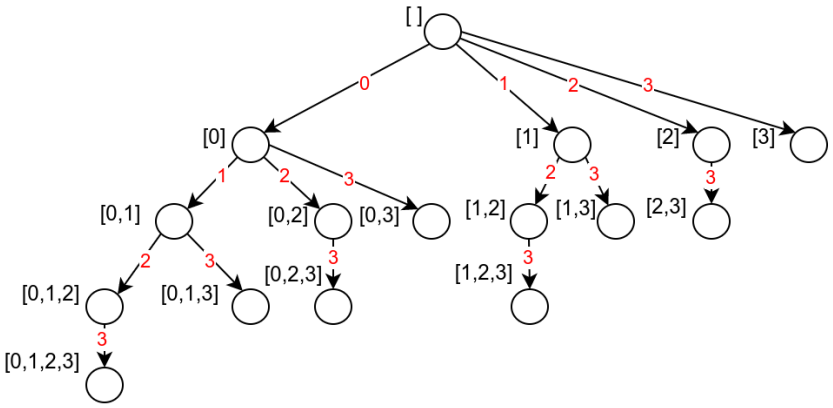
Глава 6

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 $\{a_0, a_2, a_3\}$.

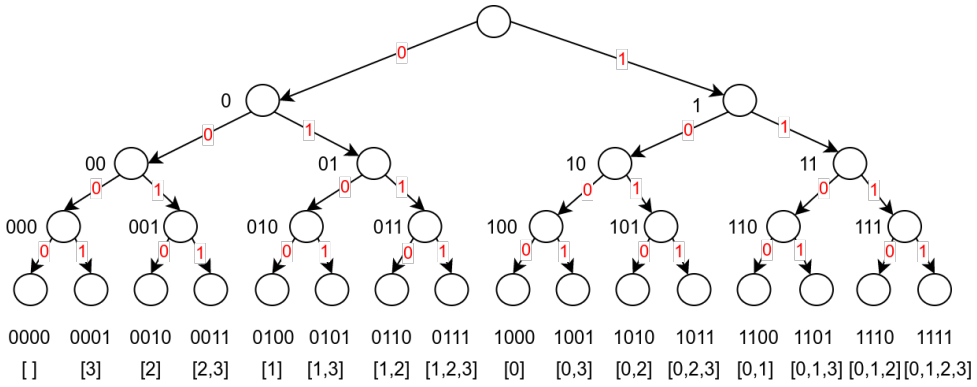
Сви објекти се обично могу представити дрветом и то дрво одговара процесу њиховог генерисања тј. обиласка (оно се не прави експлицитно, у меморији, али нам помаже да разумемо и организујемо поступак претраге). Обилазак дрвета се најједноставније изводи у дубину (често рекурзивно). За прву наведену репрезентацију 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).

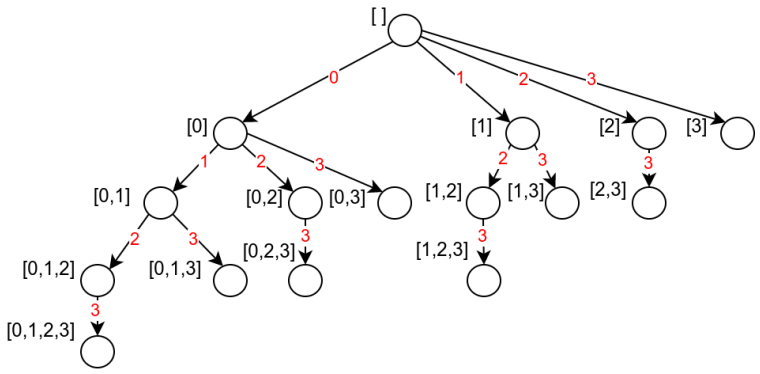
```
// ispis svih podskupova skupa {0, ..., n-1} u leksikografskom redosledu
void obradiSvePodskupove(int n) {
  // krecemo od praznog podskupa
  vector<int> podskup(n);
  // broj elemenata podskupa
  int k = 0;
  // obradjujemo (ispisujemo) tekuci podskup i prelazimo na sledeci,
  // dok god je to moguce
  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 $\{a_{i+1}, \ldots, a_{n-1}\}$.

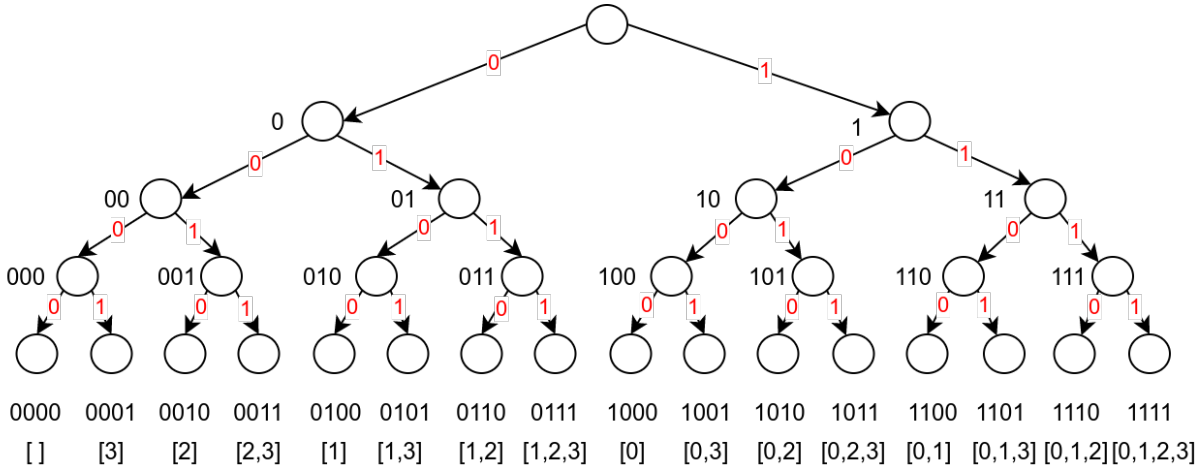
Иницијално је $i = 0$, а подскуп је празан (празан скуп је, заиста, једини подскуп празног супа $\{a_0, \ldots, a_{i-1}\}$ и он се на све могуће начине проширује елементима скупа $\{a_i, \ldots, a_{n-1}\} = \{a_0, \ldots, a_{n-1}\}$).

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

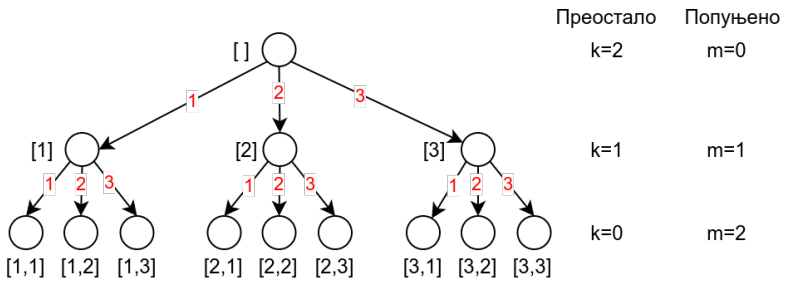
Варијације се могу набројати индуктивно рекурзивном конструкцијом.

- 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.

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

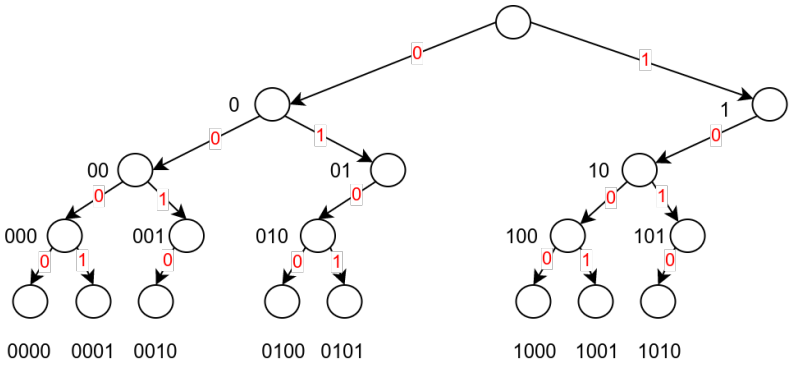
Задатак представља модификацију задатка Све варијације.

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

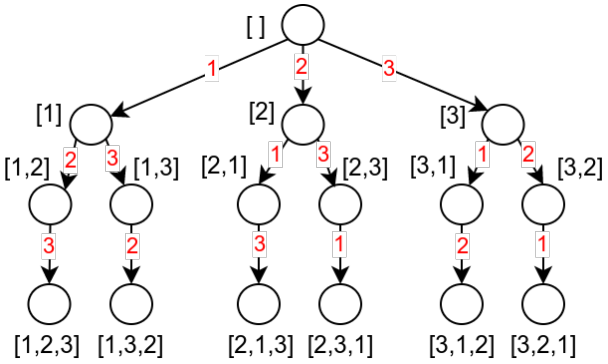
Улаз	*Излаз*
3	1 2
2	1 3
	2 1
	2 3
	3 1
	3 2

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).

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.

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

Улаз	*Излаз*
5	1 3 5
1 3 4	

Пример 2

Улаз	*Излаз*
5	1 4 5
1 3 5	

Пример 3

Улаз	*Излаз*
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.

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.

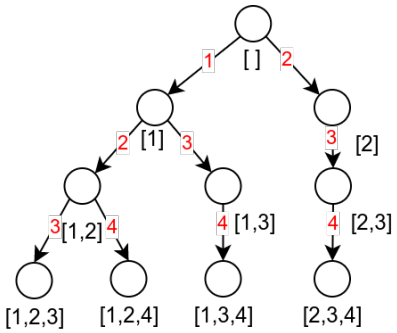
Пример

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).



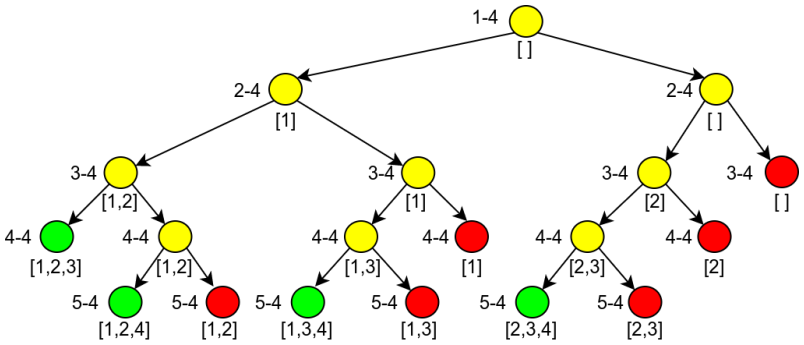
Слика 6.8: Рекурзивно генерисање комбинација дужине 3 из скупа $\{1, 2, 3, 4\}$

```
// 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

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.

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.

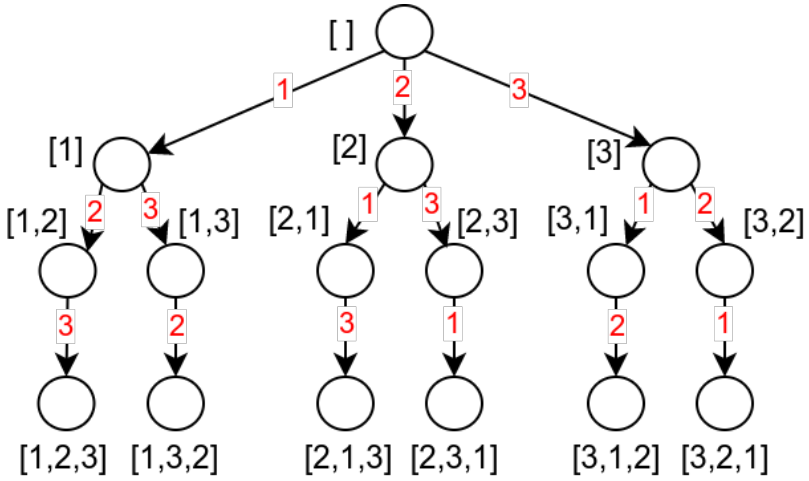
Пример

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

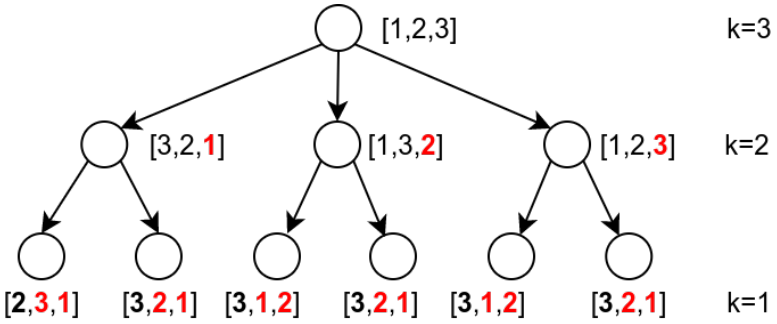
Ако се одрекнемо услова да пермутације буду уређене лексикографски, није неопходно вршити проверу да ли је текући елемент већ употребљен тј. можемо поступити на следећи начин.

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.

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

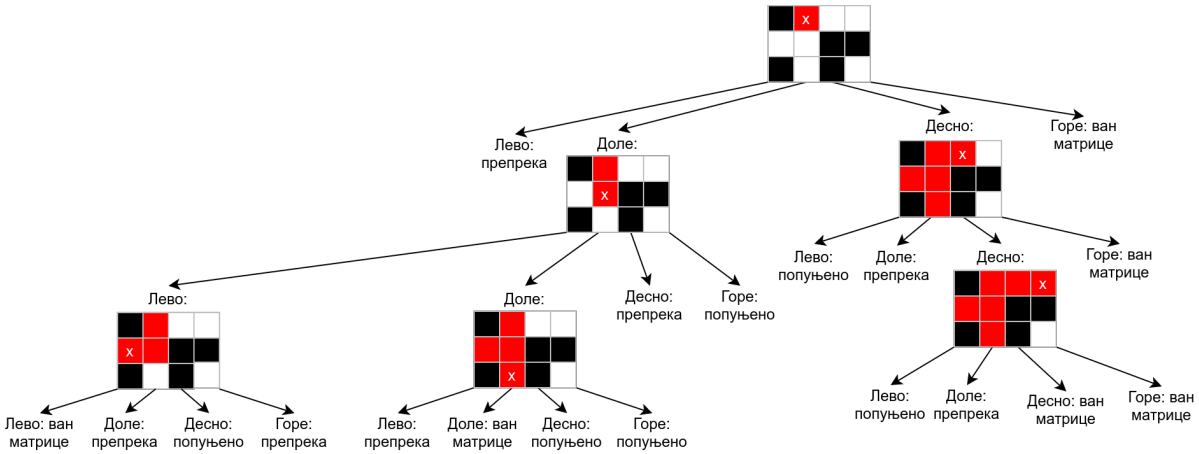
Основу решења чини рекурзивна функција која детектује и обележава белу област почевши од једног њеног поља. Функција се позива за поље у матрици које је тренутно обојено бело (у матрици на том пољу пише 0), обележава га (уписује неку другу вредност, на пример, -1), и затим се рекурзивно позива за све 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.

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.

```
// 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[N][M], 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);
  }
```

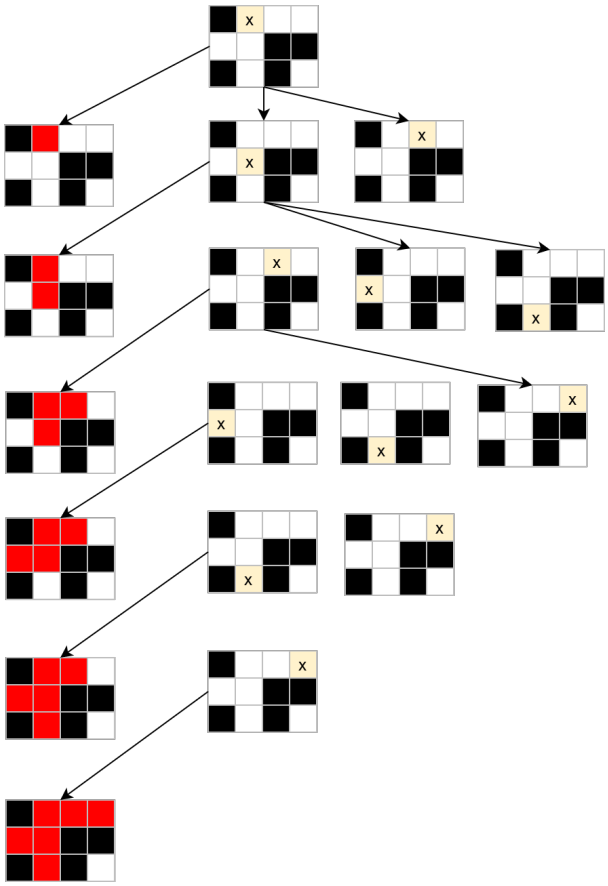
```
}

int main() {
  // ucitavamo matricu
  // ...
  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++;
      }

  cout << oblast << endl;
}
```

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[N][M], 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.

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

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	*Излаз*

`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..`
	`da`

Пример 2

Улаз	*Излаз*

`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..`
	`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.push(make_pair(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.push(make_pair(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.push(make_pair(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.push(make_pair(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

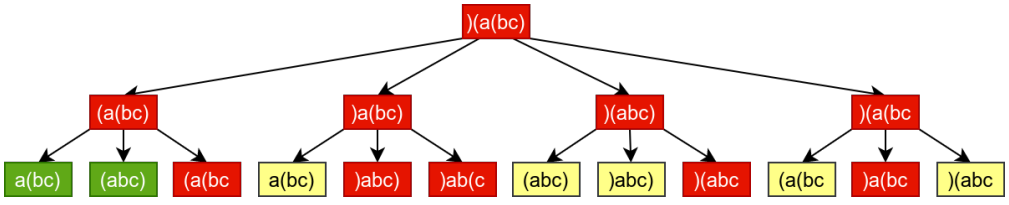
Задатак можемо решити претрагом у ширину. Претпоставићемо да су полазна ниска и ниске које се од ње добијају брисањем заграда (друге карактере нема смисла брисати, јер не утичу на упареност заграда) стања кроз која се пролази. Полазно стање је оно које одговара полазној ниски, а завршна стања су сва 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*

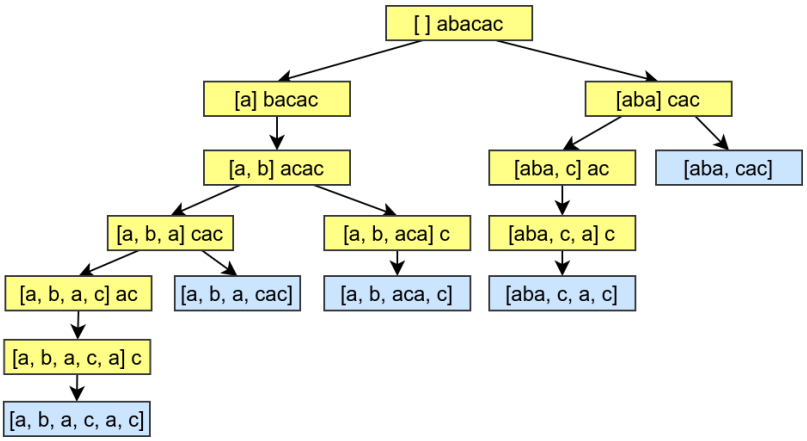
`sirimamiris`	`s i r i m a m i r i s`
	`s i r i m a m iri s`
	`s i r i mam i r i s`
	`s i r i mam iri s`
	`s i r imami r i s`
	`s i rimamir i s`
	`s iri m a m i r i s`
	`s iri m a m iri s`
	`s iri mam i r i s`
	`s iri mam iri s`
	`s irimamiri s`
	`sirimamiris`

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

Улаз	*Излаз*
`12`	`nema`

Пример 2

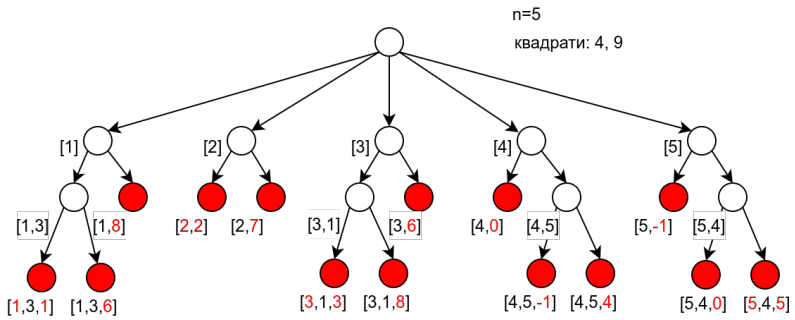
Улаз	*Излаз*
`17`	`16 9 7 2 14 11 5 4 12 13 3 6 10 15 1 8 17`

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.

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

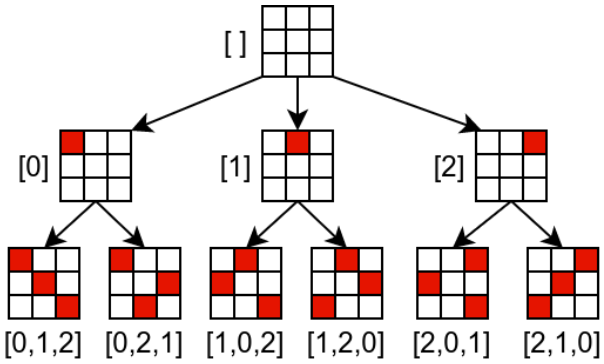
Груба сила - провера свих пермутација

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.

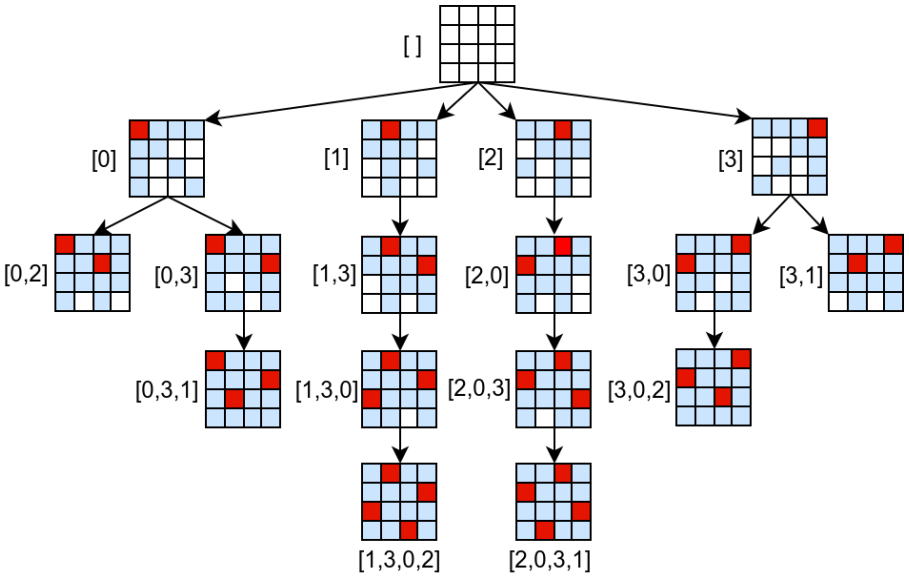


Слика 7.6: Распоређивање 3 даме - све пермутације и позиције које су њима представљене

```
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);
}
```

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.

```
// 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);
}
```

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 (слично као у задатку Латински квадрати). 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 (слично као у задатку Шаховско поље). 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

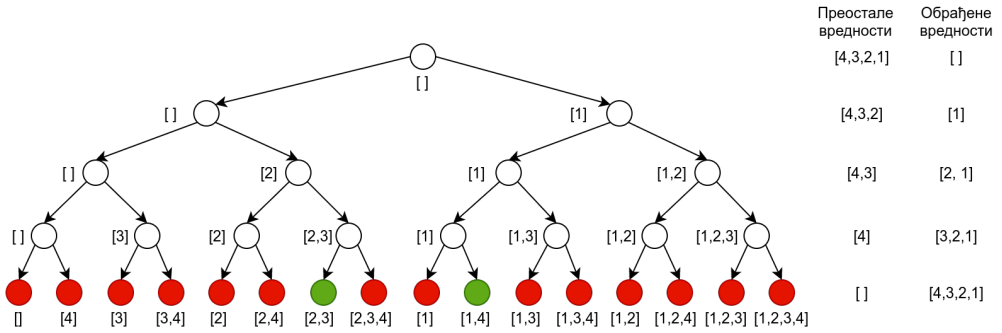
Груба сила

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 $k = n$.

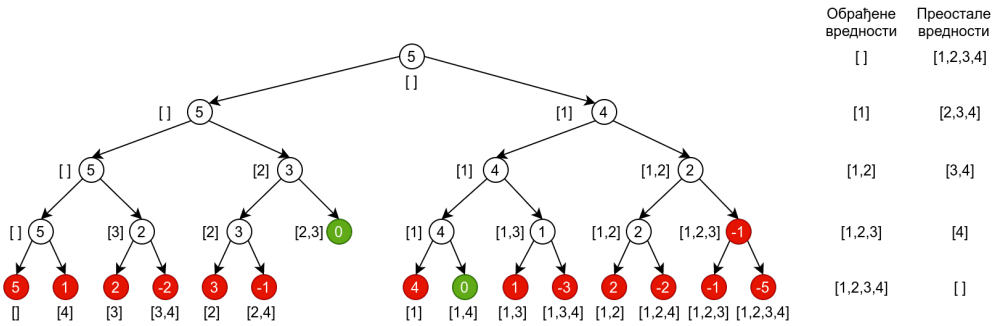
Ако је циљни збир једнак нули, то значи да је збир тренутног подниза елемената на позицијама $[0, k)$ једнак полазном траженом збиру и да смо нашли један задовољавајући подниз. Додатно, пошто су сви елементи низа позитивни, подниз се не може никако проширити додатним елементима тако да збир и даље остане 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.



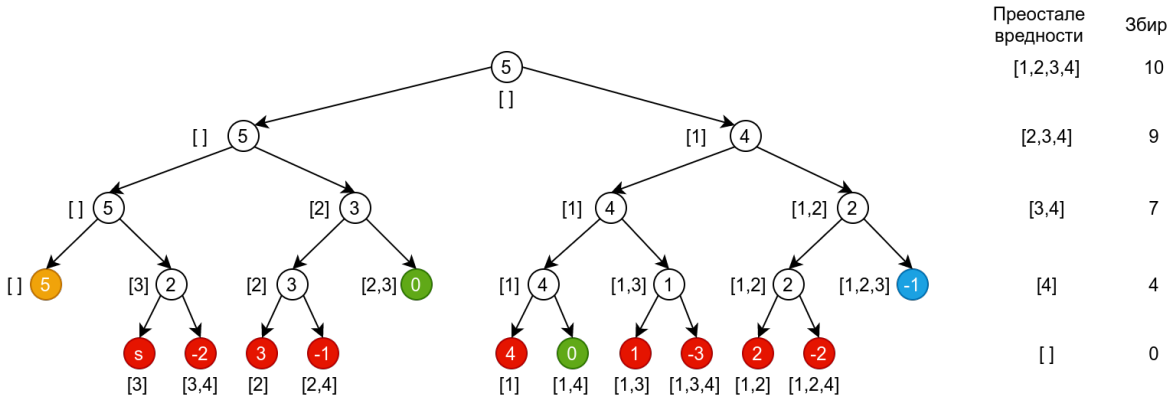
Слика 7.9: Груба сила – преостали циљни збир

```
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).

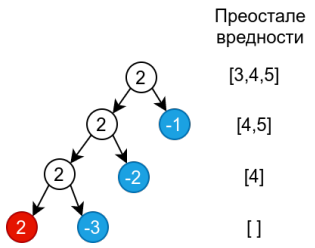


Слика 7.10: Претрага уз основно одсецање

```
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 n) {
  // 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]
```

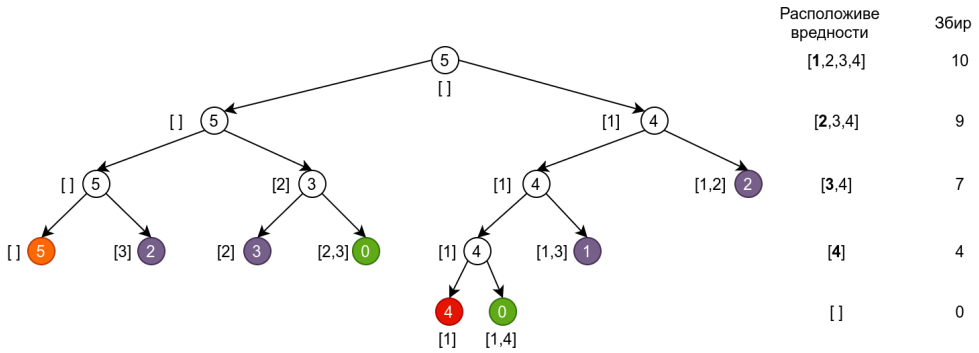
```
  return brojPodnizovaDatogZbira(niz, ciljniZbir - niz[k],
                                 zbirPreostalih - niz[k], k+1) +
    // broj podnizova u kojima ne ucestvuje element a[k]
    brojPodnizovaDatogZbira(niz, ciljniZbir,
                            zbirPreostalih - niz[k], k+1);
}

// 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).



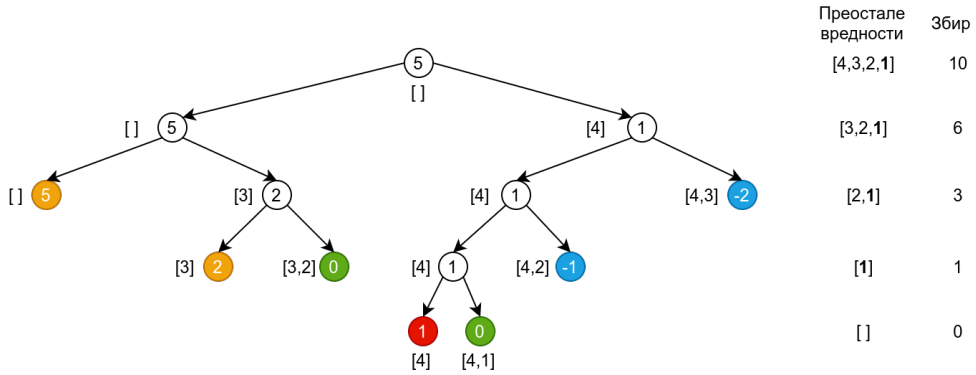
Слика 7.12: Претрага са одсецањем

```
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	0.05
2.3	
1.0	
0.5	
2.0	
0.25	
4.0	

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.

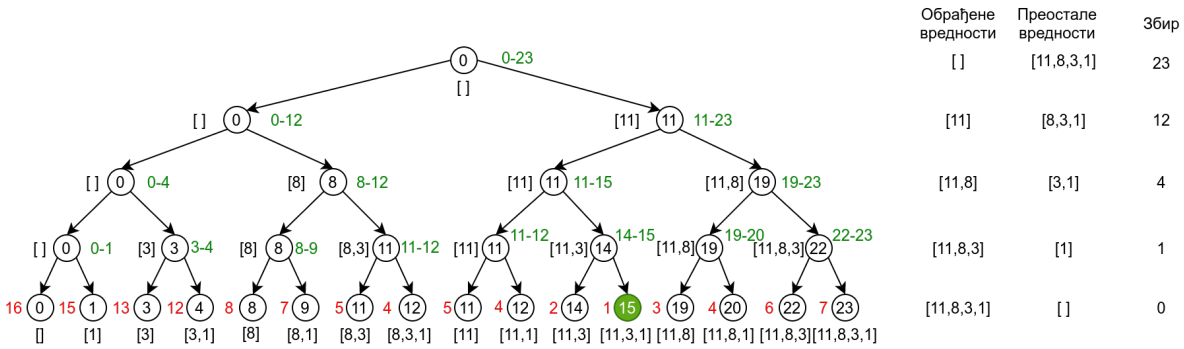
```
double merenje(const vector<double>& tegovi, double ciljnaMasa,
               int k, double tekucaMasa) {
  // nemamo vise tegova koje mozemo uzimati
  if (k == tegovi.size())
    // najmanja razlika je odredjena tekucom masom ukljucenih tegova
    return abs(ciljnaMasa - tekucaMasa);

  // 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.



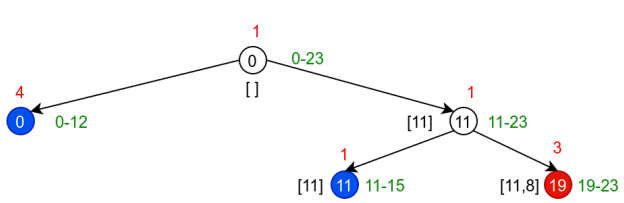
Слика 7.14: Исцрпна претрага

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: 3 бојење

У једној земљи постоји неколико планинских врхова на којима ће се поставити предајници најсавременије мобилне мреже. Они могу да раде на три различите радио-фреквенције. Сваки предајник може да преноси специјални сигнал другим предајницима који су близу њега, при чему два 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

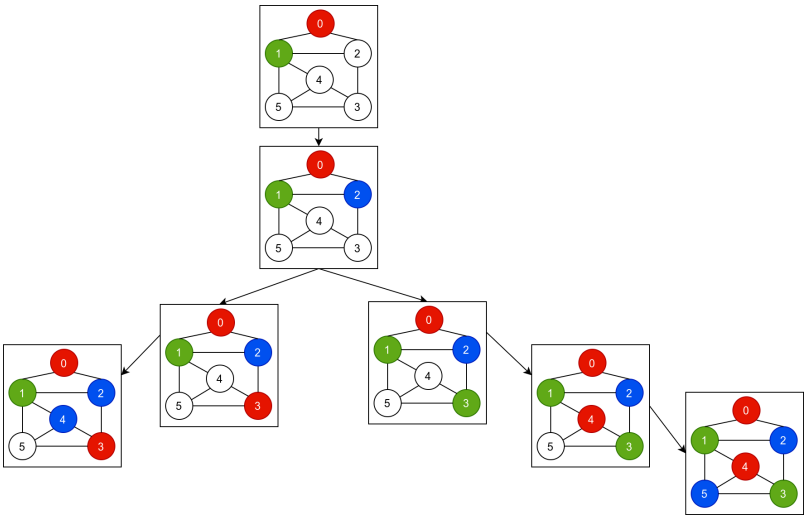
Решење

Овај проблем је у литератури познат као проблем 3-бојења графова (сваки предајник представља чвор графа, а свака фреквенција једну од 3 допуштене боје).

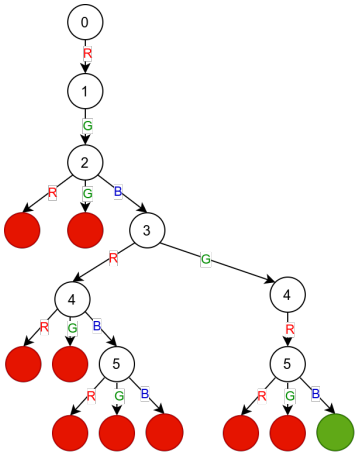
Задатак се једноставно (али не и ефикасно) решава бектрекинг претрагом.

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 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.

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 сае 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 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 3 бојење 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);
}
```

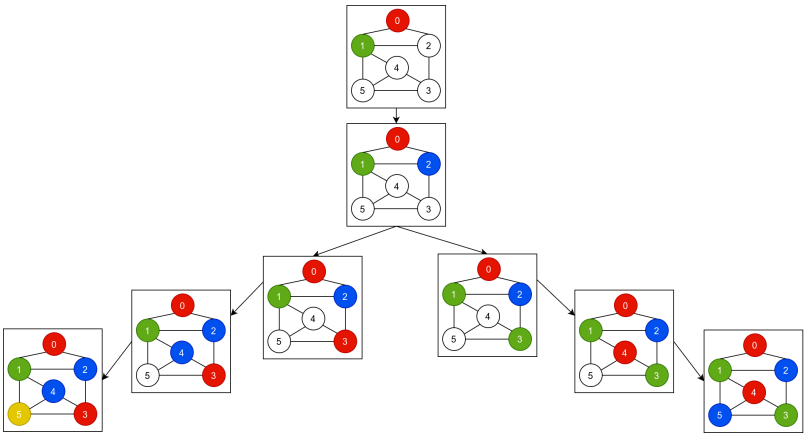
```
// najmanji broj boja kojima se mogu obojiti cvorovi grafa, tako
// da nikoja dva cvora nisu susedna
int minBrojBoja(const vector<vector<int>>& susedi) {
  // broj cvorova grafa
  int n = susedi.size();
  // binarnom pretragom nalazimo minimalan broj boja
  // sa strogo manje od l graf se ne moze 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 ограничење 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.

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).

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 расположиве боје постају само 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;
}
```