова. На сличан начин се може закључити да дрво A' одређује који за кодирање текста захтева $\Sigma' + f_i + f_j$. Дакле, ако A није оптимално дрво, тада је $\Sigma' + f_i + f_j < \Sigma + f_i + f_j$, па је зато $\Sigma' < \Sigma$, што значи да ни B није оптимално дрво (јер је B' захтева мање битова за кодирање целог текста над трансформисаном азбуком). Ово је контрадикција у односу на претпоставку да смо рекурзивним позивом добили оптимално дрво B за трансформисану азбуку.



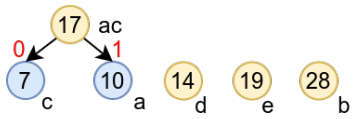
Слика 9.22: Доказ оптималности индуктивно-рекурзивне конструкције Хафмановог кода

Дакле, одређујемо два карактера са најмањим фреквенцијама појављивања, мењамо их новим карактером чија је фреквенција једнака збиру њихових фреквенција, рекурзивно конструишемо оптимално дрво и на крају у том дрвету на лист који одговара новом карактеру дописујемо два нова листа који одговарају уклоњеним карактерима са најмањим фреквенцијама. Базу индукције тј. излаз из рекурзије представља случај када остану само један или два карактера (које кодирамо са по једним битом, тј. креирамо корен дрвета испод којег се налази тај карактер тј. та два карактера).

Пример. Прикажимо рад алгоритма на примеру

```
a 10
b 28
c 7
d 14
e 19
```

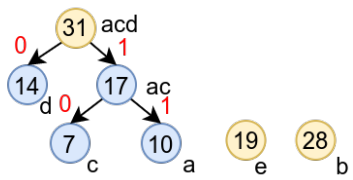
Најмањи пар фреквенција је 7 и 10 тако да крећемо од дрвета



Док азбука постаје

```
b   28
d   14
e   19
ac  17
```

Најмањи пар сада чине 14 и 17 тако да добијамо дрво

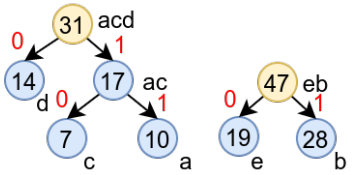


док азбука постаје

```
b 28
e 19
```

```
acd 31
```

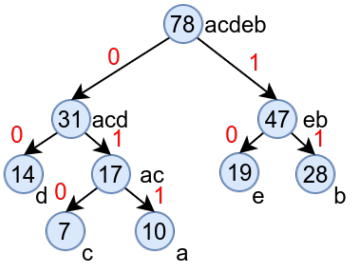
Сада је најмањи пар 19 и 28 и тако добијамо шуму



док азбука постаје

```
be  47
acd 31
```

На крају завршавамо тако што спајамо ова два карактера и добијамо дрво



Одавде читамо кодове: `a 011`, `b 11`, `c 010`, `d 00` и `e 10`.

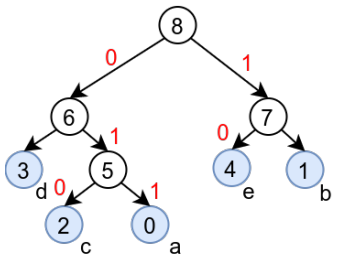
Потребно је прецизирати још неколико детаља да бисмо направили имплементацију.

У сваком кораку је у скупу карактера потребно одређивати два са најмањим фреквенцијама и мењати их са новим, чија је фреквенција једнака збиру фреквенција. Ове операције се могу веома ефикасно извршавати ако се карактери убаце у ред са приоритетом (хип) сортиран неопадајуће на основу фреквенција. У језику C++ на располагању имамо `priority_queue` који нам пружа функционалност реда са приоритетом (потребно је једино још нагласити да ред треба да буде сортиран растуће тј. да се на врху реда налази елемент са најмањом фреквенцијом (а не највећом, како је то подразумевано).

Друго питање је то како репрезентовати дрво које се гради. Унапред знамо да ће оно имати $2n-1$ чворова и знамо да ће n карактера бити листови тог дрвета, док ће $n-1$ чворова бити унутрашњи. Сваки чвор можемо нумерисати бројевима од 0 до $2n-2$. Листовима ћемо делити индексе у складу са редним бројем карактера који су им придружени c_i (карактеру c_i придружен је чвор број i). Индексе унутрашњим чворовима (бројеви од n до $2n-2$) додељиваћемо у редоследу њиховог формирања (корен ће бити формиран последњи и њему ће бити додељен индекс $2n-2$). За сваки од $n-1$ унутрашњих чворова треба да знамо индекс левог и десног детета (увек постоје оба). Те информације можемо чувати у два помоћна низа. Унутрашњи чворови имају индексе од n до $2n-2$ и за чвор број i у првом низу на позицији $i-n$ чувамо индекс левог детета, а у другом низу на позицији $i-n$ чувамо индекс десног детета.

Пример. Индекси чворова за дрво из претходног примера су приказани на слици. Низови који одређују леву и десну децу унутрашњих чворова су:

```
i      5 6 7 8
i-n    0 1 2 3
levi:  2 3 4 6
desni: 0 5 1 7
```



Слика 9.23: Индекси чворова у дрвету

У реду са приоритетом ћемо уз фреквенцију сваког чвора чувати и индекс (то ће бити позиција која одговара том чвору у низовима у којима чувамо опис деце).

Када је дрво креирано, кодове свих карактера можемо добити исцрпним обиласком целом дрвета (рекурзивном функцијом).

```
// svakom karakteru u listu drveta ciji je koren cvor sa indeksom i i kodom
// kod dodeljujemo kod i upisujemo ga u mapu kodovi
void procitajKodove(const vector<int>& levo,
                    const vector<int>& desno,
                    const vector<char>& karakteri,
                    int i, int n, const string& kod,
                    map<char, string>& kodovi) {
  if (i < n)
    // stigli smo do lista list
    kodovi[karakteri[i]] = kod;
  else {
    // u pitanju je unutrasnji cvor, pa obradjujemo levo i desno poddrvo
    procitajKodove(levo, desno, karakteri, levo[i-n], n, kod + "0", kodovi);
    procitajKodove(levo, desno, karakteri, desno[i-n], n, kod + "1", kodovi);
  }
}

// za dati niz karaktera i frekvencije njihovog pojavljivanja u tekstu
// odredjuje optimalni prefiksni kod
map<char, string> hafman(const vector<char>& karakteri, const vector<int>& frekvencije) {
  // broj karaktera
  int n = karakteri.size();

  // reprezentacija drveta - svakom unutrasnjem cvoru dodeljujemo levi i desni indeks
  vector<int> levo(n-1), desno(n-1);

  // red sa prioritetom koji sadrzi cvorove odredjene njihovim frekvencijama i indeksima
  priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
  // ubacujemo sve listove drveta (karaktere) u red - dodeljujemo im indekse od 0 do n-1
  for (int i = 0; i < n; i++)
    pq.emplace(frekvencije[i], i);

  // gradimo unutrasnje cvorove
  for (int i = n; i < 2*n - 1; i++) {
    // povezujemo dva cvora sa najnizim frekvencijama
    auto f1 = pq.top(); pq.pop();
    auto f2 = pq.top(); pq.pop();
    // ubacujemo cvor i u drvo
    levo[i - n] = f1.second; desno[i - n] = f2.second;
    // redukujemo azbuku i dodajemo joj kombinovani karakter koji odgovara cvoru i
    pq.emplace(f1.first + f2.first, i);
  }
```

```
  // iz drveta ocitavaom kodove svih karaktera
  map<char, string> kodovi;
  procitajKodove(levo, desno, karakteri, 2*n-2, n, "", kodovi);
  return kodovi;
}

// za dati niz karaktera i frekvencije njihovog pojavljivanja u tekstu,
// i dati prefiksni kod odredjuje broj bita potrebnih za kodiranje teksta
int brojBitova(const vector<char>& karakteri, const vector<int>& frekvencije,
               const map<char, string>& kodovi) {
  // broj karaktera
  int n = karakteri.size();

  // izracunavamo ukupan broj bitova potrebnih za kodiranje teksta
  int broj = 0;
  for (int i = 0; i < n; i++)
    broj += frekvencije[i] * kodovi.at(karakteri[i]).size();

  return broj;
}
```