Друштво математичара Србије

Филип Марић

Одабрани задаци из програма такмичења ученика средњих школа

Материјал за зимски републички семинар, 2020.

Садржај

Глава 1

Теорија бројева

Задатак: Исецање квадрата

Поставка: Играмо игру исецања што већих квадрата из листа папира правоугаоног облика датих димензија. Најпре се исече највећи могућ квадрат тако што се исече лист папира по дужој страници (на пример за лист димензија 3×7, највећи могућ квадрат је димензија 3×3). Квадрат се уклања и поступак се понавља све док преостали правоугаоник не постане квадрат. Написати програм којим се одређује колико се квадрата добија.

Улаз: Прва линија стандардног улаза садржи природан број a који представља једну димензију папира, а друга линија садржи природан број b који представља другу димензију папира.

Излаз: У првој линији стандардног излаза приказати број добијених квадрата.

Пример

Улаз	*Излаз*
3 7	5

Решење:

Рекурзивна формулација

Ако су дужина a и ширина b листа једнаке, онда је он већ квадратног облика и од њега се добија тачно само један квадрат. Ако је $a > b$ онда од папира исецамо квадрат димензије $b \times b$, па је укупан број квадрата за један већи од броја квадрата које можемо исећи од преосталог дела папира димензије $(a - b) \times b$. На крају, ако је $a < b$ тада се исеца квадрат димензије $a \times a$, па је укупан број квадрата за један већи од броја квадрата које можемо исећи од преосталог дела папира димензије $a \times (b - a)$. На основу овога могуће је дефинисати рекурзивну функцију за израчунавање броја квадрата.

Ослобађање од рекурзије

Те рекурзије је једноставно ослободити се тако што се употреби петља. Током петље

одржавамо бројач који броји колико квадрата смо исекли. Од листа папира исецамо квадрат по квадрат док лист не постане квадратног облика. Зато се петља извршава док је a различито од b. Квадрат исецамо тако што лист пресечемо паралелно краћој страници, и при сваком исецању дужу страницу листа смањимо за дужину краће странице. Дакле, ако је $a < b$ тада b мењамо са $b - a$, а ако је $a > b$ тада a мењамо са $a - b$. У оба случаја бројач квадрата увећавамо за 1. Бројач квадрата на почетку можемо поставити на нулу, а након петље увећати за 1 (тиме бројимо и квадрат који је остао после последњег исецања). Наравно, уместо увећања бројача након петље, бројач квадрата можемо и на почетку иницијализовати на 1.

Оптимизација заменом одузимања целобројним дељењем

Иако претходно решење даје исправан резултат, за неке вредности улазних параметара до решења се долази неефикасно тј. у непотребно великом броју корака. Основни проблем су случајеви у којима је велика разлика између дужина страница папира. На пример, ако је већа страница дужине 1001, а мања дужине 2, биће потребно 500 корака и 500 одузимања док се не стигне до квадрата димензије 1 пута 2 (бројеви 1001 и 2 биће мењани бројевима 999 и 2, затим 997 и 2 итд). Могуће је унапред закључити да ће се моћи исећи 500 квадрата странице 2, а да ће након тога остати папир димензија 1 пута 2. Наиме, број 500 се може израчунати као целобројни количник бројева 1001 и 2, док број 1 представља остатак при дељењу та два броја. Ако је $b = 0$, тада је број квадрата једнак нули. У супротном је број квадрата једнак збиру количника $a \operatorname{div} b$ и броја квадрата који се могу исећи од папира димензије $b \times a \bmod b$, што доводи до рекрузивне формулације проблема. Ако је $a < b$ у првом кораку се број квадрата се неће увећати већ ће се само разменити вредности a и b (јер је $a \operatorname{div} b = 0$, док је $a \bmod b = a$).

Уместо рекурзивног, можемо поново имплементирати и итеративно решење у којем се у петљи која се извршава све док је $b > 0$ бројач увећава за $a \operatorname{div} b$, док се променљиве a и b мењају редом вредностима b и $a \bmod b$ (за то је потребна помоћна променљива). Приметимо да је поступак описан у овом задатку веома блиско везан за поступак одређивања највећег заједничког делиоца два броја Еуклидовим алгоритмом који је описан у задатку Еуклид.

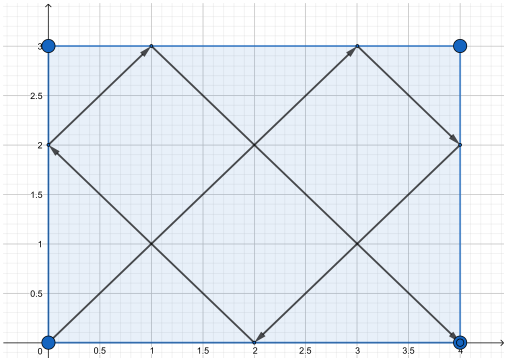
```
#include <iostream>
#include <algorithm>

using namespace std;

int main() {
  int a, b;
  cin >> a >> b;
  int bk = 0;
  while (b != 0) {
    bk += a / b;
    int ost = a % b;
    a = b;
    b = ost;
  }
```

```
  cout << bk << endl;
}
```

```
#include <iostream>

using namespace std;

int main() {
  int a, b;
  cin >> a >> b;
  int bk = 1;
  while (a != b) {
    if (a > b)
      a -= b;
    else
      b -= a;
    bk++;
  }
  cout << bk << endl;
  return 0;
}
```

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

using namespace std;

int broj_kvadrata(int a, int b) {
  if (b == 0)
    return 0;
  return a / b + broj_kvadrata(b, a % b);
}

int main() {
  int a, b;
  cin >> a >> b;
  cout << broj_kvadrata(a, b) << endl;
}
```

```
#include <iostream>

using namespace std;

int broj_kvadrata(int a, int b) {
  if (a == b)
    return 1;
  else if (a > b)
    return 1 + broj_kvadrata(a - b, b);
  else
```



Слика 1.1: Билијар

```
    return 1 + broj_kvadrata(a, b - a);
}

int main() {
  int a, b;
  cin >> a >> b;
  cout << broj_kvadrata(a, b) << endl;
}
```

Задатак: Билијар из ћошка

Поставка: Билијарски сто је правоугаоног облика димензије $m \times n$ и има четири рупе у ћошковима. Посматрајмо цртеж стола, такав да му је ширина m, а висина n. Лоптица се удара из доњег левог угла (поља са координатама $(0, 0)$) дуж линије која је под углом од 45 степени у односу на ивице стола. Ако претпоставимо да лоптица не успорава своје кретање, да се од сваке се ивице одбија под углом од $45°$, да је веома мала и да у рупу упада само ако су јој координате центра једнаке координати рупе, напиши програм који одређује у коју рупу ће после неког времена упасти, као и колико ће се пута пре тога одбити о ивице стола.

Улаз: Са стандардног улаза се уносе два цела броја m и n ($1 \leq m, n \leq 10^9$) који представљају димензије стола.

Излаз: На стандрадни излаз у првом реду исписати координате рупе у коју ће лоптица упасти, а у другом број одбијања о ивице стола пре него што се то деси.

Пример

Улаз	*Излаз*
`4 3`	`4 0`

Објашњење

Кретање лоптице је приказано на слици.

Решење: Да би се лоптица нашла у некој рупи она крећући се хоризонтално треба да пређе растојање (дужину пређеног пута) које представља неки умножак ширине

стола и крећући се вертикално треба да пређе растојање које представља неки умножак висине стола. Пошто је интензитет њене брзине и у хоризонталном и у вертикалном правцу исти (јер се стално креће под углом од 45 степени у односу на неку ивицу стола), растојање пређено у хоризонталном и у вертикалном смеру је увек исто. Дакле, лоптица упада у рупу у тренутку када је пређено растојање први пут неки умножак бројева m и n. Најмањи број који је дељив и са m и са n је њихов најмањи заједнички садржалац $nzs(m, n)$.

Број пута n_x који је лоптица хоризонтално пребрисала целу ширину стола једнак је количнику $nzs(m, n)$ и броја m, а број n_y пута који је лоптица вертикално пребрисала целу висину стола једнак је количнику $nzs(m, n)$ и броја n. Ако је број n_x непаран, лоптица се налази у некој рупи на десној, а ако је паран, налази се у некој рупи на левој ивици стола, па x координату рупе лако одређујемо анализом парности броја n_x. Аналогно, анализом парности броја n_y одређујемо координату y рупе.

Ако је лоптица хоризонталну ширину прешла n_x пута, она се о неку леву и десну ивицу одбила тачно $n_x - 1$ пута. Слично, ако је вертикалну ширину прешла n_y пута, о горњу и доњу ивицу се одбила тачно $n_y - 1$ пута. Укупан број одбијања је, дакле, $(n_x - 1) + (n_y - 1)$.

НЗС два броја можемо израчунати Еуклидовим алгоритмом, али морамо водити рачуна да због великих улазних параметара може наступити прекорачење и морамо користити адекватне типове података.

```
#include <iostream>

using namespace std;

int nzd(int a, int b) {
  while (b > 0) {
    int ost = a % b;
    a = b;
    b = ost;
  }
  return a;
}

long long nzs(int a, int b) {
  return ((long long)a / nzd(a, b)) * b;
}

int main() {
  int m, n;
  cin >> m >> n;
  long long S = nzs(m, n);
  int nx = S / m, ny = S / n;
  int x = nx % 2 == 0 ? 0 : m;
  int y = ny % 2 == 0 ? 0 : n;
  cout << x << " " << y << endl;
```

```
  cout << (nx - 1) + (ny - 1) << endl;
  return 0;
}
```

Рачунање НЗС можемо избећи. За решење нама је потребно да знамо вредности $n_x = nzs(m,n)/m$ и $n_y = nzs(m,n)/n$. Пошто је $nzs(m,n) = (mn)/nzd(m,m)$, важи да је $n_x = n/nzd(m,n)$ док је $n_y = m/nzd(m,n)$. Тиме избегавамо потребу за радом са великим бројевима и избегавамо могућност настанка прекорачења.

```
#include <iostream>

using namespace std;

int nzd(int a, int b) {
  while (b > 0) {
    int ost = a % b;
    a = b;
    b = ost;
  }
  return a;
}

int main() {
  int m, n;
  cin >> m >> n;
  int N = nzd(m, n);
  int nx = n / N, ny = m / N;
  int x = nx % 2 == 0 ? 0 : m;
  int y = ny % 2 == 0 ? 0 : n;
  cout << x << " " << y << endl;
  cout << (nx - 1) + (ny - 1) << endl;
  return 0;
}
```

Задатак: Ератостеново сито

Поставка: Напиши програм који одређује број простих бројева у интервалу $[a, b]$ и њихов збир (ако збир има више од 6 цифара, исписати само последњих 6).

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

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

Пример

Улаз	*Излаз*
1 1000	168 76127

Решење: Први начин за исписивање свих простих бројева мањих од датог броја је-

сте имплементација функције која за конкретан број проверава да ли је прост (њу смо описали у задатку Прост број) и да се затим, у петљи, за сваки број мањи од датог провери да ли је прост коришћењем те функције. Међутим, такво решење је неефикасно и много бољи резултат може се добити применом алгоритма познатог као Ератостеново сито.

Основна идеја алгоритма је да се прво напишу сви бројеви од 1 до датог броја n, затим да се прецрта број 1 (јер он по дефиницији није прост), након њега сви умношци броја 2 (нису прости зато што су дељиви са 2, док број 2 остаје непрецртан јер је он прост), затим умношци броја 3 (нису прости јер су дељиви бројем 3), затим умношци броја 5 (нису прости зато што су дељиви бројем 5) и тако даље. Умношке сложених бројева нема потребе посебно прецртавати јер су они већ прецртани током прецртавања умножака неког од њихових простих фактора (на пример, нема потребе посебно прецртавати умношке броја 4 јер су они већ прецртани током прецртавања умножака броја 2). Такође, приликом прецртавања умножака броја i довољно је кренути од $i \cdot i$ јер су мањи умношци већ прецртани раније (имају праве факторе мање од i). Потребно је да се поступак понавља све док се не прецртају умношци свих простих бројева који нису већи од корена броја n. Бројеви који су остали непрецртани су прости (јер знамо да немају правих делилаца мањих или једнаких корену од n, па самим тим и мањих или једнаких свом корену, а пошто немају делилаца испод вредности корена, на основу теореме коју смо доказали у задатку Прост број, немају правих делилаца ни изнад вредности корена). Прецртавање бројева моделоваћемо низом (или вектором) који садржи логичке вредности (вредности типа `bool`) и прецртане бројеве означаваћемо са `false`, а непрецртане са `true`.

Одређивање простих бројева (помоћу поменутог низа тј. вектора) реализоваћемо у засебној функцији, јер та функција може бити корисна и у многим наредним задацима.

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

На основу спецификације задатка потребно је одредити највише 6 последњих цифара збира свих простих бројева из интервала $[a, b]$, што, на основу задатка Операције по модулу, знамо да је еквивалентно одређивању збира тих бројева по модулу 10^6. У циклусу пролазимо кроз све бројеве од a до b, вршимо филтрирање на основу услова да је број прост и вршимо бројање и сабирање добијене филтриране серије (слично као, на пример, у задатку Просек одличних). Напоменимо да се збир рачуна тако што се на почетку иницијализује на нулу, а затим се у сваком кораку израчунава сабирање збира и текућег простог броја по модулу 10^6 (`zbir = (zbir % 1000000 + p % 100000) % 100000`). Пошто ће у сваком кораку збир бити мањи од 10^6, и пошто не постоји опасност од прекорачења када се у обзир узме максимална вредност простих бројева који се сабирају, претходни корак се може заменити кораком `zbir = (zbir + p) % 1000000`.

```
#include <iostream>
#include <vector>

using namespace std;
```

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

int main() {
  // ucitavamo granice intervala
  int a, b;
  cin >> a >> b;

  // odredjujemo proste brojeve u intervalu [0, b]
  vector<bool> prost;
  Eratosten(prost, b);

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

  // prijavljujemo rezultat
  cout << broj << " " << zbir << endl;
  return 0;
}
```

```
#include <iostream>
#include <vector>

using namespace std;

// funkcija koja proverava da li je dati broj prost
bool prost(int n) {
  if (n == 1) return false;     // broj 1 nije prost
  if (n == 2) return true;      // broj 2 jeste prost
```

```
  if (n % 2 == 0) return false; // ostali parni brojevi nisu prosti
  // proveravamo neparne delioce od 3 do korena iz n
  for (int i = 3; i*i <= n; i += 2)
    if (n % i == 0)
      return false;
  // nismo nasli delioca - broj jeste prost
  return true;
}

int main() {
  // ucitavamo granice intervala
  int a, b;
  cin >> a >> b;

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

  // prijavljujemo rezultat
  cout << broj << " " << zbir << endl;
  return 0;
}
```

Задатак: Прост број

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

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

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

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

Решење: Природан број је прост ако је већи од 1 и ако није дељив ни са једним бројем осим са један и са самим собом. По дефиницији број 1 није прост. Дакле, број већи од 1 је прост ако нема ни једног правог делиоца. Потребно је дакле извршити претрагу скупа потенцијалних делилаца и проверити да ли неки од њих стварно дели број n. Скуп потенцијалних делилаца је скуп свих природних бројева од 2 до $n-1$. Међутим, како смо већ описали у задатку Савршени бројеви, делиоци броја се увек јављају у пару. На пример, делиоци броја 100 организовани по паровима су (1, 100), (2, 50), (4, 25) (5, 20) и (10, 10). Ако је i делилац броја n, делилац је и број $\frac{n}{i}$. При том, ако је $i \geq \sqrt{n}$, тада је $\frac{n}{i} \leq \sqrt{n}$. Дакле, важи теорема која каже да број има праве делиоце који су

већи или једнаки вредности $\sqrt{n}$ ако и само ако има делиоце који су мањи или једнак вредности $\sqrt{n}$. То нам даје могућност да претрагу потенцијалних делилаца редукујемо само на интервал $[2, \sqrt{n}]$, јер ако број нема делилаца мањих или једнаких вредности $\sqrt{n}$, онда не може да има делилаца већих или једнаких тој вредности, тј. нема правих делилаца и прост је. Обратите пажњу да је ово скраћивање интервала веома значајно (ако је највећи број око 10^9 тј. око милијарду, уместо милијарду делилаца потребно је проверавати само њих корен из милијарду, што је тек нешто изнад тридесет хиљада).

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

Још једна могућа оптимизација је да се на почетку провери да ли је број паран а да се након тога проверавају само непарни делиоци, међутим, та оптимизација не доноси превише (обилазак до корена је смањио број потенцијалих кандидата са милијарде на тек тридесетак хиљада, а провера само парних делилаца тај број смањује на петнаестак хиљада, што није значајна уштеда јер је већ и провера 30000 вредности на данашњим рачунарима веома брза).

Ако је потребно за више бројева одједном проверити да ли су прости, уместо проверавања сваког појединачног, боље је употребити Ератостеново сито (види задатак Ератостеново сито).

```
#include <iostream>

using namespace std;

bool prost(int n) {
  if (n == 1) return false;
  for (int i = 2; i*i <= n; i++)
    if (n % i == 0)
      return false;
  return true;
}

int main() {
  int n;
  cin >> n;
  cout << (prost(n) ? "DA" : "NE") << endl;
```

```
  return 0;
}
```

```
#include <iostream>

using namespace std;

bool prost(int n) {
  int i = 2;
  while (i*i <= n && n % i != 0)
    i++;
  return n > 1 && i*i > n;
}

int main() {
  int n;
  cin >> n;
  cout << (prost(n) ? "DA" : "NE") << endl;
  return 0;
}
```

```
#include <iostream>

using namespace std;

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

int main() {
  int n;
  cin >> n;
  cout << (prost(n) ? "DA" : "NE") << endl;
  return 0;
}
```

Глава 2

Рекурзија и backtrack

Задатак: Следећа варијација

Поставка: 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.

```cpp
#include <iostream>
#include <vector>

using namespace std;

void ispisi(const vector<int>& varijacija) {
    for (int x : varijacija)
      cout << x << " ";
    cout << endl;
}

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;
}

int main() {
  int k, n;
  cin >> k >> n;
  vector<int> varijacija(k);
  for (int i = 0; i < k; i++)
    cin >> varijacija[i];
  if (sledecaVarijacija(n, varijacija))
    ispisi(varijacija);
  else
    cout << "-" << endl;
  return 0;
}
```

Задатак: Све варијације

Поставка: 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 ɡенерисање 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 $k-1$. Имплементацију ћемо организовати тако да уместо да враћа колекцију варијација, рекурзивна функција прима делимично попуњен низ који ће на све могуће начине допуњавати варијацијама текуће дужине k (која ће се смањивати кроз рекурзивне позиве). Дакле, на текућу позицију у низу (она се израчунава као разлика између дужине низа и тренутне вредности k) постављамо једну по једну вредност од 1 до n и затим рекурзивно позивамо функцију да попуни остатак низа (тиме што смањујемо дужину k и тиме прелазимо на наредну позицију).

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.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje tekucu varijaciju na standardni izlaz
void obradi(const vector<int>& varijacija) {
  for (int x : varijacija)
    cout << x << " ";
  cout << endl;
}

// sve varijacije duzine k elemenata skupa {1, ..., n}
// Dati niz varijacija duzine varijacije.size() - k
// se dopunjuje svim mogucim varijacijama sa ponavljanjem
// duzine k skupa {1, ..., n} i sve tako
// dobijene varijacije se obrađuju
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 prosirujemo
    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);
}


int main() {
  int n, k;
  cin >> n >> k;
  obradiSveVarijacije(k, n);
  return 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.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje tekucu varijaciju na standardni izlaz
void obradi(const vector<int>& varijacija) {
  for (int x : varijacija)
    cout << x << " ";
  cout << endl;
}

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;
}

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));
}

int main() {
  int n, k;
  cin >> n >> k;
  obradiSveVarijacije(k, n);
  return 0;
}
```

Задатак: Сви бинарни низови без суседних јединица

Поставка: 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.

Један начин је да дефинишемо рекурзивну функцију која генерише тражене бројеве.

Она добија префикс дужине i и покушава на све начине да га прошири (кроз рекурзију проширујемо низ карактера у старту алоциран на дужину n дужину i његовог попуњеног дела). 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.

```
#include <iostream>
#include <string>

using namespace std;

void obradi(const string& binarni) {
  cout << binarni << endl;
}

void obradiSveBinarneBez11(string& binarni, int i) {
  if (i == binarni.size())
    obradi(binarni);
  else {
    binarni[i] = '0';
    obradiSveBinarneBez11(binarni, i+1);
    if (i == 0 || binarni[i-1] != '1') {
      binarni[i] = '1';
      obradiSveBinarneBez11(binarni, i+1);
    }
  }
}

void obradiSveBinarneBez11(int n) {
  string binarni(n, '0');
  obradiSveBinarneBez11(binarni, 0);
}

int main() {
  int n;
  cin >> n;
  obradiSveBinarneBez11(n);
  return 0;
}
```

Једна могућност је да се употреби функција која генерише наредну у лексикографском поретку бинарну ниску без суседних јединица (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.

```
#include <iostream>
```

```
#include <string>

using namespace std;

void obradi(const string& binarni) {
  cout << binarni << endl;
}

bool sledeciBinarniBez11(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;
}

void obradiSveBinarneBez11(int n) {
  string binarni(n, '0');
  do {
    obradi(binarni);
  } while (sledeciBinarniBez11(binarni));
}


int main() {
  int n;
  cin >> n;
  obradiSveBinarneBez11(n);
  return 0;
}
```

Задатак: Све комбинације

Поставка: 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 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.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje kombinaciju na standarndi izlaz
void obradi(const vector<int>& kombinacija) {
  for (int x : kombinacija)
    cout << x << " ";
  cout << endl;
}

// 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);
}

int main() {
  int k, n;
  cin >> k >> n;
  obradiSveKombinacije(k, n);
  return 0;
}
```

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.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje kombinaciju na standarndi izlaz
void obradi(const vector<int>& kombinacija) {
  for (int x : kombinacija)
    cout << x << " ";
  cout << endl;
}

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);
}

int main() {
  int k, n;
  cin >> k >> n;
```

```
  obradiSveKombinacije(k, n);
  return 0;
}
```

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.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje kombinaciju na standarndi izlaz
void obradi(const vector<int>& kombinacija) {
  for (int x : kombinacija)
    cout << x << " ";
  cout << endl;
}

// pronalazi sledecu kombinaciju u leksikografskom redosledu
bool sledecaKombinacija(int n, vector<int>& kombinacija) {
  // tražena dužina kombinacije
  int k = kombinacija.size();

  // krećemo od kraja i tražimo prvu poziciju koja nije na maksimumu
  // tj. koja se može povećati. Maksimumi od kraja su n, n-1, n-2, ...
  int i;
  for (i = k-1; i >= 0 && kombinacija[i] == n; i--, n--)
    ;
  // ako takva pozicija ne postoji, tekuća kombinacija je maksimalna
  if (i < 0)
    return false;
  // uvećavamo poslednji element koji se može povećati
  kombinacija[i]++;
  // iza njega slažemo redom brojeve za jedan veće
  for (i++; i < k; i++)
    kombinacija[i] = kombinacija[i-1] + 1;
  return true;
}


// 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));
}

int main() {
  int k, n;
  cin >> k >> n;
  obradiSveKombinacije(k, n);
  return 0;
}
```

Лексикографски следећа комбинација - оптимизовано решење

Постоји и мала оптимизација претходног поступка. Наиме, ако знамо вредност преломне тачке у једном кораку, без поновне претраге можемо одредити вредност преломне тачке у наредном кораку. Кључно питање је то да ли након увећања преломне вредности она достиже свој максимум.

- Ако достиже тј. ако после увећања важи $komb_i = n-k+i+1$, тада после увећања преломне вредности, редом иза преломне тачке морају наћи елементи који су сви на својим максимумима, међутим, то је већ случај тако да није потребно поново их ажурирати. Наредна преломна тачка је непосредно испред текуће преломне тачке. На пример, наредна комбинација за 1356 је 1456. Преломна тачка је $i = 1$, важи да је $komb_1 = 3 = 6 - 4 + 1$ и довољно је само увећати елемент 3 на 4. Пошто су елементи од 4, 5 и 6, на својој максималној вредности, знамо да је наредна преломна вредност 1, па је наредна комбинација 2345.
- Ако након увећања преломна вредност она не достиже свој максимум тј. ако након увећања важи $komb_i < n - k + i + 1$, онда је након увећања и попуњавања низа до краја последњи елемент сигурно испод своје максималне вредности, тако да је наредна преломна тачка последња позиција у низу.

Интересантно, ова "оптимизација" не доноси никакву значајну добит и нема праткичних импликација. Ако обрада укључује било какву нетривијалну операцију или испис комбинације на екран, обрада потпуно доминира временом генерисања. Ако је обрада тривијална (на пример, само увећање глобалног бројача за један) тада не постоји значајна разлика у времену извршавања. Разлог томе је то што је у већини случајева преломна тачка последњи елемент или је врло близу десног краја, па је линеарна претрага брза.

```
#include <iostream>
#include <vector>

using namespace std;
```

```
// ispisuje kombinaciju na standarndi izlaz
void obradi(const vector<int>& kombinacija) {
  for (int x : kombinacija)
    cout << x << ” ”;
  cout << endl;
}

// 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;

   // obrađujemo prvu kombinaciju
   obradi(kombinacija);

   // specijalno obrađujemo slučaj n = k kada
   // nema drugih kombinacija
   if (n == k) return;

   // prva prelomna tačka je poslednja pozicija u nizu
   int i = k-1;
   while (i >= 0) {
      // ažuriramo kombinaciju
      kombinacija[i]++;
      // ako je uvećani prelomni element dostigao maksimum
      if (kombinacija[i] == n - k + i + 1)
        // naredna prelomna tačka je neposredno pre njega
        i--;
      else {
        // popunjavamo niz do kraja
        for (int j = i+1; j < k; j++)
           kombinacija[j] = kombinacija[j-1] + 1;
        // naredna prelomna tačka je poslednji element niza
        i = k-1;
     }
     // obrađujemo dobijenu kombinaciju
     obradi(kombinacija);
   }
}

int main() {
  int k, n;
  cin >> k >> n;
  obradiSveKombinacije(k, n);
```

```
  return 0;
}
```

Задатак: Све пермутације

Поставка: 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 пермутација

Рекурзивно генерисање пермутација у лексикографском редоследу је веома компликовано, тако да ћемо се одрећи услова да пермутације морају бити поређане лексикографски.

У том случају можемо поступити на следећи начин. На прву 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. 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. Разматрамо позицију $k - 1$. 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 исписујемо пермутацију. Ако је $k > 1$, тада је ситуација компликованија. 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. 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.

```
#include <iostream>
#include <vector>

using namespace std;

// ispisuje permutaciju na standarndi izlaz
void obradi(const vector<int>& permutacija) {
  for (int x : permutacija)
    cout << x << " ";
  cout << endl;
}

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);
}

int main() {
```

```
  int n;
  cin >> n;
  obradiSvePermutacije(n);
  return 0;
}
```

Одређивање следеће пермутације

Све пермутације у лексикографском редоследу се могу добити тако што се крене од почетне пермутације $1, 2, \ldots, n$ и у сваком кораку се исписује текућа пермутација и мења се са наредном пермутацијом у лексикографском поретку.

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

using namespace std;

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]);
  // obrcjemo deo niza od pozicije i+1 do kraja
  for (j = n - 1, i++; i < j; i++, j--)
    swap(a[i], a[j]);
  return true;
}

void obradiSvePermutacije(int n) {
  vector<int> permutacija(n);
  for (int i = 0; i < n; i++)
    permutacija[i] = i + 1;
  do {
    obradi(permutacija);
  } while (sledecaPermutacija(permutacija));
}
```

```
int main() {
  int n;
  cin >> n;
  obradiSvePermutacije(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.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> permutacija(n);
  iota(begin(permutacija), end(permutacija), 1);
  do {
    for (int x : permutacija)
      cout << x << " ";
    cout << endl;
  } while (next_permutation(begin(permutacija), end(permutacija)));
}
```

Задатак: Сви n-тоцифрени бројеви са датим збиром цифара

Поставка: Написати програм који исписује све n-тоцифрене бројеве који имају дати збир цифара.

Улаз: Прва линија садржи збир k ($1 \leq k \leq 9n$), а друга број цифара n ($2 \leq n \leq 100$).

Излаз: На стандардни излаз исписати све тражене бројеве, уређене по величини.

Пример

Улаз	*Излаз*
24	699
3	789
	798
	879
	888
	897
	969
	978
	987
	996

Решење: Наивно решење у ком би се генерисали сви n-тоцифрени бројеви (слично као у задатку Све варијације, а затим филтрирали они чији је збир цифара једнак датом броју је прилично неефикасно.

Овај задатак је донекле сличан задатку Све једноцифрене партиције, па боље решење можемо засновати на рекурзивној процедури генерисања свих партиција приказаној у том задатку. Основна разлика у овом задатку је то што се у овом задатку захтева да је број елемената у свакој партицији једнак датом броју цифара. Зато партицију исписујемо тј. излаз из рекурзије вршимо само ако је њена тренутна дужина баш једнака том броју и ако је преостала вредност параметра n постала једнака нули. Рекурзивни корак вршимо само ако је у партицији преостало још места тј. ако је број тренутно попуњених елемената строго мањи од задатог броја цифара. Још једна ситна разлика у односу на задатак Све једноцифрене партиције је то што је допуштено коришћење цифре нула (на свим позицијама, осим на почетној).

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

using namespace std;

void ispisi(vector<int> particija, int k) {
  for (int i = 0; i < k; i++)
    cout << particija[i];
  cout << endl;
}

void ispisiJednocifreneParticije(int n, vector<int>& particija, int k) {
  if (n == 0 && k == particija.size())
    ispisi(particija, k);
  else if (k < particija.size())
    for (int c = k == 0 ? 1 : 0; c <= min(9, n); c++) {
      particija[k] = c;
      ispisiJednocifreneParticije(n-c, particija, k+1);
    }
}
```

```
void ispisiJednocifreneParticije(int n, int brojCifara) {
  vector<int> particija(brojCifara);
  ispisiJednocifreneParticije(n, particija, 0);
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  int brojCifara;
  cin >> brojCifara;
  ispisiJednocifreneParticije(n, brojCifara);
  return 0;
}
```

Одсецање на основу доње īранице за сваку цифру

У претходном решењу се на сваку позицију постављају све цифре од нуле (или евентуално 1, на почетној позицији), па до 9 (уз евентуално одсецање када је број n мањи од 9). Тако, на пример, може да се деси да приликом покушаја генерисања троцифрених бројева са збиром цифара 27 на месту прве цифре испробавамо вредности од 1 до 8, а да заправо ни са једном од њих не можемо да попунимо партицију до краја (јер је једино решење 999). Много ефикасније решење добијамо ако применимо још једно одсецање и одредимо доњу границу вредности текуће цифре. Наиме, максимални могући збир цифара иза текуће се лако добије као $9(m-1)$, где је m број тренутно непопуњених цифара у партицији. Ако је текућа цифра једнака c, тада мора да важи да је преостали збир цифара n мањи или једнак $c+9(m-1)$, одакле се добија граница да је $c \geq n-9(m-1)$. Дакле, у петљи која поставља текућу цифру крећемо од веће од вредности $c+9(m-1)$ и вредности 0 (тј. 1 на почетној позицији), а завршавамо са мањом од вредности n и 9. С обзиром на овако одређене границе, имамо гаранцију да ће свака партиција моћи успешно да се попуни и при изласку из рекурзије само треба да контролишемо да ли су све цифре партиције потпуно попуњене (ако јесу, тада ће вредност преосталог збира n сигурно бити једнака нули).

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

using namespace std;

void ispisi(vector<int> particija) {
  for (int x : particija)
    cout << x;
  cout << endl;
}

void ispisiJednocifreneParticije(int n, vector<int>& particija, int k) {
  if (k == particija.size()) {
```

```
    ispisi(particija);
  } else {
    int preostaloCifara = particija.size() - k;
    int maksZbirIza = 9 * (preostaloCifara - 1);
    int minC = max(k == 0 ? 1 : 0, n - maksZbirIza);
    int maksC = min(9, n);
    for (int c = minC; c <= maksC; c++) {
      particija[k] = c;
      ispisiJednocifreneParticije(n-c, particija, k+1);
    }
  }
}

void ispisiJednocifreneParticije(int n, int brojCifara) {
  vector<int> particija(brojCifara);
  ispisiJednocifreneParticije(n, particija, 0);
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  int brojCifara;
  cin >> brojCifara;
  ispisiJednocifreneParticije(n, brojCifara);
  return 0;
}
```

Глава 3

Алгоритми сортирања

Задатак: Близанци

Поставка: Марија и Петар су близанци и желимо да свакоме од њих двоје купимо по једно одело као поклон за рођендан, али тако да се цене та два поклона што мање разликују (при томе није битно чији поклон ће бити скупљи).

Написати програм који учитава цене свих женских одела и свих мушких одела, а одређује и исписује најмању разлику између цена женског и мушког одела.

Улаз: Опис улаза: са стандардног улаза се учитава:

- у првом реду број мушких одела m ($1 \leq m \leq 50000$),
- у другом реду m целих бројева (цели бројеви између 1 и $2 \cdot 10^9$ раздвојени по једним размаком) - цене мушких одела
- у трећем реду број женских одела z ($1 \leq z \leq 50000$)
- и у четвртом реду z целих бројева (цели бројеви између 1 и $2 \cdot 10^9$ раздвојени по једним размаком) - цене женских одела.

Излаз: На стандардни излаз исписати најмању вредност разлике цена мушког и женског одела.

Пример

Улаз	*Излаз*
5 4680 2120 7940 11530 17820 4 850 13420 5770 6300	1090

Објашњење

Најмања разлика се постиже када се купе одела чије су цене 4680 и 5770 динара.

Решење:

Анализа

Наивно решење

Један могући приступ је да одредимо разлику (прецизније, апсолутну вредност разлике) у цени између сваког мушког и сваког женског одела, па од тих разлика нађемо најмању. Овакво решење ће дати тачан резултат у мањим примерима, али на већим примерима се неће извршити на време.

```
#include <iostream>
#include <vector>
#include <cmath>
#include <limits>
using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int n1; cin >> n1;
  vector<int> a1(n1);
  for (int i = 0; i < n1; i++)
    cin >> a1[i];
  int n2; cin >> n2;
  vector<int> a2(n2);
  for (int i = 0; i < n2; i++)
    cin >> a2[i];

  int minRazlika = numeric_limits<int>::max();
  for (int i1 = 0; i1 < n1; i1++)
    for (int i2 = 0; i2 < n2; i2++)
      minRazlika = min(minRazlika, abs(a1[i1] - a2[i2]));

  cout << minRazlika << endl;

  return 0;
}
```

Упоредни пролаз кроз уређене низове

Ефикаснији приступ је да се низови цена најпре сортирају, а да се затим истовремено пролази кроз оба низа, рачунајући разлику текућих елемената и напредујући у оном низу у којем је цена тренутно мања. Успут се, наравно, по потреби ажурира најмања забележена разлика. Када се стигне до краја било којег низа, поступак је завршен и најмања забележена разлика је тада и укупно најмања.

Заиста, пошто су низови сортирани, када се упореде почетни елементи из оба низа, онај који је мањи од њих нема потребе упоређивати са осталим елементима низа коме он не припада, јер ће разлика моћи бити само већа (јер је тај низ сортиран). Тај елемент онда можемо избацити из даљег разматрања тако што ћемо у низу у ком се он налази прећи на следећи елемент. У специјалном случају када су почетни елементи оба низа једнаки, разлика је једнака нули, што је најмања могућа разлика, па нема

потребе вршити даљу анализу.

На пример, нека су након сортирања вредности једнаке следећим.

```
1 14 28 33 45
8 21 22 41 56 68
```

- Прво поредимо елементе 1 и 8. Разлика је 7. Разлика између броја 1 и свих даљих бројева у доњем низу је већа од 7, па број 1 не морамо више анализирати.
- Након тога поредимо бројеве 14 и 8 и добијамо разлику 6. Разлика између броја 8 и свих бројева иза 14 је већа, па сада ни 8 не морамо више анализирати.
- Поредимо сада бројеве 14 и 21, разлика је 7, а 14 не морамо више да анализирамо.
- И разлика између 28 и 21 је 7, а број 21 не морамо више да анализирамо.
- Разлика између 28 и 22 је 6, а 22 не морамо да анализирамо даље.
- Разлика између 28 и 41 је 13, а 28 не морамо да анализирамо даље.
- Разлика између 33 и 41 је 8, а 33 не морамо да анализирамо даље.
- Разлика између 45 и 41 је 4, а 41 не морамо да анализирамо даље.
- Разлика између 45 и 56 је 11, а 45 не морамо да анализирамо даље. Пошто нема више елемената у горњем низу, поступак се завршава.

Можемо закључити да је најмања могућа разлика једнака 4 (за бројеве 41 и 45).

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <limits>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int n1; cin >> n1;
  vector<int> a1(n1);
  for (int i = 0; i < n1; i++)
    cin >> a1[i];
  int n2; cin >> n2;
  vector<int> a2(n2);
  for (int i = 0; i < n2; i++)
    cin >> a2[i];

  sort(begin(a1), end(a1));
  sort(begin(a2), end(a2));
  int i1 = 0, i2 = 0;

  int minRazlika = numeric_limits<int>::max();
```

```
  while (i1 < n1 && i2 < n2)
    if (a1[i1] <= a2[i2]) {
      minRazlika = min(minRazlika, a2[i2] - a1[i1]);
      i1++;
    } else {
      minRazlika = min(minRazlika, a1[i1] - a2[i2]);
      i2++;
    }

  cout << minRazlika << endl;

  return 0;
}
```

Задатак: Атп листа

Поставка: Током године играју се многи тениски турнири на којима тенисери освајају поене. Поени се сабирају и на крају године објављује се завршна листа на којој су тенисери рангирани на основу укупног броја поена током те године. Напиши програм који на основу резултата свих турнира одређује k најбољих тенисера и њихове поене (претпоставити да ће сви тенисери имати различит број поена).

Улаз: Са стандардног улаза се учитава број турнира n, а затим подаци о освојеним поенима на тим турнирима. За сваки турнир се учитава број m који представља број тенисера који су освајали поене, и након тога m линија које садрже податке о освојеним поенима тенисера на том турниру (презиме тенисера које има највише 50 карактера и број поена између 10 и 2000). На крају се уноси број k који одређује колико најбољих тенисера треба одредити.

Излаз: На стандардни излаз исписати презимена и укупан освојени број поена за k најбољих тенисера у опадајућем редоследу укупног освојеног броја поена.

Пример

Улаз	*Излаз*
3	Djokovic 2600
3	Nadal 2400
Djokovic 1000	Federer 1400
Nadal 800	
Federer 600	
3	
Nadal 1000	
Federer 800	
Djokovic 600	
3	
Djokovic 1000	
Cicipas 800	
Nadal 600	
3	

Решење:

Анализа

Смернице за алгоритам

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

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  map<string, int> bodovi;

  int broj_turnira;
  cin >> broj_turnira;
  for (int i = 0; i < broj_turnira; i++) {
    int broj_tenisera;
    cin >> broj_tenisera;
    for (int j = 0; j < broj_tenisera; j++) {
      string teniser; int bodovi_na_turniru;
      cin >> teniser >> bodovi_na_turniru >> ws;
      bodovi[teniser] += bodovi_na_turniru;
    }
  }

  int k;
  cin >> k;
  vector<pair<string, int>> najbolji(k);
  partial_sort_copy(bodovi.begin(), bodovi.end(),
                    najbolji.begin(), najbolji.end(),
                    [](const pair<string, int>& p1, const pair<string, int>& p2) {
                      return p1.second > p2.second;
                    });
  for (auto p : najbolji)
    cout << p.first << " " << p.second << endl;
  return 0;
}
```

Задатак: Разврставање по општинама

Поставка: Државна комисија је направила списак свих такмичара у држави. Потребно је да се свакој општини дистрибуира списак такмичара са територије те општине, али тако да редослед остане исти какав је на полазном списку државне комисије.

Улаз: Са стандардног улаза се уноси списак такмичара, све до краја улаза. За сваког такмичара се уноси назив општине и шифра такмичара, раздвојене једним табулатором (карактером Tab). У тексту су коришћени само ASCII карактери. Напомена: приликом интерактивног тестирања крај улаза се може унети помоћу `ctrl + z` тј. `ctrl`

`+ d`.

Излаз: На стандардни излаз исписати спискове за све општине, сваки у посебном реду. Општине су уређене лексикографски, растуће. Сваки списак почиње називом општине који је праћен знаком `:`. Након тога се наводе шифре свих такмичара раздвојене запетама (знацима `,`). Иза сваког знака интерпункције (знака `:` и знака `,`) наводи се по један размак.

Пример

Улаз

```
vozdovac    programmer
svilajnac   teamX
vozdovac    astro
alibunar    mika
svilajnac   pera.peric
medijana    luke skywalker
```

Излаз

```
alibunar: mika
medijana: luke skywalker
svilajnac: teamX, pera.peric
vozdovac: programmer, astro
```

Решење: Најефикаснији начин да извршимо разврставање елемената по општинама је линеарне сложености, али подразумева да се елементи из полазног премештају у помоћни низ. Разврставање вршимо у неколико пролаза. У првом одређујемо број такмичара у свакој општини. У другом одређујемо позиције у низу на које ћемо смештати такмичаре из сваке општине, док у трећем вршимо пребацивање из полазног у резултујући низ (ажурирајући при том позиције такмичара за сваку општину).

Дакле, први пролаз је сличан као код сортирања пребројавањем које смо срели у задатку Сортирање пребројавањем. За пример дат у тексту задатка, у првом пролазу одређујемо наредне бројеве такмичара за сваку општину.

```
alibunar: 2
medijana: 1
svilajnac: 2
vozdovac: 2
```

Позицију првог такмичара из сваке општине у резултујућем низу можемо одредити као укупан број такмичара у свим општинама које јој претходе (последњу позицију добијамо ако урачунамо и број такмичара у тој општини). Те позиције можемо одредити инкрементално (као, на пример, у задатку Префикс највећег збира).

На тај начин добијамо следеће позиције

```
alibunar: 0
medijana: 2
svilajnac: 3
vozdovac: 5
```

Приликом преписивања пролазимо редом кроз елементе полазног низа и смештамо их у резултујући низ на позицију придружену тој општини, уједно повећавајући ту позицију за 1, како би наредни елемент дошао на своје место.

Тако се `programer` уписује на место број 5, а позиција наредног такмичара са општине `vozdovac` се повећава на 6, затим се `teamX` уписује на место број 3, а позиција наредног

такмичара са општине `svilajnac` се повећева на 4, затим се `astro` upisuje na mesto broj 6, а позиција за општину `vozdovac` се повећава на 7 итд.

Из претходне дискусије је јасно да је свакој општини је потребно придружити број такмичара са те општине, а након тога и позицију у резултујућем низу на коју ће се приликом преписивања из полазног у резултујући низ придружити наредни такмичар из те општине. Пошто списак општина није унапред фиксиран, за та придруживања морамо користимо пресликавање (мапу, речник). Овакве структуре података смо већ користили у задатку Фреквенције речи. У језику C++ можемо користити мапу тј. `map<string, int>`.

Сложеност алгоритма зависи од броја такмичара и броја општина. Ако претпоставимо да је број такмичара n много већи од броја општина (што је прилично реална претпоставка), можемо закључити да ће временом доминирати два пролаза кроз оригинални низ и под претпоставком да ажурирање података придружених општинама тумачимо као да су операције константне сложености, укупна сложеност ће бити $O(n)$.

Приметимо да је претходни алгоритам изводио *стабилно сортирање* тј. да је редослед свих такмичара са исте општине задржан.

```
#include <iostream>
#include <map>
#include <vector>
#include <string>

using namespace std;

// za svakog takmicara je poznata opstina sa koje dolazi i sifra
struct Takmicar {
  string opstina;
  string sifra;
};

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavanje vektora takmicara
  vector<Takmicar> takmicari;
  string s;
  while (getline(cin, s)) {
    int p = s.find('\t');
    string opstina = s.substr(0, p);
    string sifra = s.substr(p + 1);
    takmicari.push_back({opstina, sifra});
  }

  // izracunavamo broj takmicara iz svake opstine
  // za dati naziv opstine brojTakmicara odredjuje broj takmicara
  map<string, int> brojTakmicara;
```

```
  // prolazimo kroz spisak svih takmicara
  for (auto takmicar : takmicari)
    // uvecavamo broj takmicara na opstini trenutnog takmicara
    brojTakmicara[takmicar.opstina]++;

  // izracunavamo poziciju u sortiranom nizu na kojoj
  // pocinju takmicari sa date opstine
  map<string, int> pozicije;
  // ukupan broj takmicara u do sada obradjenim opstinama
  int prethodnoTakmicara = 0;
  // prolazimo kroz sve opstine u sortiranom redosledu
  // (abecedno, leksikografski)
  for (auto it : brojTakmicara) {
    // tekuca opstina pocinje na poziciji odredjenoj
    // brojem takmicara u prethodnim opstinama
    pozicije[it.first] = prethodnoTakmicara;
    // uvecavamo broj takmicara u prethodnim opstinama za broj
    // takmicara u trenutnoj opstini, pripremajuci se za novu
    // iteraciju
    prethodnoTakmicara += it.second;
  }

  // konacan sortirani niz takmicara
  vector<Takmicar> sortirano(takmicari.size());
  // prolazimo kroz sve takmicare iz polaznog niza
  for (auto takmicar : takmicari) {
    // postavljamo takmicara na tekuce mesto u njegovoj opstini
    sortirano[pozicije[takmicar.opstina]] = takmicar;
    // uvecavamo slobodnu poziciju u toj opstini
    pozicije[takmicar.opstina]++;
  }

  // ispisujemo konacni spisak u trazenom formatu
  cout << sortirano[0].opstina << ": " << sortirano[0].sifra;
  for (int i = 1; i < sortirano.size(); i++)
    if (sortirano[i].opstina != sortirano[i-1].opstina)
      cout << endl << sortirano[i].opstina << ": " << sortirano[i].sifra;
    else
      cout << ", " << sortirano[i].sifra;
  cout << endl;

  return 0;
}
```

У решењу можемо применити библиотечке функције за стабилно сортирање (описане у задатку Разврставање по првом слову). Ипак, уместо тога демонстрираћемо да се стабилно сортирање може остварити и обичном библиотечком функцијом сортирања, коју смо описали у задатку Сортирање такмичара). У језику C++ то је функција `sort`.

Та функција не мора обавезно бити стабилна, међутим стабилност можемо наметнути кроз функцију поређења која одређује редослед сортирања. У структуру којом се репрезентују такмичари, поред општине и шифре такмичара уписаћемо и редни број такмичра на оригиналном списку. Функција поређења онда прво пореди општину, а ако су два такмичара који се пореде из исте општине, редослед одређује на основу редних бројева са оригиналног списка (овакво хијерархијско, тј. лексикографско поређење смо увели у задатку Пунолетство).

```
#include <iostream>
#include <map>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

// za svakog takmicara je poznata opstina sa koje dolazi i sifra
// pamtimo jos i redni broj u polaznom nizu
struct Takmicar {
  string opstina;
  string sifra;
  size_t redniBroj;
};

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavanje vektora takmicara
  vector<Takmicar> takmicari;
  string s;
  while (getline(cin, s)) {
    int p = s.find('\t');
    string opstina = s.substr(0, p);
    string sifra = s.substr(p + 1);
    takmicari.push_back({opstina, sifra, takmicari.size()});
  }

  // sortiranje takmicara
  sort(takmicari.begin(), takmicari.end(),
       [](const Takmicar& t1, const Takmicar& t2) {
     // takmicari se sortiraju najpre na osnovu opstine,
     // a u okviru iste opstine na osnovu rednog broja u originalnom nizu
     return t1.opstina < t2.opstina ||
            t1.opstina == t2.opstina && t1.redniBroj < t2.redniBroj;
       });

  // ispisujemo konacni spisak u trazenom formatu
  cout << takmicari[0].opstina << ": " << takmicari[0].sifra;
```

```
  for (int i = 1; i < takmicari.size(); i++)
    if (takmicari[i].opstina != takmicari[i-1].opstina)
      cout << endl << takmicari[i].opstina << ": " << takmicari[i].sifra;
    else
      cout << ", " << takmicari[i].sifra;
  cout << endl;

  return 0;
}
```

Задатак: Сортирање бројева вишеструким разврставањем (RadixSort)

Поставка: Дат је низ природних бројева. Имплементирај програм који их сортира алгоритмом вишеструког разврставања (RadixSort).

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 50000$), а затим n природних бројева између 1 и 999999999 (сваки у посебном реду).

Излаз: На стандардни излаз испиши те бројеве у неопадајућем редоследу.

Пример

Улаз	*Излаз*
9	43
342	342
43	432
5432	546
432	636
34222	3423
3423	5432
546	6363
6363	34222
636	

Решење: У задатку Разврставање по првом слову видели смо да се разврставање може применити више пута (под претпоставком да се имплементира на стабилан начин) да би се низ сортирао у лексикографском редоследу кључева представљених одређеним n-торкама, задржавајући при том оригинални редослед елемената чији су кључеви једнаки.

Ако замислимо да су природни бројеви допуњени водећим нулама тако да су сви бројеви који се сортирају са истим бројем цифара, сортирање природних бројева се своди на сортирање торки у лексикографском редоследу и може се извршити вишеструким разврставањем, тако што се сортирање врши на основу сваке декадне позиције, кренувши од цифре јединица.

Прикажимо како се сортирање разврставањем може применити на следећи низ бројева.

```
37 140 7 10 99 102 17 25 1 48 14 3 18
```

Прво вршимо сортирање на основу цифре јединица

```
140 10 1 102 3 14 25 37 7 17 48 18 99
```

Након тога, на основу цифре десетица.

```
1 102 3 7 10 14 17 18 25 37 140 48 99
```

На крају, сортирање вршимо на основу цифре стотина.

```
1 3 7 10 14 17 18 25 37 48 99 102 140
```

Свако појединачно разврставање вршимо на уобичајени начин (као, на пример, у задатку Разврставање по општинама). У првом пролазу пребројимо елементе у свакој категорији (за сваку цифру од 0 до 9 пребројимо колико има елемената низа који на текућој декадној позицији садрже баш ту цифру). Након тога израчунамо парцијалне збирове тако добијеног низа и они нам за сваку категорију дају позицију у новом низу на коју треба поставити елемент из те категорије. На крају копирамо елементе из полазног у нови низ на одговарајуће позиције (ажурирајући те позиције након сваког уписа).

```
#include <iostream>
#include <vector>
#include <limits>

using namespace std;

void sortiranjeRazvrstavanjem(vector<int>& a, int s) {
  // brojimo pojavljivanje svake cifre uz koeficijent s (stepen desetke)
  int frekvencija[10] = {0};
  for (int i = 0; i < a.size(); i++)
    frekvencija[(a[i] / s) % 10]++;
  // pozicije u rezultujucem nizu ispred kojih se zavrsavaju grupe
  // elemenata sa odgovarajucim ciframa uz koeficijent s
  for (int i = 1; i < 10; i++)
    frekvencija[i] += frekvencija[i-1];
  // prepisujemo elemente u pomocni niz
  vector<int> pom(a.size());
  for (int i = a.size() - 1; i >= 0; i--)
    pom[--frekvencija[(a[i] / s) % 10]] = a[i];
  // vracamo elemente nazad u glavni niz
  a = pom;
}

void sortiranjeVisestrukimRazvrstavanjem(vector<int>& a) {
  // odredjujemo maksimum niza a
  int max = numeric_limits<int>::min();
  for (int x : a)
    if (x > max)
      max = x;

  // sortiramo na osnovu svake cifre krenuvsi od cifara jedinica
```

```
  for (int s = 1; max / s > 0; s *= 10)
    sortiranjeRazvrstavanjem(a, s);
}

int main() {
  // ucitavamo niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  sortiranjeVisestrukimRazvrstavanjem(a);

  // ispisujemo rezultat
  for (int x : a)
    cout << x << endl;

  return 0;
}
```

Глава 4

Бинарна претрага

Задатак: Збир што мање узастопних

Поставка: Дуж једне улице деца продају слаткише. Јована има G динара и жели да их потроши тако што ће кренути да се шета дуж улице и од сваког детета, редом, ће купити тачно један слаткиш (ни једно дете неће прескочити). Ако су познате цене свих слаткиша и ако је позната позиција са које Јована креће у куповину, одредити број слаткиша које ће Јована на тај начин купити.

Улаз: Са стандардног улаза се уноси број деце који продају слаткише n ($1 \leq n \leq 5 \cdot 10^4$), а затим и низ који садржи цене слаткиша (позитивни природни бројеви мањи од 100). Након тога се уноси број упита k ($1 \leq k \leq 5 \cdot 10^4$), а затим у k наредних редова упити који садрже позицију првог детета које ће Јована обићи (позиције се броје од нуле), а затим број динара које Јована има.

Излаз: На стандардни излаз за сваки упит у посебном реду исписати број слаткиша које ће Јована купити.

Пример

Улаз	*Излаз*
7 3 5 1 2 3 1 4 4 3 1 2 5 2 13 0 10	0 2 5 3

Објашњење

У првом упиту Јована има један динар, а креће од детета које продаје слаткиш за 2 динара, па не може да купи ни један слаткиш.

У другом упиту Јована има пет динара и купује слаткише који коштају 1 и 2 динара.

У трећем упиту Јована има 13 динара и купује слаткише који коштају 1, 2, 3, 1 и 4 динара.

У четвртом упиту Јована има 10 динара и купује слаткие који коштају 3, 5 и 1 динар.

Решење:

Анализа

Смернице за алгоритам

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

using namespace std;

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  int n;
  cin >> n;
  vector<int> a(n);
  vector<int> zbir_prefiksa_cena(n+1);
  zbir_prefiksa_cena[0] = 0;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    zbir_prefiksa_cena[i+1] = zbir_prefiksa_cena[i] + x;
  }
  int q;
  cin >> q;
  for (int i = 0; i < q; i++) {
    int p, novac;
    cin >> p >> novac;
    auto poc = next(begin(zbir_prefiksa_cena), p);
    auto kraj = end(zbir_prefiksa_cena);
    auto it = upper_bound(poc, kraj, zbir_prefiksa_cena[p] + novac);
    cout << distance(poc, it) - 1 << '\n';
  }
  return 0;
}
```

Задатак: Пуно фигурица

Поставка: Друштвену игру игра k играча и сваки играч игру почиње са по k фигура, при чему све фигуре једног играча морају бити исте јачине, док су јачине фигура различитих играча различите. Фигуре су доступне у неограниченим количинама, при чему је познат низ јачина доступних фигура. Напиши програм који одређује највећи број играча k који могу играти игру тако да разлика између укупне јачине свих фигура било која два играча не пређе задату границу.

Улаз: Са стандардног улаза се учитава број n ($1 \leq n \leq 10^5$), а затим у наредном реду n различитих расположивих јачина фигура (природни бројеви између 1 и 10^5). Наредни ред садржи границу (природни број између 1 и 10^{12}).

Излаз: На стандардни излаз исписати два броја раздвојена размаком – број k и најмању разлику укупних јачина фигура када игра k играча.

Пример

Улаз	*Излаз*
5 5 4 2 7 3 15	4 12

Објашњење

Ако играчи узму по четири фигуре јачине 5, 4, 2 и 3 највећа разлика јачина фигура играча биће $4 \cdot 5 - 4 \cdot 2 = 12$. Ако би играло 5 играча, морали би да узму и фигуре јачине 7 и највећа разлика би била $5 \cdot 7 - 5 \cdot 2 = 25$, што је веће од дозвољене границе.

Решење:

Наивно решење

Задатак се може решити тако што се за свако k између 2 и n провери да ли је могуће игру одиграти са k играча.

Централно питање је како за дато k изабрати k играча, тако да разлика између укупне јачине фигура најјачег и најслабијег међу њима буде што мања. Ефикасан алгоритам се може направити ако се низ јачина фигура претходно сортира. Играчи тада треба да узимају фигуре чије су јачине узастопних k елемената низа (ако би неки играч заменио фигуре, добила би се већа разлика између најјачег и најслабијег играча). Зато је довољно проверити све узастопне k-точлане поднизове низа (којих има $n - k$), одузети последњи од првог члана и помножити резултат са k (јер сваки играч узима по k фигура).

Пошто се са порастом броја играча разлика може само повећати, претрагу можемо прекинути када први пут нађемо вредност k такву да игру не могу играти k играча. Сложеност таквог алгоритма, суштински заснованог на линеарног претрази је $O(n^2)$.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <limits>

using namespace std;

long long najmanja_razlika_k(const vector<int>& a, int k) {
  int min = numeric_limits<int>::max();
  for (size_t i = 0; i + k - 1 < a.size(); i++)
    if (a[i+k-1] - a[i] < min)
      min = a[i+k-1] - a[i];
  return min;
```

```
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  long long granica;
  cin >> granica;
  sort(begin(a), end(a));
  int k = 2;
  while (k <= n && k * najmanja_razlika_k(a, k) <= granica)
    k++;
  cout << k-1 << " " << (k-1) * najmanja_razlika_k(a, (k-1)) << endl;
  return 0;
}
```

Бинарна претрага

Брже решење можемо добити ако оптималну вредност k тражимо бинарном претрагом (то је могуће, јер вредности разлика расту са порастом k). Сложеност таквог алгоритма је $O(n \log n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <limits>
#include <cassert>

using namespace std;

long long najmanja_razlika_k(const vector<int>& a, int k) {
  int min = numeric_limits<int>::max();
  for (size_t i = 0; i + k - 1 < a.size(); i++)
    if (a[i+k-1] - a[i] < min)
      min = a[i+k-1] - a[i];
  return min;
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  long long granica;
  cin >> granica;
  sort(begin(a), end(a));
```

```
  int lK = 2, dK = n;
  while (lK <= dK) {
    int k = lK + (dK - lK) / 2;
    if (k * najmanja_razlika_k(a, k) <= granica)
      lK = k+1;
    else
      dK = k-1;
  }
  cout << dK << " " << dK * najmanja_razlika_k(a, dK) << endl;
  return 0;
}
```

Два показивача

Још једно брзо решење можемо да добијемо техником два показивача. За сваки леви крај сегмента одређујемо највећу могућу вредност десног краја која не прелази границу. Након сортирања користимо сегмент $[poc, kraj]$ који представља изабране фигуре. Овај сегмент је на почетку $[0, 0]$. Ако је разлика између укупне јачине најјачих и најслабијих фигура из сегмента довољно мала, продужавамо сегмент надесно (повећавамо $kraj$), а ако је већа од дозвољене, онда скраћујемо сегмент слева (повећавамо poc). Сигурни смо да након повећања левог краја, десни крај не морамо смањивати (размисли зашто). Успут памтимо највећу дужину сегмента и оптималну разлику при тој дужини сегмента. Нада прођемо цео низ јачина фигура, исписујемо коначне вредности највеће дужине сегмента и оптималне разлике. Сложеност оваквог решења је $O(n \log n)$ у почетној фази сортирања, а након тога, у другој фази је линеарна $O(n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <limits>
#include <cassert>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  long long granica;
  cin >> granica;

  sort(begin(a), end(a));
  int poc = 0, kraj = 0;
  int opt_k = 1;
  long long opt_razlika = 0;
  while (kraj < n) {
```

```
    long long k = kraj - poc + 1;
    long long razlika = k * (a[kraj] - a[poc]);
    if (razlika > granica)
      poc += 1;
    else {
      if (k > opt_k) {
        opt_k = k;
        opt_razlika = razlika;
      } else if (k == opt_k)
        opt_razlika = min(opt_razlika, razlika);
      kraj += 1;
    }
  }
  cout << opt_k << " " << opt_razlika << endl;
  return 0;
}
```

Глава 5

Геометријски алгоритми

Задатак: Колинеарне тачке

Поставка: Напиши програм који одређује колико тачака из датог скупа тачака припада датој правој.

Улаз: Са стандардног улаза се учитавају координате две различите тачке којима је одређена права (у првој линији се налазе два цела броја раздвојена размаком које представљају координате прве тачке, а у другој два цела броја раздвојена размаком које представљају координате друге тачке). Након тога се учитава број тачака n ($1 \leq n \leq 100$), а затим из наредних n линија координате тих тачака (свака линија садржи два цела броја раздвојена размаком).

Излаз: На стандардни излаз исписати тражени број тачака које припадају правој.

Пример

Улаз	*Излаз*

`1 2`
`3 4`
`5`
`-8 -7`
`-8 -9`
`1 2`
`2 3`
`0 0`
	`3`

Решење: Тачке $A = (x_1, y_1)$, $B = (x_2, y_2)$ и $C = (x_3, y_3)$ су колинеарне ако и само ако су вектори $\vec{AB} = (x_2 - x_1, y_2 - y_1, 0)$ и $\vec{AC} = (x_3 - x_1, y_3 - y_1, 0)$ колинеарни, тј. ако им је векторски производ 0. До овог закључка се може доћи и ако се примети да су тачке коолинеарне ако и само ако је површина троугла који образују једнака нули (а површина троугла једнака је половини интензитета векторског производа).

```
#include <iostream>
```

```cpp
using namespace std;

void vektorski_proizvod(int x1, int y1, int z1,
                        int x2, int y2, int z2,
                        int& x, int& y, int& z) {
  // izracunavamo determinantu:
  //  i  j  k
  //  x1 y1 z1
  //  x2 y2 z2
  x = y1*z2 - y2*z1;
  y = z1*x2 - x1*z2;
  z = x1*y2 - x2*y1;
}

bool kolinearni_vektori(int x1, int y1, int z1,
                        int x2, int y2, int z2) {
  int x, y, z;
  vektorski_proizvod(x1, y1, z1,
                     x2, y2, z2,
                     x, y, z);
  return x == 0 && y == 0 && z == 0;
}

bool kolinearne_tacke(int x1, int y1, int x2, int y2, int x3, int y3) {
  return kolinearni_vektori(x1 - x2, y1 - y2, 0,
                            x1 - x3, y1 - y3, 0);
}

int main() {
  int x1, y1;
  cin >> x1 >> y1;
  int x2, y2;
  cin >> x2 >> y2;
  int n;
  cin >> n;
  int broj = 0;
  for (int i = 0; i < n; i++) {
    int x, y;
    cin >> x >> y;
    if (kolinearne_tacke(x1, y1, x2, y2, x, y))
      broj++;
  }
  cout << broj << endl;
  return 0;
}
```

Пошто претпостављамо да вектори $\vec{AB}$ и $\vec{AC}$ припадају равни xOy, њихов векторски производ је ортогоналан на ту раван. Заиста, његове координате су $((x_2 - x_1)(y_3 -$

$y_1)-(y_2-y_1)(x_3-x_1),0,0)$ и он је једнак нули ако и само ако је $(x_2-x_1)(y_3-y_1)=(y_2-y_1)(x_3-x_1)$.

```
#include <iostream>

using namespace std;

bool kolinearne_tacke(int x1, int y1, int x2, int y2, int x3, int y3) {
  return (x1-x2)*(y1-y3) == (y1-y2)*(x1-x3);
}

int main() {
  int x1, y1;
  cin >> x1 >> y1;
  int x2, y2;
  cin >> x2 >> y2;
  int n;
  cin >> n;
  int broj = 0;
  for (int i = 0; i < n; i++) {
    int x, y;
    cin >> x >> y;
    if (kolinearne_tacke(x1, y1, x2, y2, x, y))
      broj++;
  }
  cout << broj << endl;
  return 0;
}
```

Задатак: Тачка у троуглу

Поставка: Напиши програм који проверава да ли се тачка налази у унутрашњости троугла.

Улаз: Са стандардног улаза се уноси 8 реалних бројева из интервала $[-10,10]$, заокружених на две децимале. У првом реду се уносе x и y координата тачке која се анализира, а у наредна три реда x и y координате темена троугла. Међу 4 унете тачке нема колинеарних.

Излаз: На стандардни излаз исписати `da` ако тачка припада троуглу или `ne` ако не припада.

Пример 1		**Пример 2**	
Улаз	*Излаз*	*Улаз*	*Излаз*
`0.00 2.00`	`da`	`-3.50 3.50`	`ne`
`-3.00 -3.00`		`-3.00 -3.00`	
`-1.00 4.00`		`-1.00 4.00`	
`3.00 2.00`		`3.00 2.00`	

Решење: Тачка T је унутар троугла ABC ако и само ако је површина троугла ABC

једнака збиру површина троуглова ABT, ATC и TBC.

```
#include <iostream>
#include <cmath>

using namespace std;

// tolerancija
const double EPS = 1e-6;

// tacka je zadata svojim dvema koordinatama
struct Tacka {
  double x, y;
  Tacka(double x_, double y_) {
    x = x_; y = y_;
  }
};

double povrsinaTrougla(const Tacka& A, const Tacka& B, const Tacka& C) {
  // pertle (cik-cak)
  return abs(+ A.x*B.y + B.x*C.y + C.x*A.y
             - A.x*C.y - C.x*B.y - B.x*A.y) / 2.0;
}

bool tackaUTrouglu(const Tacka& T,
                   const Tacka& A, const Tacka& B, const Tacka& C) {
  // racunamo povrsinu trougla ABC
  double P = povrsinaTrougla(A, B, C);
  // racunamo povrsine trouglova TBC, TAC i TAB
  double P1 = povrsinaTrougla(T, B, C);
  double P2 = povrsinaTrougla(A, T, C);
  double P3 = povrsinaTrougla(A, B, T);
  // proveravamo da li one u zbiru daju povrsinu trougla ABC
  // poredimo dve realne vrednosti na jednakost
  return abs(P - (P1 + P2 + P3)) < EPS;
}


int main() {
  double x, y;
  cin >> x >> y;
  Tacka P(x, y);
  cin >> x >> y;
  Tacka A(x, y);
  cin >> x >> y;
  Tacka B(x, y);
  cin >> x >> y;
  Tacka C(x, y);
```

```
  if (tackaUTrouglu(P, A, B, C))
    cout << "da" << endl;
  else
    cout << "ne" << endl;
  return 0;
}
```

Тачка T је унутар троугла ABC ако и само ако је тачка T са исте стране праве AB као и тачка C, са исте стране праве BC као и тачка A и са исте стране тачке AC као и тачка B тј. ако сви троуглови ABT, BCT и CAT имају исту оријентацију.

```
#include <iostream>
#include <cmath>

using namespace std;

// tolerancija
const double EPS = 1e-6;

// tacka je zadata svojim dvema koordinatama
struct Tacka {
  double x, y;
  Tacka(double x_, double y_) {
    x = x_; y = y_;
  }
};

// moguce orijentacije trojke tacaka
enum Orijentacija {POZITIVNA, KOLINEARNE, NEGATIVNA};

Orijentacija orijentacija(const Tacka& A, const Tacka& B, const Tacka& C) {
  double d = (B.x-A.x)*(C.y-A.y) - (C.x-A.x)*(B.y-A.y);
  if (abs(d) < EPS)
    return KOLINEARNE;
  else if (d > EPS)
    return POZITIVNA;
  else
    return NEGATIVNA;
}

bool tackaUTrouglu(const Tacka& T,
                   const Tacka& A, const Tacka& B, const Tacka& C) {
  Orijentacija o1 = orijentacija(A, B, T);
  Orijentacija o2 = orijentacija(B, C, T);
  Orijentacija o3 = orijentacija(C, A, T);
  return o1 == o2 && o2 == o3;
}
```

```
int main() {
  double x, y;
  cin >> x >> y;
  Tacka P(x, y);
  cin >> x >> y;
  Tacka A(x, y);
  cin >> x >> y;
  Tacka B(x, y);
  cin >> x >> y;
  Tacka C(x, y);
  if (tackaUTrouglu(P, A, B, C))
    cout << "da" << endl;
  else
    cout << "ne" << endl;
  return 0;
}
```

Задатак: Са исте стране

Поставка: Напиши програм који утврђује колико се датих тачака налази са исте стране праве као и једна задатака тачка.

Улаз: Са стандардног улаза се уносе две различите тачке A и B које одређују праву, затим тачка T, након тога број тачака n ($10 \leq n \leq 100$), и након тога n тачака. Свака тачка је задата у посебном реду, помоћу своје две целобројне координате раздвојене једним размаком.

Излаз: Опис излазних података.

Пример

Улаз	*Излаз*
1 1 2 2 0 1 5 3 4 7 5 2 6 -1 -5 -5 -2	3

Објашњење

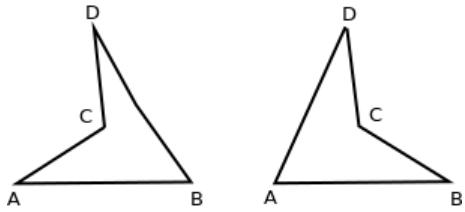
Тачке $(3, 4)$, $(2, 6)$ и $(-5, -2)$ се налазе са исте стране праве одређене тачкама $(1, 1)$ и $(2, 2)$ као и тачка $(0, 1)$.

Решење: Тачка T_1 и $_2$ су са исте стране праве B ако и само ако је оријентација тројки ABT_1 и ABT_2 једнака. Оријентација тројке ABT се може одредити на основу знака векторског производа вектора AT и BT.

```cpp
#include <iostream>

using namespace std;

// moguce orijentacije trojke tacaka
enum Orijentacija {POZITIVNA, KOLINEARNE, NEGATIVNA};

Orijentacija orijentacija(int xa, int ya, int xb, int yb, int xc, int yc) {
  int d = (xb-xa)*(yc-ya) - (xc-xa)*(yb-ya);
  if (d > 0)
    return POZITIVNA;
  else if (d < 0)
    return NEGATIVNA;
  else
    return KOLINEARNE;
}

bool saIsteStrane(int xa, int ya, int xb, int yb,
                  int x1, int y1, int x2, int y2) {
  return orijentacija(xa, ya, xb, yb, x1, y1) ==
    orijentacija(xa, ya, xb, yb, x2, y2);
}

int main() {
  int xa, ya;
  cin >> xa >> ya;
  int xb, yb;
  cin >> xb >> yb;
  int x1, y1;
  cin >> x1 >> y1;
  int n;
  cin >> n;
  int broj = 0;
  for (int i = 0; i < n; i++) {
    int x2, y2;
    cin >> x2 >> y2;
    if (saIsteStrane(xa, ya, xb, yb, x1, y1, x2, y2))
      broj++;
  }
  cout << broj << endl;
  return 0;
}
```

Задатак: Конструкција простог многоугла

Поставка: Прост многоугао је многоугао у ком се никоје две странице не секу (осим што се суседне странице додирују у заједничком темену). Напиши програм који за



Слика 5.1: Тачке A, B, C и D и два различита проста многоугла која оне одређују.

дати скуп тачака (у ком нису све тачке колинеарне) одређује неки прост многоугао ком је скуп темена једнак том скупу тачака.

Улаз: Са стандрадног улаза се учитава број n ($3 \leq n \leq 50000$), а затим и n тачака (свака тачка је описана у посебном реду помоћу своје две целобројне координате раздвојене размаком). Све учитане тачке су различите и нису све колинеарне.

Излаз: На стандардни излаз исписати пермутацију учитаног низа тачака која представља редослед темена конструисаног простог многоугла (тачке треба да буду описане на исти начин као и на улазу).

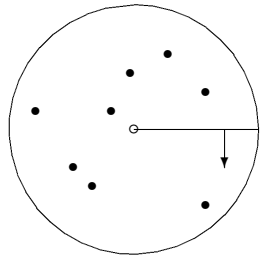
Пример

Улаз	*Излаз*
`5`	`5 1`
`3 1`	`5 2`
`0 4`	`2 3`
`5 1`	`0 4`
`2 3`	`3 1`
`5 2`	

Решење: Некада је за дати скуп тачака у равни могуће конструисати више различитих простих многоуглова, односно решење није једнозначно одређено (слика 5.1).

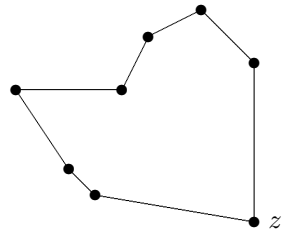
Нека је C неки круг, чија унутрашњост садржи све тачке. За налажење таквог круга довољно је израчунати највеће међу растојањима произвољне тачке равни (центра круга) до свих осталих тачака – сложеност овог корака је $O(n)$. Површина C може се "пребрисати" (прегледати) ротирајућом полуправом којој је почетак центар C (слика 5.2). Претпоставимо за тренутак да ротирајућа права у сваком тренутку садржи највише једну тачку. Очекујемо да ћемо спајањем тачака оним редом којим полуправа наилази на њих добити прост многоугао. Докажимо то. Означимо тачке, уређене у складу са редоследом наиласка полуправе на њих, са $P_1, P_2, \ldots, P_n$ (прва тачка бира се произвољно). За свако i, $1 \leq i \leq n$, страница P_iP_{i-1} (односно P_1P_n за $i = 1$) садржана је у новом (дисјунктном) исечку круга, па се не сече ни са једном другом страницом. Ако би ово тврђење било тачно, добијени многоугао би морао да буде прост. Међутим, угао између полуправих кроз неке две узастопне тачке P_i и P_{i+1} може да буде већи од π. Тада исечак који садржи дуж P_iP_{i+1} садржи више од пола круга и није конвексна фигура, а дуж P_iP_{i+1} пролази кроз друге исечке круга, па може да сече друге странице многоугла. Да бисмо се уверили да је то могуће, довољно је да

уместо нацртаног замислимо круг са центром ван круга са слике 5.2. Ово је пример специјалних случајева на које се наилази при решавању геометријских проблема.



Слика 5.2: Пролазак тачака у кругу ротирајућом полуправом.

Да би се решио уочени проблем, могу се, на пример, фиксирати произвољне три неколинеарне тачке из скупа, а за центар круга изабрати нека тачка унутар њима одређеног троугла (на пример тежиште, које се лако налази). Овакав избор гарантује да ни један од добијених сектора круга неће имати угао већи од π. Затим сортирамо тачке према положају у кругу са центром z. Прецизније, сортирају се углови између x-осе и полуправих од z ка осталим тачкама. Ако две или више тачака заклапају исти угао са x-осом, оне се даље сортирају растуће према растојању од тачке z. На крају, тачке повезујемо у складу са добијеним уређењем, по две узатопне. Пошто све тачке леже лево од тачке z, до дегенерисаног случаја о коме је било речи не може доћи. Прост многоугао добијен овим поступком за тачке са слике 5.2 приказан је на слици 5.3. Основна компонента временске сложености овог алгоритма потиче од сортирања. Сложеност алгоритма је дакле $O(n \log n)$.



Слика 5.3: Конструкција простог многоугла.

Угао φ који права $y = mx+b$ заклапа са x осом добија се коришћењем везе $m = \tan\varphi$, односно из једначине $\varphi = \arctan m$. Међутим, углови се не морају експлицитно израчунавати. Углови се користе само за налажење редоследа којим треба повезати тачке и исти редослед добија се уређењем нагиба одговарајућих полуправих; то чини непотребним израчунавање аркустангенса. Из истог разлога непотребно је израчунавање растојања кад две тачке имају исти нагиб — довољно је израчунати квадрате растојања. Дакле, нема потребе за израчунавањем квадратних коренова.

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

```cpp
#include <cmath>

using namespace std;

// tacka je zadata svojim dvema koordinatama
struct Tacka {
  int x, y;
  Tacka(int x_ = 0, int y_ = 0) {
    x = x_; y = y_;
  }
};

bool kolinearne(const Tacka& t1, const Tacka& t2, const Tacka& t3) {
  return (t1.x-t2.x)*(t1.y-t3.y) == (t1.y-t2.y)*(t1.x-t3.x);
}

double kvadratRastojanja(double x1, double y1, double x2, double y2) {
  double dx = x1 - x2, dy = y1 - y2;
  return dx*dx + dy*dy;
}

void prostMnogougao(vector<Tacka>& tacke) {
  int i = 2;
  while (kolinearne(tacke[0], tacke[1], tacke[i]))
    i++;
  double x0 = (tacke[0].x + tacke[1].x + tacke[i].x) / 3.0;
  double y0 = (tacke[0].y + tacke[1].y + tacke[i].y) / 3.0;
  sort(begin(tacke), end(tacke),
       [x0, y0](const Tacka& t1, const Tacka& t2) {
         double x1 = t1.x - x0, y1 = t1.y - y0;
         double x2 = t2.x - x0, y2 = t2.y - y0;
         double ugao1 = atan2(y1, x1);
         double ugao2 = atan2(y2, x2);
         const double EPS = 1e-12;
         if (ugao1 < ugao2 - EPS) {
           return true;
         }
         if (ugao2 < ugao1 - EPS) {
           return false;
         }
         return kvadratRastojanja(x0, y0, x1, y1) <
           kvadratRastojanja(x0, y0, x2, y2);
       });
}


int main() {
```

```
  int n;
  cin >> n;
  vector<Tacka> tacke(n);
  for (int i = 0; i < n; i++) {
    int x, y;
    cin >> x >> y;
    tacke[i] = Tacka(x, y);
  }
  prostMnogougao(tacke);
  for (const Tacka& t : tacke)
    cout << t.x << " " << t.y << endl;
  return 0;
}
```

Уместо тежишта троугла одређеног са неке три неколинеарне тачке из скупа, за центар круга може се узети и једна од тачака из скупа — тачка z са највећом x-координатом (и са најмањом y-координатом, ако има више тачака са највећом x-координатом). Овакву тачку ћемо често користити приликом репавања геометријских проблема и зваћемо је *екстремна тачка*. Овако одабрану тачку повезујемо са свим осталим тачкама и тачке сортирамо растуће у односу на угао који полуправа од тачке z заклапа са x-осом. Ако две или више тачака заклапају исти угао са x-осом, оне се даље сортирају према растојању од тачке z и то на следећи начин: уколико првих неколико тачака заклапају исти угао, њих сортирамо у растућем редоследу растојања од тачке z, уколико је последњих неколико тачака колинеарно са тачком z њих сортирамо у опадајућем редоследу растојања од тачке z, док је за остале тачке које су колинеарне са тачком z све једно да ли ћемо их сортирати растуће или опадајуће.

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

using namespace std;

// tacka je zadata svojim dvema koordinatama
struct Tacka {
  int x, y;
  Tacka(int x_ = 0, int y_ = 0) {
    x = x_; y = y_;
  }
};

enum Orijentacija { POZITIVNA, NEGATIVNA, KOLINEARNE };

Orijentacija orijentacija(const Tacka& t0, const Tacka& t1, const Tacka& t2) {
  int d = (t1.x-t0.x)*(t2.y-t0.y) - (t2.x-t0.x)*(t1.y-t0.y);
  if (d > 0)
    return POZITIVNA;
  else if (d < 0)
    return NEGATIVNA;
```

```
  else
    return KOLINEARNE;
}

int kvadratRastojanja(const Tacka& t1, const Tacka& t2) {
  int dx = t1.x - t2.x, dy = t1.y - t2.y;
  return dx*dx + dy*dy;
}

void prostMnogougao(vector<Tacka>& tacke) {
  // trazimo tacku sa maksimalnom x koordinatom,
  // u slucaju da ima vise tacaka sa maksimalnom x koordinatom
  // biramo onu sa najmanjom y koordinatom
  auto max = max_element(begin(tacke), end(tacke),
                         [](const Tacka& t1, const Tacka& t2) {
                           return t1.x < t2.x ||
                                  (t1.x == t2.x && t1.y > t2.y);
                         });
  // dovodimo je na početak niza - ona predstavlja centar kruga
  swap(*begin(tacke), *max);
  const Tacka& t0 = tacke[0];

  // sortiramo ostatak niza (tačke sortiramo na osnovu ugla koji
  // zaklapaju u odnosu vertikalnu polupravu koja polazi naviše iz
  // centra kruga), a kolinearne na osnovu rastojanja od centra kruga
  sort(next(begin(tacke)), end(tacke),
       [t0](const Tacka& t1, const Tacka& t2) {
         Orijentacija o = orijentacija(t0, t1, t2);
         if (o == KOLINEARNE)
           return kvadratRastojanja(t0, t1) <= kvadratRastojanja(t0, t2);
         return o == POZITIVNA;
       });

  // obrcemo redosled tacaka na poslednjoj pravoj
  auto it = prev(end(tacke));
  while (orijentacija(*prev(it), *it, t0) == KOLINEARNE)
    it = prev(it);
  reverse(it, end(tacke));
}


int main() {
  int n;
  cin >> n;
  vector<Tacka> tacke(n);
  for (int i = 0; i < n; i++) {
    int x, y;
```

```
    cin >> x >> y;
    tacke[i] = Tacka(x, y);
  }
  prostMnogougao(tacke);
  for (const Tacka& t : tacke)
    cout << t.x << " " << t.y << endl;
  return 0;
}
```

Задатак: Припадност тачке конвексном многоуглу

Поставка: Напиши програм који утврђује да ли тачка припада конвексном многоуглу (ивице многоугла сматрају се његовим делом).

Улаз: Са стандардног улаза се учитава број темена конвексног многоугла n ($3 \leq n \leq 50000$), а затим редом темена у редоследу супротном од казаљке на сату. Свако теме се задаје у посебном реду, помоћу два цела броја раздвојена једним размаком. Након тога се задаје број m ($1 \leq m \leq 50000$) тачака чију припадност многоуглу треба испитати, а затим у наредних m редова координате тих тачака (координате су целобројне, раздвојене једним размаком).

Излаз: За сваку од m тачака на стандардни излаз у посебном реду исписати `da` ако тачка припада многоуглу тј. `ne` у супротном.

Пример

Улаз	*Излаз*
6	da
4 0	da
2 2	ne
-2 2	ne
-4 0	da
-2 -2	
2 -2	
5	
-2 2	
-3 1	
-5 1	
3 -2	
0 0	

Решење: Сваки конвексни многоугао се може поделити на троуглове тако што се повуку све дијагонале из његовог произвољног темена. Тачка припада многоуглу ако и само ако припада неком од тих троуглова. Припадност тачке троуглу већ смо анализирали у задатку Тачка у троуглу. Нагласимо да је проверу потребно мало прилагодити да би се у обзир узело то да се ивице многоугла сматрају његовим делом. Ако једна тачка лежи на датој дужи она ће се сматрати да је са исте стране те дужи као и било која друга тачка. Такође, пошто су вредности координата у овом задатку релативно велики бројеви, потребно је приликом израчунавања векторског производа обратити пажњу на могућност настанка прекорачења.

Пошто се припадност троуглу може испитати у времену $O(1)$, а посматрамо $n-2$ троугла, сложеност испитивања припадности једне тачке је $O(n)$, док је сложеност провере за свих m тачака $O(mn)$.

Задатак можемо ефикасније решити бинарном претрагом. Свака дијагонала многоугао дели на два мања многоугао. Ако се установи да је тачка са једне стране дијагонале, знамо да она не може да припада оном од та два многоугла који лежи са супротне стране те дијагонале и потребно је испитати само да ли тачка припада оном другом многоуглу. Ако се дијагонала увек бира тако да та два многоугла имају приближно исти број темена, проблем ће се сводити на проблем истог облика, али двоструко мање димензије, што даје ефикасан алгоритам. Половљење прекидамо када остане троугао и тада проверу припадности троуглу вршимо слично као у задатку Тачка у троуглу (прилагодивши поступак томе да се ивице троугла сматрају његовим делом, као и могућности настанка прекорачења приликом израчунавања векторског производа, због релативно великих координата).

Пошто се у сваком кораку димензија проблема полови, сложеност припадности једне тачке је $O(\log n)$, док је сложеност провере за свих m тачака $O(m \log n)$.

```
#include <iostream>
#include <vector>
#include <utility>

using namespace std;

typedef pair<int, int> Tacka;

enum Orijentacija {POZITIVNA, NEGATIVNA, KOLINEARNE};

Orijentacija orijentacija(Tacka a, Tacka b, Tacka c) {
  int xa = a.first, ya = a.second;
  int xb = b.first, yb = b.second;
  int xc = c.first, yc = c.second;
  long long d = (long long)(xb-xa)*(long long)(yc-ya) -
                (long long)(xc-xa)*(long long)(yb-ya);
  if (d > 0)
    return POZITIVNA;
  else if (d < 0)
    return NEGATIVNA;
  else
    return KOLINEARNE;
}

bool saIsteStrane(const Tacka& T1, const Tacka& T2,
                  const Tacka& A1, const Tacka& A2) {
  Orijentacija o1 = orijentacija(T1, T2, A1);
  Orijentacija o2 = orijentacija(T1, T2, A2);
  if (o1 == KOLINEARNE || o2 == KOLINEARNE)
    return true;
```

```
  return o1 == o2;
}

bool tackaUTrouglu(const Tacka& T1, const Tacka& T2, const Tacka& T3,
                   const Tacka& A) {
  return saIsteStrane(T1, T2, T3, A) &&
         saIsteStrane(T1, T3, T2, A) &&
         saIsteStrane(T2, T3, T1, A);
}

bool sadrzi(const vector<Tacka>& poligon, const Tacka& A, int l, int d) {
  if (d - l == 1)
    return tackaUTrouglu(poligon[0], poligon[l], poligon[d], A);
  int s = l + (d - l) / 2;
  if (orijentacija(poligon[0], poligon[s], A) == POZITIVNA)
    return sadrzi(poligon, A, s, d);
  else
    return sadrzi(poligon, A, l, s);
}

bool sadrzi(const vector<Tacka>& poligon, const Tacka& A) {
  int n = poligon.size();
  return sadrzi(poligon, A, 1, n-1);
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  int n;
  cin >> n;
  vector<Tacka> poligon(n);
  for (int i = 0; i < n; i++) {
    cin >> poligon[i].first >> poligon[i].second;
  }

  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    Tacka A;
    cin >> A.first >> A.second;
    if (sadrzi(poligon, A))
      cout << ”da” << '\n';
    else
      cout << ”ne” << '\n';
  }

  return 0;
```

```
}
```

```
#include <iostream>
#include <vector>
#include <utility>

using namespace std;

typedef pair<int, int> Tacka;

enum Orijentacija {POZITIVNA, NEGATIVNA, KOLINEARNE};

Orijentacija orijentacija(Tacka a, Tacka b, Tacka c) {
  int xa = a.first, ya = a.second;
  int xb = b.first, yb = b.second;
  int xc = c.first, yc = c.second;
  long long d = (long long)(xb-xa)*(long long)(yc-ya) -
                (long long)(xc-xa)*(long long)(yb-ya);
  if (d > 0)
    return POZITIVNA;
  else if (d < 0)
    return NEGATIVNA;
  else
    return KOLINEARNE;
}

bool saIsteStrane(const Tacka& T1, const Tacka& T2,
                  const Tacka& A1, const Tacka& A2) {
  Orijentacija o1 = orijentacija(T1, T2, A1);
  Orijentacija o2 = orijentacija(T1, T2, A2);
  if (o1 == KOLINEARNE || o2 == KOLINEARNE)
    return true;
  return o1 == o2;
}

bool tackaUTrouglu(const Tacka& T1, const Tacka& T2, const Tacka& T3,
                   const Tacka& A) {
  return saIsteStrane(T1, T2, T3, A) &&
         saIsteStrane(T1, T3, T2, A) &&
         saIsteStrane(T2, T3, T1, A);
}

bool sadrzi(const vector<Tacka>& poligon, const Tacka& A, int l, int d) {
  if (d - l == 1)
    return tackaUTrouglu(poligon[0], poligon[l], poligon[d], A);
  int s = l + (d - l) / 2;
  if (orijentacija(poligon[0], poligon[s], A) == POZITIVNA)
    return sadrzi(poligon, A, s, d);
```

```
  else
    return sadrzi(poligon, A, l, s);
}

bool sadrzi(const vector<Tacka>& poligon, const Tacka& A) {
  int n = poligon.size();
  return sadrzi(poligon, A, 1, n-1);
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);

  int n;
  cin >> n;
  vector<Tacka> poligon(n);
  for (int i = 0; i < n; i++) {
    cin >> poligon[i].first >> poligon[i].second;
  }

  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    Tacka A;
    cin >> A.first >> A.second;
    if (sadrzi(poligon, A))
      cout << "da" << '\n';
    else
      cout << "ne" << '\n';
  }

  return 0;
}
```

Глава 6

Похлепни алгоритми

Задатак: Шаховске екипе

Поставка: Шаховска екипа Београда је позвала на припреме шаховску екипу остатка Србије. Свака екипа има исти број такмичара и за сваког такмичара је познат рејтинг. Екипа домаћина има могућност да одабере парове који ће играти у првом колу. Ако сваки играч домаћина побеђује госта који има мањи или једнак рејтинг, а губи од госта који има строго већи рејтинг, напиши програм који одређује који је највећи број победа које екипа Београда може да оствари у првом колу.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 50000$), а затим у наредом реду рејтинзи играча екипе домаћина (природни бројеви) раздвојени размацима, а у наредом реду рејтинзи играча екипе гостију (природни бројеви) раздвојени размацима.

Излаз: На стандардни излаз исписати само један број који представља највећи могући број победа домаћина.

Пример

Улаз	*Излаз*

`4`
`2120 1985 2205 1842`
`2045 2100 1990 1980`
	`3`

Објашњење

Домаћин може да оствари највише три победе. На пример, играч 2205 може да добије играча 2100, играч 2120 може да добије играча 2045, а играч 1985 може да добије играча 1980.

Решење: Задатак можемо решити помоћу неколико различитих грамзивих алгоритама.

Једна могућност је да се парови формирају тако што се упари k најбољих домаћих играча са k најлошијих гостујућих играча. Та стратегија би била коректна, али њена имплементација није тривијална, јер није јасно колико максимално може да буде k.

Варијација коју ћемо једноставно имплементирати је следећа. Ако домаћин може да оствари бар једну победу, њу може да донесе најбољи домаћи играч. Наиме, ако би он изгубио свој меч, увек бисмо неког од играча који је добио меч могли заменити њиме (јер је он бољи од свих играча домаћина) и добити исти број победа. Поставља се питање са којим гостујућем играчем он треба да игра. Циљ нам је да након формирања тог пара преостану што лошији гостујући играчи, да би слабији играчи тима домаћина имали шансе да остваре победе. Јасно је да морамо да елиминишемо госте који су бољи од тог најбољег домаћег играча (јер њих нико од играча домаћина не може да победи), а од преосталих гостујућих играча можемо да изаберемо најбољег. Након елиминисања најбољег домаћег играча, свих гостију бољих од њега и госта са којим ће он да игра, проблем је сведен на проблем истог облика, али мање димензије. Излаз из овог рекурзивног поступка представља случај када су сви гостујући играчи бољи од најбољег међу преосталим домаћинима.

Приликом имплементације скупове домаћих и гостујућих играча можемо чувати у низовима уређеним у нерастућем редоследу рејтинга и алгоритам можемо реализовати техником два показивача (продискутоваћемо касније и остале могуће варијанте). Низ домаћина обилазимо редом, елемент по елемент, а низ гостију раздвајамо на оне које су елиминисани (оне који су до сада упарени и оне које нису упарени, али су бољи од текућег домаћег играча) и преостале. Одржавамо место почетка низа гостију који још нису обрађени и приликом тражења пара за текућег домаћег играча низ гостију обилазимо од те позиције. Сваког госта или елиминишемо, јер је бољи од текућег домаћег играча или га додељујемо текућем домаћем играчу и онда их обојицу елиминишемо. Нагласимо да се у имплементацији не морамо враћати на елиминисане госте, јер ако је неки гост бољи од текућег домаћина (најбољег међу преосталим), биће бољи и од свих наредних (преосталих) домаћина. Стога се оба показивача крећу само у једном смеру и сложеност фазе додељивања је линеарна. Укупним алгоритмом, дакле, доминира сложеност сортирања, па је укупна сложеност $O(n \log n)$.

Докажимо и формално коректност овакве грамзиве стратегије. Решење које претходни алгоритам даје задовољава услове задатка јер се сваки домаћин упарује са гостом која није бољи од њега (то се експлицитно проверава) и након упаривања се елиминише из разматрања, тако да смо сигурни да заиста постоји коректно упаривање за број победа који се враћа. Покажимо и да наша стратегија прави оптимални број победа за домаћу екипу. Доказ ће ићи техником размене, тј. тиме што ћемо се показати да се оптимално упаривање може трансформисати у оно добијено грамзивом стратегијом, одржавајући укупан број парова у којима домаћин побеђује (за играче домаћина који побеђују рећи ћемо да су добитно упарени). Посматрајмо неко оптимално упаривање. Нека је d_i најбољи домаћин који учествује у добитном упаривању. Ако он није укупно најбољи домаћин d_s, тада најбољи домаћин сигурно није добитно упарен. Можемо домаћина d_i избацити из добитног упаривања и њему придруженог госта g_i придружити укупно најбољем домаћину d_s (то је могуће јер је $d_s \geq d_i \geq g_i$). Такво упаривање је и даље оптимално (јер се број добитних парова за домаћина није променио). Нека је g_s гост која би био одабран стратегијом (најбољи гост која није бољи од d_s, тј. најбољи гост за кога важи $d_s \geq g_s$). Ако он није део тренутног добитног упаривања, онда госта g_i који тренутно игра са домаћином d_s можемо избацити и заменити њиме (то је могуће јер је $d_s \geq g_s$). Ако јесте распоређен да игра са неким d_j, онда можемо направити размену тако да d_s игра са g_s, а d_j са g_i. Докажимо да је ово и даље

коректно упаривање. Важи да је $d_s \geq g_s$ и $d_s \geq g_i$. Пошто је g_s најбољи гост кога d_s може да победи, важи да је $g_s \geq g_i$. Зато је $d_j \geq g_s \geq g_i$. Са ове две евентуалне размене добијамо и даље оптималан распоред који је у складу са нашом стратегијом што се тиче првог пара. Настављајући размене по истом принципу (тј. на основу индуктивног аргумента), упаривање можемо трансформисати у оно формирано нашом стратегијом, задржавајући све време оптималност.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

int main() {
   // ucitavamo ulazne podatke
   int n;
   cin >> n;
   vector<int> domaci(n);
   for (int i = 0; i < n; i++)
     cin >> domaci[i];
   vector<int> gosti(n);
   for (int i = 0; i < n; i++)
     cin >> gosti[i];

   // Ako domacini mogu da ostvare nekih k pobeda, onda tih k pobeda moze
   // da ostvari njihovih k najjacih igraca. Zato domacine obradjujemo u
   // opadajucem redosledu rejtinga i za svakog redom odredjujemo gosta kojeg
   // moze da pobedi. Za svakog domacina odredjujemo najjaceg gosta kojeg
   // moze da pobedi jer time slabijim igracima domace ekipe ostavljamo prostor
   // da pobede nekoga.

   // zelimo da i domacine i goste obilazimo u opadajucem redosledu rejtinga
   sort(begin(domaci), end(domaci), greater<int>());
   sort(begin(gosti), end(gosti), greater<int>());
   int brojPobeda = 0;
   // tekuci indeks domaceg i gostujuceg igraca
   int d = 0, g = 0;
   while (true) {
       // trazimo najjaceg gosta kojeg moze da pobedi domacin na poziciji d
       while (g < n && domaci[d] < gosti[g])
          g++;
       // ako takav gost ne postoji, ne mozemo povecati broj pobeda
       if (g >= n) break;
       // nasli smo gosta
       brojPobeda++;
       g++, d++;
   }
```

```
    cout << brojPobeda << endl;

    return 0;
}
```

Скренимо пажњу и на важност ефикасне имплементације. Посматрајмо разноврсности решења ради решење које је дуално оном претходно описаном. У тој грамзивој стратегији обрађујемо госте у неопадајућем редоследу рејтинга и сваком госту додељујемо што лошијег домаћина који може да га победи. Као што смо видели, у ефикасној имплементацији приликом преласка на сваког новог госта требало би домаћина тражити само међу онима који у ранијим корацима нису елиминисани (било тако што су упарени или тако што је установљено да не могу да победе неког од слабијих гостију). Ако претрагу домаћина сваки пут почињемо из почетка (водећи рачуна о томе да раније упарене домаћине не упарујемо поново, тако што у посебном низу региструјемо оне домаћине које смо већ упарили) добићемо алгоритам чија је сложеност најгорег случаја $O(n^2)$.

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

using namespace std;

int main() {
  // ucitavamo ulazne podatke
  int n;
  cin >> n;
  vector<int> domaci(n);
  for (int i = 0; i < n; i++)
    cin >> domaci[i];
  vector<int> gosti(n);
  for (int i = 0; i < n; i++)
    cin >> gosti[i];

  // Ako je moguce pobediti nekih k gostiju, onda tih k gostiju
  // mogu biti k najslabijih gostiju. Zato goste obradjujemo u
  // rastucem redosledu rejtinga i za svakog redom odredjujemo domacina koji
  // moze da pobedi tog gosta, a nije ranije uparen.
  sort(begin(domaci), end(domaci));
  sort(begin(gosti), end(gosti));
  int brojPobeda = 0;

  vector<bool> zauzet(n, false);
  for (int g = 0; g < n; g++)
    for (int d = 0; d < n; d++)
      if (!zauzet[d] && domaci[d] >= gosti[g]) {
    brojPobeda++;
    zauzet[d] = true;
```

```
    break;
      }

  cout << brojPobeda << endl;

  return 0;
}
```

Још једна исправна варијација на тему наше победничке стратегије (тј. њене дуалне варијанте) обилази све домаћине и госте у неопадајућем редоследу рејтинга и ако домаћин може да победи најслабијег тренутно нераспоређеног госта, онда га упарујемо са њим, а у супротном га упарујемо са најјачим гостом. Ту стратегију можемо имплементирати тако што чувамо скуп преосталих домаћина и гостију, проналазимо најмањи тј. највећи елемент у скупу и уклањамо их. Ако се скуп имплементира преко низа (вектора, листе), добијамо веома неефикасан алгоритам, квадратне сложености (јер се и проналажење минимума и максимума и уклањање елемента са дате позиције врши у линеарној сложености).

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


int main()
{
  int n;
  cin >> n;
  vector<int> domaci(n);
  for (int i = 0; i < n; i++)
    cin >> domaci[i];
  vector<int> gosti(n);
  for (int i = 0; i < n; i++)
    cin >> gosti[i];

  int brojPobeda = 0;

  // sve dok je duzina vektora razlicita od nule
  while (domaci.size() > 0) {
    int najmanjidom = *min_element(begin(domaci), end(domaci));
    auto it = find(begin(domaci), end(domaci), najmanjidom);
    domaci.erase(it);

    int najmanjigost = *min_element(begin(gosti), end(gosti));
    if (najmanjidom >= najmanjigost) {
      brojPobeda++;
      it = find(begin(gosti), end(gosti), najmanjigost);
      gosti.erase(it);
```

```
    } else {
      int najvecigost = *max_element(begin(gosti), end(gosti));
      it = find(begin(gosti), end(gosti), najvecigost);
      gosti.erase(it);
    }
  }
  cout << brojPobeda << endl;
}
```

Програм постаје много ефикаснији ако употребимо библиотечке колекције које нам пружају ефикаснију имплементацију скупа (која дозвољава ефикасно тражење и уклањање минималног и максималног елемента). У језику C++ можемо употребити мултискупове (јер можда постоји више играча са истим рејтингом) које на располагању имамо кроз колекцију `multiset`. Пошто је мултискуп уређен итератор `begin()` указује на најмањи елемент, а итератор `prev(end())` на највећи елемент. Уклањање елемента можемо извршити помоћу метода `erase`. Под претпоставком да се операције са мултискупом врше у логаритамској сложености у односу број елемената у мултискупу, овај алгоритам ће бити сложености $O(n \log n)$.

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


int main()
{
  int n;
  cin >> n;
  multiset<int> domaci;
  for (int i = 0; i < n; i++) {
    int x; cin >> x;
    domaci.insert(x);
  }
  multiset<int> gosti;
  for (int i = 0; i < n; i++) {
    int x; cin >> x;
    gosti.insert(x);
  }

  int brojPobeda = 0;

  // sve dok ne rasporedimo sve domace igrace
  while (domaci.size() > 0) {
    // rasporedjujemo najboljeg domacina
    int najmanjidom = *domaci.begin();
    domaci.erase(domaci.begin());
    // sa najlosijim gostom ako moze da ga pobedi ili sa najboljim
```

```
    // gostom ako ne moze
    int najmanjigost = *gosti.begin();
    if (najmanjidom >= najmanjigost) {
      brojPobeda++;
      gosti.erase(gosti.begin());
    } else {
      gosti.erase(prev(gosti.end()));
    }
  }
  cout << brojPobeda << endl;
}
```

Ипак најефикасније решење добијамо ако мултискупове представимо сортираним низом, а брисање не вршимо ефективно, већ само чувамо показивач на тренутно необрађеног домаћина, док у скупу гостију необрађене госте чувамо између два показивача.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

int main() {
  // ucitavamo ulazne podatke
  int n;
  cin >> n;
  vector<int> domaci(n);
  for (int i = 0; i < n; i++)
    cin >> domaci[i];
  vector<int> gosti(n);
  for (int i = 0; i < n; i++)
    cin >> gosti[i];

  // sortiramo oba tima u rastucemo redosledu rejtinga
  sort(begin(domaci), end(domaci));
  sort(begin(gosti), end(gosti));
  int brojPobeda = 0;
  // pozicija najslabijeg i najjaceg nerasporedjenog gosta
  int gSlabi = 0, gJaki = n-1;
  // rasporedjujemo sve domace igrace
  for (int d = 0; d < n; d++) {
    // rasporedjujemo najboljeg domacina sa najlosijim gostom ako moze
    // da ga pobedi ili sa najboljim gostom ako ne moze
    if (domaci[d] >= gosti[gSlabi]) {
      brojPobeda++;
      gSlabi++;
    } else {
```

```
      gJaki--;
    }
  }

  cout << brojPobeda << endl;

  return 0;
}
```

За веома мале улазне примере, задатак би могао да се реши и грубом силом, тако што бисмо покушали сва могућа упаривања.

```
#include <iostream>
#include <vector>

using namespace std;

int maksPobeda(const vector<int>& domaci, vector<int>& gosti, int k = 0) {
  if (k == domaci.size())
    return 0;
  int maks = 0;
  for (int i = k; i < gosti.size(); i++) {
    swap(gosti[i], gosti[k]);
    int pobeda = (domaci[k] >= gosti[k] ? 1 : 0) +
                 maksPobeda(domaci, gosti, k+1);
    if (pobeda > maks)
      maks = pobeda;
    swap(gosti[i], gosti[k]);
  }
  return maks;
}

int main() {
   // ucitavamo ulazne podatke
   int n;
   cin >> n;
   vector<int> domaci(n);
   for (int i = 0; i < n; i++)
     cin >> domaci[i];
   vector<int> gosti(n);
   for (int i = 0; i < n; i++)
     cin >> gosti[i];

   cout << maksPobeda(domaci, gosti) << endl;
}
```

Примећујемо да у претходној претрази долази до преклапања рекурзивних позива, па ствар можемо покушати да поправимо мемоизацијом. Параметар мемоизације може бити скуп преосталих гостију, који можемо кодирити битовима неозначеног целог бро-

ја. Пошто је мемоизациона табела огромна ово решење се може употребити само за релативно мале димензије улаза (пар десетина такмичара).

```
#include <iostream>
#include <vector>

using namespace std;

int maksPobeda(const vector<int>& domaci, const vector<int>& gosti,
               int k, vector<int>& memo, unsigned preostali) {
  int n = domaci.size();

  if (memo[preostali] != ~0)
    return memo[preostali];

  if (k == domaci.size())
    return 0;
  int maks = 0;

  for (int g = 0; g < n; g++) {
    if (preostali & (1 << g)) {
      int pobeda = (domaci[k] >= gosti[g] ? 1 : 0) +
        maksPobeda(domaci, gosti, k+1, memo, preostali ^ (1 << g));
      if (pobeda > maks)
        maks = pobeda;
    }
  }

  return memo[preostali] = maks;
}

int maksPobeda(vector<int>& domaci, vector<int>& gosti) {
  int n = domaci.size();
  vector<int> memo(1 << n, ~0);
  return maksPobeda(domaci,  gosti, 0, memo, (1 << n) - 1);
}

int main() {
   // ucitavamo ulazne podatke
   int n;
   cin >> n;
   vector<int> domaci(n);
   for (int i = 0; i < n; i++)
     cin >> domaci[i];
   vector<int> gosti(n);
   for (int i = 0; i < n; i++)
     cin >> gosti[i];
```

```
    cout << maksPobeda(domaci, gosti) << endl;
}
```

Задатак: Распоред активности

Поставка: У једном кабинету се суботом одржава обука програмирања. Сваки наставник је написао термин у ком жели да држи наставу (познат је сат и минут почетка и сат и минут завршетка часа). Одреди како је могуће направити распоред часова тако да што више наставника буде укључено.

Улаз: Са стандардног улаза се учитава прво број n (укупан број наставника, $1 \leq n \leq 50000$), а затим у n наредних редова по четири броја раздвојена размацима који представљају сат и минут почетка тј. завршетка часа (претпоставити да је завршетак увек иза почетка).

Излаз: На стандардни излаз исписати највећи број наставника који могу да одрже своје часове.

Пример

Улаз	*Излаз*
7	3
8 15 9 20	
10 45 11 30	
11 20 12 45	
9 30 12 40	
10 20 11 20	
12 00 13 00	
11 30 13 30	

Објашњење

Могу се одржати, на пример, часови од 8:15 до 9:20, затим час од 10:20 до 11:20 и на крају од 11:30 до 13:30.

Решење: Сваки час можемо представити паром бројева који представљају број минута од претходне поноћи до почетка и до краја часа (већ приликом учитавања сате и минуте можемо превести само у минуте).

Исцрпна претрага

Наивно решење се заснива на испитивању свих могућих подскупова скупа тачака који су такви да се сви часови могу одржати (никоја два часа из тог скупа се не секу). Испитивање постојања пресека два часа се може урадити на исти начин као у задатку Интервали - пресек постоји ако и само ако је каснији почетак часа после ранијег краја часа. Генерисање свих подскупова вршимо рекурзивно, бектрекингом на исти начин као у задатку Сви подскупови. С обзиром на велики број подскупова које треба испитати ово решење је веома неефикасно (сложеност му је експоненцијална).

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <utility>
```

```cpp
using namespace std;

// casove predstavljamo uredjenim parovima
typedef pair<int, int> cas;

cas napraviCas(int pocSat, int pocMin, int krajSat, int krajMin) {
  return make_pair(pocSat*60 + pocMin, krajSat*60 + krajMin);
}

inline int pocetakCasa(const cas& c) {
  return c.first;
}

inline int krajCasa(const cas& c) {
  return c.second;
}


// provera da li postoji presek dva casa
bool postojiPresek(const cas& a, const cas& b) {
  return max(pocetakCasa(a), pocetakCasa(b)) < min(krajCasa(a), krajCasa(b));
}

// odredjuje najveci broj casova koji se moze dobiti tako sto se na k
// casova sa pocetka vektora zakzani dodaju casovi iz vektora casovi
// od pozicije i nadalje
int najviseCasova(const vector<cas>& casovi, int i, vector<cas>& zakazani, int k) {
  if (i == casovi.size()) {
    // ako nema vise kandidata u vektoru casovi, najvise mozemo
    // odrzati k casova (onih iz vektora zakazani)
    return k;
  } else {
    // racunamo najveci broj casova koji se moze dobiti ako se
    // preskoci cas na poziciji i
    int bezItog = najviseCasova(casovi, i + 1, zakazani, k);
    int rez = bezItog;
    // proveravamo da li se cas na poziciji i moze odrzati tako sto
    // proveravamo da li taj cas ima presek sa nekim od vec zakazanih
    // casova
    bool mozeIti = true;
    for (int j = 0; j < k; j++)
      if (postojiPresek(casovi[i], zakazani[j])) {
    mozeIti = false;
    break;
      }
    // ako se moze odrzati i-ti cas
```

```
    if (mozeIti) {
      // dodajemo ga na kraj vektora zakazanih casova
      zakazani[k] = casovi[i];
      // proveravamo koliko se najvise casova moze odrzati kada je on
      // ukljucen
      int saItim = najviseCasova(casovi, i + 1, zakazani, k + 1);
      // rezultat je veci od broja casova bez i-tog i sa i-tim
      rez = max(rez, saItim);
    }
    return rez;
  }
}

// najveci broj casova koji se mogu odrzati
int najviseCasova(const vector<cas>& casovi) {
  vector<cas> zakazani(casovi.size());
  return najviseCasova(casovi, 0, zakazani, 0);
}

int main() {
  int n;
  cin >> n;
  vector<cas> casovi(n);
  for (int i = 0; i < n; i++) {
    int pocSat, pocMin, krajSat, krajMin;
    cin >> pocSat >> pocMin >> krajSat >> krajMin;
    casovi[i] = napraviCas(pocSat, pocMin, krajSat, krajMin);
  }
  cout << najviseCasova(casovi) << endl;

  return 0;
}
```

Грамзиви приступ

Ефикасније решење се може добити грамзивим приступом. Часове можемо сортирати неопадајуће на основу времена њиховог завршетка и обилазити их у том редоследу. Од свих нераспоређених часова бирамо онај који се најраније завршава и који се може одржати (не сече се са до сада одржаним часовима, тј. почиње након завршетка претходног часа). Интуитивно, таквим избором остављамо што већу могућност за распоређивање накнадних часова.

Формално, претпоставимо да је $O = \{c_1, \ldots, \ldots, c_k\}$, скуп часова који представља неко оптимално решење, при чему су часови c_1 до c_k сортирани неопадајуће по редоследу њиховог завршетка. Пошто се сви ти часови могу одржати, између њих нема преклапања и сваки наредни почиње након завршетка претходног. Нека је c_i први час у овом скупу који не би био изабран нашом стратегијом. Претпоставимо да би наша стратегија уместо њега одабрала час c_i'. Покажимо да се заменом часа c_i часом c_i' добија такође распоред који је оптималан. Покажимо да се c_i завршава после c_i'.

Заиста, ако је $i = 0$, тада наша стратегија бира c'_i који се први завршава, па се стога c_i не може завршавати пре њега. Ако је $i > 0$, тада знамо да се c_i мора да почиње после c_{i-1}, међутим, наша стратегија за c'_i бира онај час који почиње након c_{i-1} који се први завршава, па се c_i ни у овом случају не може завршавати пре c'_i. Ако постоје часови у O пре часа c_i, они остају непромењени и час c'_i се не крши са њима (јер је одабран тако да почиње након завршетка часа c_{i-1}). Пошто се c'_i не завршава касније него c_i он се сигурно не судара ни са једним часом из O који иде после c_i (јер сви они почињу и завршавају се након краја часа c_i). Дакле, када се c_i замени са c'_i и даље се добија исправан распоред са истим бројем одржаних часова као O за који смо претпоставили да је оптималан. По истом приницпу можемо мењати наредне часове (ово се мора зауставити јер у сваком наредном оптималном скупу имамо по један час више који је у складу са нашом стратегијом) и тако показати да ће наша стратегија вратити оптималан скуп.

Рецимо да постоји и дуално решење у којем се бира онај час који последњи почиње и часови се обилазе уназад, по нерастућем редоследу њиховог почетка.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <utility>

using namespace std;

// casove predstavljamo uredjenim parovima
typedef pair<int, int> cas;

cas napraviCas(int pocSat, int pocMin, int krajSat, int krajMin) {
  return make_pair(pocSat*60 + pocMin, krajSat*60 + krajMin);
}

inline int pocetakCasa(const cas& c) {
  return c.first;
}

inline int krajCasa(const cas& c) {
  return c.second;
}

int main() {
  int n;
  cin >> n;
  vector<cas> casovi(n);
  for (int i = 0; i < n; i++) {
    int pocSat, pocMin, krajSat, krajMin;
    cin >> pocSat >> pocMin >> krajSat >> krajMin;
    casovi[i] = napraviCas(pocSat, pocMin, krajSat, krajMin);
  }
```

```
  sort(begin(casovi), end(casovi),
       [](const cas& a, const cas& b) {
     return krajCasa(a) < krajCasa(b);
       });
  int brojOdrzanihCasova = 1;
  int kraj = krajCasa(casovi[0]);
  for (int i = 1; i < n; i++)
    if (pocetakCasa(casovi[i]) >= kraj) {
      kraj = krajCasa(casovi[i]);
      brojOdrzanihCasova++;
    }
  cout << brojOdrzanihCasova << endl;

  return 0;
}
```

Задатак: Максимални збир апсолутних разлика

Поставка: Јовица зарађује џепарац тако што доноси пакете својим комшијама. Поделу креће од своје куће и потребно је да пакете разнесе у друге куће распоређене дуж улице и да се врати назад у своју кућу. За сваку кућу познато је растојање од почетка улице. Најкраћи пут би прешао ако би пакете сложио тако да их редом дели комшијама дуж улице. Пошто Јовица жели да буде у доброј физичкој форми, он током поделе пакета трчи и жели да пакете уреди тако да пређе што већи пут. Напиши програм који одређује највећи пут који може да пређе.

Улаз: Са стандардног улаза се уноси број кућа у које треба донети пакете (међу њима се налази и Јовицина кућа), а затим и растојања тих кућа од почетка улице.

Излаз: На стандардни излаз исписати највеће растојање које Јовица може прећи током поделе пакета.

Пример 1

Улаз	*Излаз*
5 7 3 6 10 2	24

Објашњење

Постоји више начина да Јовица претрчи 24 дужне јединице. На пример, ако је његова кућа на позицији 3, он може да редом обилази куће 3, 7, 2, 10, 6, 3.

Пример 2

Улаз

```
7
3 5 11 4 2 17 9
```

Излаз

```
56
```

Решење: Решење грубом силом подразумева да се провере сви могући редоследи обиласка n кућа, тј. свих $n!$ пермутација датих бројева и да се утврди која од њих даје највећу могућу вредност пређеног пута. Овај алгоритам је изразито неефикасан и може се применити само на веома, веома мале улазе (практично, само за $n \leq 10$ програм може да реши задатак у датом временском ограничењу).

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

using namespace std;

int zbirApsRazlika(const vector<int>& a) {
  int zbir = 0;
  for (size_t k = 1; k < a.size(); k++)
    zbir += abs(a[k] - a[k-1]);
  zbir += abs(a[a.size()-1] - a[0]);
  return zbir;
}

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

  int maks = 0;
  do {
    maks = max(zbirApsRazlika(a), maks);
  } while(next_permutation(begin(a), end(a)));

  cout << maks << endl;
  return 0;
}
```

Интуитивно нам је јасно да ће се пуно претрчати ако се стално трчи са једног на други крај улице. Један грамзиви приступ је да се прво обиђе крајња лева кућа, па затим крајња десна, па друга слева, па друга здесна и тако "цик-цак". Докажимо да је ово грамзиво решење исправно.

За дати распоред $x_0, \ldots x_{n-1}$ одређујемо суму апсолутних вредности разлика елемената тј. покушавамо да максимизујемо суму

$$|x_0 - x_1| + |x_1 - x_2| + ... + |x_{n-2} - x_{n-1}| + |x_{n-1} - x_0|.$$

Сваки елемент се у суми јавља тачно два пута. У зависности од међусобног односа бројева неки елементи ће бити узети са знаком $+$, а неки са знаком $-$, и то тако да се тачно елемената узима са знаком $+$ и тачно елемената узима са знаком $-$. Циљ

нам је да елементи који се узимају са знаком $+$ буду што већи, а да ови са знаком $-$ буду што мањи. Распоред који иде цик-цак постиже да се за знаком $+$ узме $\frac{n}{2}$ већих елемената низа, а са знаком $-$ узме $\frac{n}{2}$ мањих елемената низа (ако их је непаран број, тада се средњи узима једном са знаком $-$, а једном са знаком $+$). Заиста, ако имамо 6 елемената $a_0 \leq a_1 \leq a_2 \leq a_3 \leq a_4 \leq a_5$, цик-цак распоред даје вредност

$$|a_0 - a_5| + |a_5 - a_1| + |a_1 - a_4| + |a_4 - a_2| + |a_2 - a_3| + |a_3 - a_0|,$$

што је једнако

$$(a_5 - a_0) + (a_5 - a_1) + (a_4 - a_1) + (a_4 - a_2) + (a_3 - a_2) + (a_3 - a_0),$$

тј.

$$2 \cdot (a_5 + a_4 + a_3) - 2 \cdot (a_2 + a_1 + a_0)$$

Јасно је да се не може добити више од овога, а ово се, видели смо, увек може експлицитно достићи баш цик-цак распоредом.

Елементе низа можемо експлицитно сортирати елементе низа, па затим распоредити елементе у нови низ по цик-цак редоследу у нови низ и за тај нови низ израчунати збир апсолутних вредности разлика суседних елемената.

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

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  sort(begin(a), end(a));
  vector<int> b(n);
  int i = 0, j = n-1;
  for (int k = 0; k < n; k++)
    if (k % 2 != 0)
      b[k] = a[i++];
    else
      b[k] = a[j--];

  int zbir = 0;
  for (int k = 1; k < n; k++)
    zbir += abs(b[k] - b[k-1]);
  zbir += abs(b[n-1] - b[0]);

  cout << zbir << endl;
```

```
  return 0;
}
```

Помоћни низ се може веома једноставно избећи у имплементацији.

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

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  sort(begin(a), end(a));
  int i = 0, j = n-1;
  int zbir = 0;
  bool paran = true;
  while (i < j) {
    zbir += abs(a[j]-a[i]);
    if (paran)
      i++;
    else
      j--;
    paran = !paran;
  }
  zbir += abs(a[0] - a[i]);
  cout << zbir << endl;
  return 0;
}
```

Ако пажљиво размотримо доказ коректности, примећујемо да распоред у коме идемо од прве до последње, па до друге, затим претпоследње куће итд., није једини који даје максимални пређени пут. Довољно је само да наизменично узимамо елементе из прве и друге половине низа (у било ком редоследу). Ово инспирише још једноставнији алгоритам за израчунавање траженог максимума (саберемо $\frac{n}{2}$ бројева са почетка, одузмемо $\frac{n}{2}$ бројева са краја низа и помножимо са 2).

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

using namespace std;

int main() {
  int n;
```

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

  int zbir = 0;
  for (int i = 0; i < n/2; i++) {
    zbir += a[n-1-i];
    zbir -= a[i];
  }

  cout << zbir * 2 << endl;
}
```

Глава 7

Динамичко програмирање

Задатак: Број партиција

Поставка: Партиција позитивног природног броја n је растављање броја n на збир неколико позитивних природних бројева при чему је редослед сабирака небитан (стога можемо претпоставити да је тај редослед или увек нерастући или увек неопадајући). На пример, ако је редослед нерастући, партиције броја 4 су $1+1+1+1$, $2+1+1$, $2+2$, $3+1$, 4. Написати програм који одређује број партиција за дати природан број n.

Улаз: Прва и једина линија стандардног улаза садржи природан број n ($n \leq 100$).

Излаз: На стандардном излазу приказати у првој линији број партиција природног броја n.

Пример 1

Улаз	*Излаз*
`6`	`11`

Објашњење

Ако су партиције са неопадајуће сортираним сабирцима, то су партиције:

```
1+1+1+1+1+1
1+1+1+1+2
1+1+1+3
1+1+2+2
1+1+4
1+2+3
1+5
2+2+2
2+4
3+3
6
```

Пример 2

Улаз

```
100
```

Излаз

```
190569292
```

Решење: Рекурзивне процедуре које набрајају све партиције, које су описане у задатку Све партиције се могу прилагодити тако да израчунају број партиција.

Свака партиција има свој први сабирак. Свакој партицији броја n којој је први сабирак s (при чему је $1 \leq s \leq n$) једнозначно одговара нека партиција броја $n - s$. Наметнућемо услов да су сабирци у свакој партицији сортирани нерастуће. Зато, ако је први сабирак s, сви сабирци иза њега морају да буду мањи или једнаки од s. Зато нам није довољно само да умемо да пребројимо све партиције броја $n - s$, већ је потребно да ојачамо индуктивну хипотезу. Означимо са $p_{n,s_{max}}$ број партиција броја n у којима су сви сабирци мањи или једнаки од s_{max}.

- Базу индукције чини случај $n = 0$, јер број нула има само једну партицију која не садржи сабирке. Дакле, важи да је $p_{0,s_{max}} = 1$. Ако је n веће од нула и $s_{max} = 0$, тада не постоји ни једна партиција, јер од сабирака који су сви једнаки нули (јер сви морају да буду мањи или једнаки s_{max}) не можемо никако направити неки позитиван број. Дакле, за $n > 0$ важи да је $p_{n,0} = 0$.
- Индуктивни корак можемо остварити на више начина. Најједноставнији је следећи. Приликом израчунавања $p_{n,s_{max}}$ можемо разматрати два случаја: да се у збиру не јавља сабирак s_{max} или да се у збиру јавља сабирак s_{max}. Ако се у збиру не јавља сабирак s_{max}, тада је највећи сабирак $s_{max} - 1$ и број таквих партиција је $p_{n,s_{max}-1}$. Други случај је могућ само када је $n \geq s_{max}$ и број таквих партиција је $p_{n-s_{max},s_{max}}$. На основу овога, добијамо наредну рекурзивну функцију.

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

// particije broja n u kojima je najveci sabirak jednak smax
int brojParticija(int n, int smax) {
  if (n == 0) return 1;
  if (smax == 0) return 0;
  int broj = brojParticija(n, smax - 1);
  if (n >= smax)
    broj += brojParticija(n - smax, smax);
  return broj;
}

int brojParticija(int n) {
  // particije broja n u kojima je najveći sabirak jednak n
  return brojParticija(n, n);
}
```

```
int main() {
  int n;
  cin >> n;
  cout << brojParticija(n) << endl;
}
```

Рекурзивну функцију за израчунавање броја партиција можемо добити и тако што на текуће место у партицији постављамо све могуће кандидате за проширивање текуће партиције (то су сви бројеви s од 1 до мањег од бројева s_{max} и n) и настављамо генерисање партиција броја $n - s$ код којих је највећи сабирак једнак s.

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


// particije broja n u kojima je najveći sabirak jednak smax
int brojParticija(int n, int smax) {
   if (n == 0) return 1;
   if (smax == 0) return 0;
   int broj = 0;
   for (int s = min(smax, n); s >= 1; s--)
     broj += brojParticija(n - s, s);
   return broj;
}

int brojParticija(int n) {
  // particije broja n u kojima je najveći sabirak jednak n
  return brojParticija(n, n);
}

int main() {
  int n;
  cin >> n;
  cout << brojParticija(n) << endl;
}
```

Ако уместо услова да су сабирци уређени нерастуће ставимо услов да су сабирци уређени неопадајуће, тада уз број чије партиције тражимо шаљемо и број s_{min} који представља доњу границу наредних сабирака у партицији (сви сабирци морају да имају вредност бар s_{min}).

- Базу чини случај када је $n = 0$, јер број нула има само једну партицију која не садржи сабирке. Такође, када је $s_{min} > n$, тада је број партиција нула, јер ниједна партиција не може да има сабирак већи од збира.
- Број партиција броја n које садрже s_{min} једнак је броју партиција вредности $n - s_{min}$ са минималним сабирком s_{min}, док је број партиција које не садрже s_{min} једнак броју партиција броја n у којима је најмањи сабирак $s_{min} + 1$. Дакле,

$$p_{n,s_{min}} = p_{n-s_{min},s_{min}} + p_{n,s_{min}+1}.$$

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

// particije broja n u kojima je najveci sabirak jednak smax
int brojParticija(int n, int smin) {
  if (n == 0) return 1;
  if (smin > n) return 0;
  return  brojParticija(n, smin + 1) +
    brojParticija(n - smin, smin);
}

int brojParticija(int n) {
  // particije broja n u kojima je najmanji sabirak jednak 1
  return brojParticija(n, 1);
}

int main() {
  int n;
  cin >> n;
  cout << brojParticija(n) << endl;
}
```

Можемо формулисати још једно решење у коме се броје неопадајуће сортиране партиције у којем у петљи разматрамо све могућности за вредност наредног сабирка. Размотримо, на пример, партиције броја 6.

```
1 1 1 1 1 1
1 1 1 1 2
1 1 1 3
1 1 2 2
1 1 4
1 2 3
1 5
2 2 2
2 4
3 3
6
```

Приметимо да је максимална вредност првог сабирка у свим партицијама осим у оној једночланој једнакој броју n, мања или једнака $\frac{n}{2}$ (у претходном примеру ни једна партиција не почиње ни са 4, ни са 5). Заиста, ако би први сабирак био строго већи од $\frac{n}{2}$ и ако партиција не би била једночлана и други сабирак би морао бити строго већи од $\frac{n}{2}$, па би збир био строго већи од n, што је немоуће. Стога једночлану партицију посебно урачунавамо, а рекурзивно генеришемо партиције за све могуће вредности првог сабирка од s_{min} до $\lfloor \frac{n}{2} \rfloor$.

```
#include <iostream>
```

```cpp
using namespace std;

int brojParticija(int n, int smin) {
  if (n == 0)
    return 0;
  int br = 1;
  for(int i = smin; i <= n/2; i++)
    br += brojParticija(n - i, i);
  return br;
}

int brojParticija(int n) {
  return brojParticija(n, 1);
}

int main() {
  int n;
  cin >> n;
  cout << brojParticija(n) << endl;
  return 0;
}
```

Сва решења заснована на простој рекурзији су неефикасна, јер долази до понављања идентичних рекурзивних позива и могу се поправити динамичким програмирањем. Прикажимо то на примеру бројања нерастуће уређених пермутација у којима вршимо два рекурзивна позива.

Кренимо са мемоизацијом. Уводимо матрицу димензије $(n+1) \times (n+1)$ коју попуњавамо вредностима -1, чиме означавамо да резултат позива функције још није познат. Пре него што кренемо са израчунавањем проверавамо да ли је у матрици вредност различита од -1 и ако јесте, враћамо ту упамћену вредност. Пре сваке повратне вредности функције резултат памтимо у матрици.

[недостаје] broj_particija-ex3.cpp

Уместо мемоизације можемо употребити и динамичко програмирање навише. Прикажимо табелу вредности функције за $n = 7$.

n\smax	0	1	2	3	4	5	6	7
0	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1
2	0	1	2	2	2	2	2	2
3	0	1	2	3	3	3	3	3
4	0	1	3	4	5	5	5	5
5	0	1	3	5	6	7	7	7
6	0	1	4	7	9	10	11	11
7	0	1	4	8	11	13	14	15

На основу базе индукције знамо да ће сви елементи прве врсте бити једнаки 1, а да ће у првој колони сви елементи осим почетног бити једнаки 0. Један од начина да се

матрица попуњава је постепено увећавајући вредност n, тј. попуњавајући врсту по врсту.

Елемент $p_{n,s}$ зависи од елемената $p_{n,s-1}$ и (ако је $n \geq s$) $p_{n-s,s}$ и ако се врсте попуњавају од горе наниже и слева надесно, приликом његовог израчунавања оба елемента од којих зависи су већ израчуната, што даје коректан алгоритам.

```
#include <iostream>
#include <vector>

using namespace std;

int brojParticija(int N) {
  // alociramo matricu
  vector<vector<int>> dp(N+1);
  for (int n = 0; n <= N; n++)
    dp[n].resize(N+1);
  // popunjavamo prvu vrstu
  for (int smax = 0; smax <= N; smax++)
    dp[0][smax] = 1;
  // popunjavamo preostale elemente prve kolone
  for (int n = 1; n <= N; n++)
    dp[n][0] = 0;
  // popunjavamo jednu po jednu vrstu
  for (int n = 1; n <= N; n++)
    for (int smax = 1; smax <= N; smax++) {
      dp[n][smax] = dp[n][smax-1];
      if (n >= smax)
        dp[n][smax] += dp[n-smax][smax];
    }
  return dp[N][N];
}


int main() {
  int n;
  cin >> n;
  cout << brojParticija(n) << endl;
  return 0;
}
```

И временска и меморијска сложеност претходног алгоритма је $O(n^2)$ иако се матрица попуњава врсту по врсту, то није могуће поправити (јер елементи зависе од елемената који се јављају не само у претходној, већ и у ранијим врстама, тако да је потребно да истовремено чувамо све претходне врсте). Међутим, ако матрицу попуњавамо колону по колону одозго наниже, можемо добити меморијску сложеност $O(n)$ – временска сложеност остаје $O(n^2)$. Наиме, сваки елемент зависи од елемента у истој врсти у претходној колони и елемента у истој колони у некој од претходних врста, тако да ако колоне попуњавамо одозго наниже, можемо чувати само две узастопне колоне.

Заправо, можемо чувати и само једну колону, ако њено попуњавање организујемо тако да се током ажурирања сви елементи пре текуће врсте односе на вредности текуће колоне, а од текуће врсте до краја односе на вредности претходне колоне. Приметимо да се у делу где је $n < s_{max}$, вредности између две суседне колоне не мењају. Тиме добијамо наредну оптимизовану имплементацију.

```
#include <iostream>
#include <vector>

using namespace std;

int brojParticija(int N) {
  vector<int> dp(N+1, 0);
  dp[0] = 1;
  for (int smax = 1; smax <= N; smax++)
    for (int n = smax; n <= N; n++)
      dp[n] += dp[n-smax];
  return dp[N];
}

int main() {
  int n;
  cin >> n;
  cout << brojParticija(n) << endl;
  return 0;
}
```

Динамичким програмирањем можемо оптимизовати и случај када се броје партиције са неопадајуће сортираним сабирцима. Прикажимо табелу вредности функције за $n = 8$.

n\smin	1	2	3	4	5	6	7	8
1	1	0	0	0	0	0	0	0
2	2	1	0	0	0	0	0	0
3	3	1	1	0	0	0	0	0
4	5	2	1	1	0	0	0	0
5	7	2	1	1	1	0	0	0
6	11	4	2	1	1	1	0	0
7	15	4	2	1	1	1	1	0
8	22	7	3	2	1	1	1	1

Табелу можемо попуњавати врсту по врсту. Нажалост, меморијску оптитмизацију је овде теже остварити.

```
#include <iostream>

using namespace std;

const int MAX = 100;
```

```
int brojParticija(int N) {
  int br[MAX + 1][MAX + 1] = {0};

  // br[i][j] je broj particija broja i pomocu sabiraka >= smin
  for(int n = 1; n <= N; n++) {
    br[n][n] = 1;
    for(int smin = n-1; smin > 0; smin--)
      br[n][smin] = br[n][smin+1] + br[n-smin][smin];
  }
  return br[N][1];
}

int main() {
  int n;
  cin >> n;
  cout << brojParticija(n) << endl;
  return 0;
}
```

Задатак: Најдужи растући подниз

Поставка: Напиши програм који одређује најдужи строго растући подниз (не обавезно узастопних елемената) унутар датог низа.

Улаз: Са стандардног улаза се учитава број елемената низа n ($1 \leq n \leq 50000$), а затим елементи низа (цели бројеви, сваки у посебном реду).

Излаз: На стандардни излаз исписати дужину најдужег растућег подниза.

Пример

Улаз	*Излаз*
10 3 2 6 9 5 4 3 7 2 8	4

Објашњење

Један растући подниз дужине 4 је `2 6 7 8`.

Решење: Приказаћемо два решења заснована на динамичком програмирању, која су различите ефикасности.

У првој групи решења разматраћемо позицију по позицију у низу и одредићемо најдужи растући подниз чији је последњи елемент на свакој од њих. Приликом одређивања дужине најдужег растућег подниза који се завршава на позицији $i \geq 0$, претпоставићемо да за сваку претходну позицију (ако их има) умемо да одредимо дужину најдужег растућег подниза који се на њој завршава. Низ који се завршава на позицији i сигурно садржи елемент a_i, а може продужити све оне низове који се завршавају на некој позицији $0 \leq j < i$ ако је $a_j < a_i$. Да би низ који се завршава на позицији j био што дужи, његов префикс који се завршава на позицији j мора бити што дужи (а дужине тих низова можемо одредити рекурзивно). Зато од свих низова који се завршавају на позицијама j таквим да је $a_j < a_i$ одређујемо најдужи и продужавамо га елементом a_i (ако таквих елемената нема, тада је најдужи низ који се завршава на позицији a_i једночлан).

```
#include <iostream>
#include <vector>

using namespace std;

// pronalazi duzinu najduzeg rastuceg podniza niza a koji se zavrsava
// elementom na poziciji i
int najduziRastuciPodniz(const vector<int>& a, int i) {
  int maksI = 1;
  for (int j = 0; j < i; j++)
    if (a[j] < a[i]) {
      int maksJ = najduziRastuciPodniz(a, j);
      if (maksJ + 1 > maksI)
        maksI = maksJ + 1;
    }
  return maksI;
}

int najduziRastuciPodniz(const vector<int>& a) {
  int maks = 0;
  for (int i = 0; i < a.size(); i++) {
    int maksI = najduziRastuciPodniz(a, i);
    if (maksI > maks)
      maks = maksI;
  }
  return maks;
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
```

```
  cout << najduziRastuciPodniz(a) << endl;
  return 0;
}
```

Када се претходна рекурзивна веза директно имплементира, долази до понављања идентичних рекурзивних позива, што доводи до велике неефикасности. Проблем решавамо динамичким програмирањем. За мемоизацију довољно је да памтимо низ дужина најдужих растућих низова који се завршавају на свакој позицији у низу. Пошто су све те дужине веће или једнаке од 1 (сваки елемент сам за себе чини растући низ), низ у који меморишемо решења можемо иницијализовати нулама (што значи да је тражена дужина још непозната).

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

// pronalazi duzinu maksP najduzeg rastuceg podniza niza a koji se
// zavrsava elementom na poziciji p, kao i duzinu maks najduzeg
// rastuceg podniza unutar prefiksa koji se zavrsava na poziciji p
int najduziRastuciPodniz(const vector<int>& a, int p,
                         vector<int>& memo) {
  if (memo[p] != 0)
    return memo[p];
  int maksP = 1;
  for (int i = 0; i < p; i++) {
    if (a[i] < a[p]) {
      int maksI = najduziRastuciPodniz(a, i, memo);
      if (maksI + 1 > maksP)
        maksP = maksI + 1;
    }
  }
  return memo[p] = maksP;
}

int najduziRastuciPodniz(const vector<int>& a) {
  vector<int> memo(a.size(), 0);
  int maks = 0;
  for (int i = 0; i < a.size(); i++) {
    int maksI = najduziRastuciPodniz(a, i, memo);
    if (maksI > maks)
      maks = maksI;
  }
  return maks;
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
```

```
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  cout << najduziRastuciPodniz(a) << endl;
  return 0;
}
```

Једноставно можемо формулисати и динамичко програмирање навише, тако што низ попуњавамо слева надесно. Након попуњавања низа одређујемо његов максимум. Нагласимо да сваки наредни елемент низа потенцијално зависи од великог броја претходних тако да није могуће редуковати меморијску сложеност тиме што би се памтили само неки елементи низа (морамо увек знати дужине свих претходних елемената). Решење које се добија на овај начин је меморијске сложености $O(n)$ и временске сложености $O(n^2)$.

Прикажимо на примеру како ће се тај низ попуњавати.

```
i  0 1 2 3 4 5 6 7 8 9
ai 3 2 6 9 5 4 3 7 8 2
dp 1 1 2 3 2 2 2 3 4 1
```

На пример, када израчунавамо елемент на позицији 5 анализирамо низове који се завршавају елементима мањим од вредности 4 која се налази на позицији 5. То су вредности 3 на позицији 0 и 2 на позицији 1. У оба случаја максимална дужина подниза који се завршава на тој позицији је 1, па се било који од тих низова продужава елементом 4 и добија се растући низ дужине 2.

```
#include <iostream>
#include <vector>

using namespace std;

int najduzi_rastuci_podniz(const vector<int>& a) {
  int n = a.size();
  vector<int> dp(n);
  for (int i = 0; i < n; i++) {
    dp[i] = 1;
    for (int j = 0; j < i; j++)
      if (a[i] > a[j] && dp[j] + 1 > dp[i])
    dp[i] = dp[j] + 1;
  }

  int max = dp[0];
  for (int i = 0; i < n; i++)
    if (dp[i] > max)
      max = dp[i];
  return max;
}
```

```
int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  cout << najduzi_rastuci_podniz(a) << endl;

  return 0;
}
```

Претходно решење је сложености $O(n^2)$, али изменом индуктивно-рекурзивне конструкције можемо добити и много ефикасније решење. Кључна идеја је да претпоставимо да уз дужину d_{max} најдужег растућег подниза до сада обрађеног дела низа можемо за сваку дужину подниза $1 \leq d \leq d_{max}$ да одредимо најмањи елемент којим се завршава неки растући подниз дужине d. Приметимо да низ тих вредности увек строго растући (ако постоји строго растући низ дужине d који се завршава неким елементом a_i, тада се његов префикс дужине $d-1$ мора завршавати неким елементом који је строго мањи од елемента a_i).

Низ обрађујемо елемент по елемент. Размотримо један пример (колоне табеле су означене дужинама низа), а елементи низа који се обрађују су написани десно.

```
1 2 3 4 5 6 7 8 9 10
  - - - - - - - - - -
3 - - - - - - - - -         3
2 - - - - - - - - -         2
2 6 - - - - - - - -         6
2 6 9 - - - - - - -         9
2 5 9 - - - - - - -         5
2 4 9 - - - - - - -         4
2 3 9 - - - - - - -         3
2 3 7 - - - - - - -         7
2 3 7 - - - - - - -         2
2 3 7 8 - - - - - -         8
```

Ако се досадашњи најдужи подниз завршавао елементом који је мањи од текућег, онда смо нашли подниз који је дужи за један и најмањи елемент на крају тог подниза је текући. То се у примеру дешава приликом обраде елемента `3`, елемента `6`, елемента `9` и елемента `8`.

Размотримо ситуацију у којој обрађујемо елемент `5`. До тада смо видели елементе `3`, `2`, `6` и `9`. Елемент `2` на првој позицији у табели означава да је најмањи елемент којим се може завршити једночлани растући низ једнак `2`. Елемент `6` на другој позицији у табели означава да је најмањи елемент којим се може завршити двочлани растући низ једнак `6` (у питању је низ `2 6` или низ `3 6`). Елемент `9` на трећој позицији у табели означава да је најмањи елемент којим се може завршити трочлани растући низ једнак `9` (у питању је низ `2 6 9` или `3 6 9`). Пошто је `5` мањи од `9` ниједан од ових

трочланих низова није могуће проширити елементом 5, па је четворочланих растућих низова нема. Поставља се питање да ли се можда трочлани низови могу завршити елементом 5, но ни то није могуће. Наиме, пошто је у табели двочланим низовима придружена вредност 6, то значи да се сви двочлани растући низови завршавају бар са 6, па није могуће 9 заменити са 5. Са друге стране, пошто је 5 веће од 2, завршни елемент двочланих низова 6 је могуће заменити са 5 и тиме добити мању завршну вредност двочланих низова (то су у овом случају низови 3 5 и 2 5). Дакле, у табели вредност 6 треба заменити вредношћу 5. Вредност 2 лево од 6 нема смисла заменити са 5, јер би се тиме завршна вредност једночланих низова увећала, а ми у табели памтимо најмање завршне вредности.

На основу анализе овог примера можемо да закључимо да је приликом анализе сваког текућег елемента потребно пронаћи прву позицију d у табели на којој се налази елемент који је већи или једнак од текућег и позицију d уместо тога уписати текући елемент. Ако су сви елементи мањи од текућег (ако је $d = d_{max}$), онда се текући елемент додаје на крај низа (и у том случају заправо радимо исто - уписујемо елемент на позицију d). Остали елементи у табели остају непромењени. Заиста на свим позицијама у табели лево од позиције d уписани су елементи строго мањи од текућег и њиховом заменом са текућим се не би смањила вредност завршног елемената тих низова. За елементе десно од позиције d, иако су већи од текућег, ажурирање није могуће. У свим низовима дужине $d' > d$ неки префикс се морао завршавати елементом на позицији d или елементом већим од њега, а пошто је он био већи или једнак од текућег, заменог последњег елемента текућим не бисмо добили више растући низ.

Кључни добитак настаје када се примети да, пошто су елементи у табели сортирани, позицију првог елемента који је већи или једнак од текућег можемо остварити бинарном претрагом. Отуда следи ефикасна имплементација (у низу dp вредност најмањег завршног елемента за низове дужине d памтимо на позицији $d-1$). Временска сложеност такве имплементације је $O(n \cdot \log(n))$, док је меморијска сложеност $O(n)$.

Бинарна претрага може бити извршена библиотечком функцијом.

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

using namespace std;

int najduziRastuciPodniz(const vector<int>& a) {
  int n = a.size();
  vector<int> dp(n);
  int max = 0;
  for (int i = 0; i < n; i++) {
    auto it = lower_bound(dp.begin(), next(dp.begin(), max), a[i]);
    *it = a[i];
    int d = distance(dp.begin(), it);
    if (d + 1 > max)
      max = d + 1;
  }
```

```
  return max;
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n, 0);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  cout << najduziRastuciPodniz(a) << endl;

  return 0;
}
```

Још једна могућност је да се бинарна претрага ручно имплементира (исто као у задатку Први већи и последњи мањи).

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

using namespace std;

int prviVeciIliJednak(const vector<int>& a, int l, int d, int x) {
  while (l < d) {
    int s = l + (d - l) / 2;
    if (a[s] < x)
      l = s + 1;
    else
      d = s;
  }
  return l;
}

int  najduziRastuciPodniz(const vector<int>& a) {
  int n = a.size();
  vector<int> dp(n+1);
  int max = 0;
  for (int i = 0; i < n; i++) {
    int k = prviVeciIliJednak(dp, 0, max, a[i]);
    dp[k] = a[i];
    if (k + 1 > max)
      max = k + 1;
  }
  return max;
}
```

```
int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  cout << najduziRastuciPodniz(a) << endl;

  return 0;
}
```

Постоји веома једноставно свођење овог проблема на проблем проналажења најдужег заједничког подниза два низа, чије смо решење већ приказали у задатку Најдужи заједнички подниз две ниске. Наиме, дужина најдужег растућег подниза датог низа једнака је дужини најдужег заједничког подниза тог низа и низа који се добија неопадајућим сортирањем и уклањањем дупликата тог низа.

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

int najduziZajednickiPodniz(const vector<int>& s1, const vector<int>& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1, 0);
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1]) {
        dp[j] = prethodni + 1;
      } else {
        dp[j] = max(dp[j-1], dp[j]);
      }
      prethodni = tekuci;
    }
  }
  return dp[n2];
}

int najduziRastuciPodniz(const vector<int>& a) {
  // sortirana kopija niza a
  vector<int> b = a;
  sort(begin(b), end(b));
  // uklanjamo duplikate iz vektora b
```

```
  b.erase(unique(begin(b), end(b)), end(b));
  return najduziZajednickiPodniz(a, b);
}

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  cout << najduziRastuciPodniz(a) << endl;
  return 0;
}
```

Задатак: Најдужи заједнички подниз две ниске

Поставка: Напиши програм који израчунава дужину највећег заједничког подниза две ниске. Подниз чине карактери ниске који не морају бити узастопни, али се јављају у истом редоследу као у оригиналној ниски. На пример за ниске `abacbc` и `babbca` најдужа заједничка подниска је `babc`.

Улаз: Две линије стандардног улаза садрже две ниске које се састоје од малих слова енглеске азбуке и дугачке су највише 1000 карактера.

Излаз: На стандардни излаз исписати само тражену дужину.

Пример

Улаз	*Излаз*
`xmjyauz` `mzjawxu`	`4`

Решење: Ако је било која од две ниске празна, тада је једини њен подниз празан, па је дужина најдужег заједничког подниза једнака нули. Ако су обе ниске непразне, тада можемо упоредити њихова последња слова. Ако су она једнака, могу бити укључена у најдужи заједнички подниз и проблем се рекурзивно своди на проналажење најдужег заједничког подниза њихових префикса. У супротном, није могуће да оба последња слова буду укључена у заједнички подниз. Зато разматрамо најдужи заједнички подниз прве ниске и префикса друге ниске без њеног последњег слова и заједнички подниз друге ниске и префикса прве ниске без њеног последњег слова. Дужи од два подниза биће најдужи заједнички подниз те две ниске. Нагласимо и да није неопходно разматрати најдужи заједнички подниз два префикса, јер се проширивањем неког од два префикса за последње слово не може добити подниз који би био краћи. Дакле, пошто рекурзија тече по префиксима ниски, једини променљиви параметри током рекурзије могу бити дужине тих префикса.

Ако са $f(m, n)$ означимо дужину најдужег заједничког подниза префикса ниске a дужине m и префикса ниске b дужине n, тада важи следећа рекурентна веза:

$$
\begin{aligned}
f(0,n) &= 0 \\
f(m,0) &= 0 \\
f(m,n) &= f(m-1,n-1)+1, \quad \text{za } a_{m-1} = b_{n-1} \\
f(m,n) &= \max\left(f(m,n-1), f(m-1,n)\right), \quad \text{za } a_{m-1} \neq b_{n-1}
\end{aligned}
$$

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

int najduziZajednickiPodniz(const string& s1, int n1,
                            const string& s2, int n2) {
    if (n1 == 0 || n2 == 0)
       return 0;
    int rez = max(najduziZajednickiPodniz(s1, n1, s2, n2-1),
                  najduziZajednickiPodniz(s1, n1-1, s2, n2));
    if (s1[n1-1] == s2[n2-1])
       rez = max(rez, najduziZajednickiPodniz(s1, n1-1, s2, n2-1) + 1);
    return rez;
}

int najduziZajednickiPodniz(const string& s1, const string& s2) {
  return najduziZajednickiPodniz(s1, s1.size(), s2, s2.size());
}

int main() {
  string s1, s2;
  cin >> s1 >> s2;
  cout << najduziZajednickiPodniz(s1, s2) << endl;
  return 0;
}
```

У директном рекурзивном решењу има много преклапајућих рекурзивних позива. Стога је ефикасност могуће поправити техником динамичког програмирања. Један могући приступ је да употребимо мемоизацију. Вредност дужине најдужег подниза за сваки пар дужина префикса можемо памтити у матрици.

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int najduziZajednickiPodniz(const string& s1, int n1,
                            const string& s2, int n2,
                            vector<vector<int>>& memo) {
```

```
  if (memo[n1][n2] != -1)
    return memo[n1][n2];

  if (n1 == 0 || n2 == 0)
    return memo[n1][n2] = 0;
  int rez = max(najduziZajednickiPodniz(s1, n1, s2, n2-1, memo),
                najduziZajednickiPodniz(s1, n1-1, s2, n2, memo));
  if (s1[n1-1] == s2[n2-1])
    rez = max(rez, najduziZajednickiPodniz(s1, n1-1, s2, n2-1, memo) + 1);
  return memo[n1][n2] = rez;
}

int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> memo(n1+1);
  for (int i = 0; i <= n1; i++)
    memo[i].resize(n2 + 1, -1);

  return najduziZajednickiPodniz(s1, n1, s2, n2, memo);
}

int main() {
  string s1, s2;
  cin >> s1 >> s2;
  cout << najduziZajednickiPodniz(s1, s2) << endl;
  return 0;
}
```

Проблем прекалпајућих рекурзивних позива се може решити ако се употреби динамичко програмирање навише. Дужине најдужих поднизова префикса можемо чувати у матрици. Елемент матрице на позицији (m, n) зависи само од елемената на позицијама $(m-1, n)$, $(m, n-1)$ и $(m-1, n-1)$, тако да матрицу пожемо да попуњавамо било врсту по врсту, било колону по колону. Прикажимо матрицу за пример две ниске.

```
xmjyauz
mzjawxu

     mzjawxu
    01234567
   ---------
  0|00000000
x 1|00000011
m 2|01111111
j 3|01122222
y 4|01122222
a 5|01123333
u 6|01123334
z 7|01223334
```

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> dp(n1+1);
  for (int i = 0; i <= n1; i++)
    dp[i].resize(n2 + 1);

  for (int i = 0; i <= n1; i++)
    dp[i][0] = 0;
  for (int j = 0; j <= n2; j++)
    dp[0][j] = 0;

  for (int i = 1; i <= n1; i++)
    for (int j = 1; j <= n2; j++) {
      dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
      if (s1[i-1] == s2[j-1])
        dp[i][j] = max(dp[i][j], dp[i-1][j-1] + 1);
    }

  return dp[n1][n2];
}

int main() {
  string s1, s2;
  cin >> s1 >> s2;
  cout << najduziZajednickiPodniz(s1, s2) << endl;
  return 0;
}
```

Можемо приметити да се приликом попуњавања матрице врсту по врсту садржај свакe наредне врсте попуњава само на основу претходне врсте. Стога није потребно истовремено памтити целу матрицу, већ је довољно памтити само једну, текућу врсту. Ажурирање врсте морамо вршити с лева надесно, јер сваки елемент у текућој врсти зависи од елемента који му претходи у тој врсти. Приметимо да нам је у неком тренутку потребно да знамо претходни елемент текуће врсте, а понекад претходни елемент претходне врсте, тако да приликом ажурирања врсте морамо да у помоћној променљивој памтимо стару вредност претходног елемента врста (јер се ажурирањем претходног елемента његова стара вредност губи, а она нам може затребати у случају да су одговарајући карактери у нискама једнаки).

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
```

```
using namespace std;

int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1, 0);
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1])
        dp[j] = prethodni + 1;
      else
        dp[j] = max(dp[j-1], dp[j]);
      prethodni = tekuci;
    }
  }
  return dp[n2];
}

int main() {
  string s1, s2;
  cin >> s1 >> s2;
  cout << najduziZajednickiPodniz(s1, s2) << endl;
  return 0;
}
```

Задатак: Најдужи подниз палиндром

Поставка: Написати програм којим се за дати стрингу s одређује дужину најдужег подниза ниске s који је палиндром. Подниз не мора да садржи узастопне карактере ниске, али они морају да се јављају у истом редоследу (подниз се добија брисањем произвољног броја карактера).

Улаз: Са стандардног улаза се учитава ниска s састављена само од малих слова енглеске абецеде, чија је дужина највише 5000 карактера.

Излаз: На стандардни излаз исписати само тражену дужину најдужег палиндромског подниза.

Пример

Улаз	*Излаз*
`najduzipalindrom`	`5`

Решење: Кренимо од рекурзивног решења.

- Празна ниска има само празан подниз, па је дужина најдужег палиндромског подниза једнака нули. Ниска дужине 1 је сама свој палиндромски подниз, па је дужина њеног најдужег палиндромског подниза једнака 1.

- Ако ниска има бар два карактера, онда рамзатрамо да ли су њен први и последњи карактер једнаки. Ако јесу, онда они могу бити део најдужег палиндромског подниза и проблем се своди на проналажење најдужег палиндромског подниза дела ниске без првог и последњег карактера. У супротном они не могу истовремено бити део најдужег палиндромског подниза и потребно је елиминисати бар један од њих. Проблем, дакле, сводимо на проналажење најдужег палиндромског подниза суфикса ниске без првог карактера и на проналажење најдужег палиндромског подниза префикса ниске без последњег карактера. Дужи од та два палиндромска подниза је тражени палиндромски подниз целе ниске.

Овим је практично дефинисана рекурзивна процедура којом се решава проблем. У сваком рекурзивном позиву врши се анализа неког сегмента (низа узастопних карактера полазне ниске), па је сваки рекурзивни позив одређен са два броја који представљају границе тог сегмента. Ако са $f(l, d)$ означимо дужину најдужег палиндромског подниза дела ниске $s[l, d]$, тада важе следеће рекурентне везе.

$$
\begin{aligned}
f(l,d) &= 0, \quad \text{za } l > d \\
f(l,d) &= 1, \quad \text{za } l = d \\
f(l,d) &= 2 + f(l+1, d-1), \quad \text{za } l < d \text{ i } s_l = s_d \\
f(l,d) &= \max\left(f(l+1,d), f(l,d-1)\right), \quad \text{za } l < d \text{ i } s_l \neq s_d
\end{aligned}
$$

На основу овога, функцију је веома једноставно имплементирати.

```
#include <iostream>
#include <string>

using namespace std;

int najduziPalindrom(const string& s, int l, int d) {
  if (l > d)
    return 0;
  if (l == d)
    return 1;
  if (s[l] == s[d])
    return 2 + najduziPalindrom(s, l+1, d-1);
  return max(najduziPalindrom(s, l, d-1),
             najduziPalindrom(s, l+1, d));
}

int najduziPalindrom(const string& s) {
  return najduziPalindrom(s, 0, s.length() - 1);
}

int main() {
   string s;
   cin >> s;
```

```
   cout << najduziPalindrom(s) << endl;
   return 0;
}
```

У претходној функцији долази до преклапања рекурзивних позива, па је пожељно употребити мемоизацију. За мемоизацију користимо матрицу (практично, њен горњи троугао у којем је $l < d$).

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

int najduziPalindrom(const string& s, int l, int d,
                     vector<vector<int>>& memo) {
  if (memo[l][d] != -1)
    return memo[l][d];
  if (l > d)
    return memo[l][d] = 0;
  if (l == d)
    return memo[l][d] = 1;
  if (s[l] == s[d])
    return memo[l][d] = 2 + najduziPalindrom(s, l+1, d-1, memo);
  return memo[l][d] = max(najduziPalindrom(s, l, d-1, memo),
                          najduziPalindrom(s, l+1, d, memo));
}

int najduziPalindrom(const string& s) {
  vector<vector<int>> memo(s.length(), vector<int>(s.length(), -1));
  return najduziPalindrom(s, 0, s.length() - 1, memo);
}

int main() {
   string s;
   cin >> s;
   cout << najduziPalindrom(s) << endl;
   return 0;
}
```

До ефикасног решења можемо доћи и динамичким програмирањем одоздо навише. Елемент на позицији (l, d) матрице зависи од елемената на позицијама $(l+1, d)$, $(l, d-1)$ и $(l+1, d-1)$, док се коначно решење налази у горњем левом углу матрице, тј. на пољу $(0, n-1)$. Због оваквих зависности матрицу не можемо попуњавати ни врсту по врсту, ни колону по колону, већ дијагоналу по дијагоналу. На дијагоналу испод главне уписујемо све нуле, на главну дијагоналу све јединице, а затим попуњавамо једну по једну дијагоналу изнад главне, све док не дођемо до елемента у горњем левом углу.

Прикажимо како изгледа попуњена матрица на примеру ниске `abaccba`.

```
    abaccba
    0123456
    -------
a 0|1133346
b 1|0111244
a 2| 011224
c 3|  01222
c 4|   0111
b 5|    011
a 6|     01
```

Коначно решење 6 одговара поднизу `abccba`.

Ово решење има и меморијску и временску сложеност $O(n^2)$.

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

int najduziPalindrom(const string& s) {
  int n = s.length();
  vector<vector<int>> dp(n);
  for (int i = 0; i < n; i++) {
    dp[i].resize(n, 0);
    dp[i][i] = 1;
  }
  for (int k = 1; k < n; k++)
    for (int l = 0; l + k < n; l++) {
      int d = l + k;
      if (s[l] == s[d])
        dp[l][d] = dp[l+1][d-1] + 2;
      else
        dp[l][d] = max(dp[l+1][d], dp[l][d-1]);
    }

  return dp[0][n - 1];
}

int main() {
   string s;
   cin >> s;
   cout << najduziPalindrom(s) << endl;
   return 0;
}
```

Меморијску сложеност је могуће редуковати. Примећујемо да елементи сваке дијаго-

нале зависе само од елемената претходне две дијагонале. Могуће је да чувамо само две дијагонале - текућу и претходну. Током ажурирања текуће дијагонале њене постојеће елементе истовремено преписујемо у претходну. Када су карактери једнаки, тада у привремену промењиву бележимо одговарајући елемент претходне дијагонале, на његово место уписујемо одговарајући елемент текуће дијагонале, а онда на место тог елемента уписујемо вредност привремене промењиве увећану за два. Када су карактери различити одговарајући елемент текуће дијагонале уписујемо на одговарајуће место у претходној дијагонали, а на његово место уписујемо максимум те и наредне вредности текуће дијагонале.

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

int najduziPalindrom(const string& s) {
  int n = s.length();
  // elementi dve prethodne dijagonale
  vector<int> dpp(n, 0);
  vector<int> dp(n, 1);
  for (int k = 1; k < n; k++) {
    for (int l = 0; l + k < n; l++) {
      int d = l + k;
      if (s[l] == s[d]) {
        int tmp = dp[l];
        dp[l] = dpp[l+1] + 2;
        dpp[l] = tmp;
      }
      else {
        dpp[l] = dp[l];
        dp[l] = max(dp[l], dp[l+1]);
      }
    }
    dpp[n-k] = dp[n-k];
  }

  return dp[0];
}

int main() {
   string s;
   cin >> s;
   cout << najduziPalindrom(s) << endl;
   return 0;
}
```

Рецимо још и да је решење овог задатка могуће добити и свођењем на проблем одре-

ђивања најдужег заједничког подниза две ниске који је описан у задатку Најдужи заједнички подниз две ниске. Наиме, најдужи палиндромски подниз једнак је најдужем заједничком поднизу оригиналне ниске и ниске која се добија њеним обртањем. Сложеност ове редукције је иста као и сложеност директног решења (временски $O(n^2)$, а просторно $O(n)$).

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

int najduziZajednickiPodniz(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1, 0);
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1]) {
        dp[j] = prethodni + 1;
      } else {
        dp[j] = max(dp[j-1], dp[j]);
      }
      prethodni = tekuci;
    }
  }
  return dp[n2];
}

int najduziPalindrom(const string& s) {
  string sObratno = s;
  reverse(begin(sObratno), end(sObratno));
  return najduziZajednickiPodniz(s, sObratno);
}

int main() {
   string s;
   cin >> s;
   cout << najduziPalindrom(s) << endl;
   return 0;
}
```

Глава 8

Подели па владај

Задатак: Збир k најбољих

Поставка: Ученик је радио n задатака и за сваки задатак је добио одређени број поена. Одредити збир поена на k задатака које је најбоље урадио.

Улаз: У првој линији стандардног улаза унети природан број n ($1 \leq n \leq 10^6$) - број задатака које је ученик радио, у другој природан број k ($1 \leq k \leq n$) - број задатака које је најбоље урадио, а затим у следећих n линија број поена које је добио на задацима.

Излаз: Укупан број поена које је освојио на k најбоље оцењених задатака.

Пример

Улаз	*Излаз*
10 3 15 80 25 60 10 20 50 45 40 30	190

Решење:

Сортирање целог низа

Задатак се може решити и тако што ће се низ сортирати библиотечком функцијом за сортирање (слично као у задатку Сортирање бројева). Ако се низ сортира нерастуће, онда је након сортирања потребно сабрати првих k елемената низа, а ако се сортира

неопадајуће, онда је након сортирања потребно сабрати последњих k елемената низа (сортирање неопадајуће је обично подразумевани начин сортирања и лакше га је реализовати). Ако се сортирање врши библиотечким функцијама, временска сложеност овог решења је $O(n \cdot \log(n))$, док је меморијска сложеност једнака $O(n)$.

Може се приметити да овакав алгоритам непотребно губи време прецизно одређујући редослед тј. сортирајући елементе који су мањи од првих k као и одређујући прецизан редослед првих k елемената (да би се сабрало првих k елемената они не морају бити поређани, већ је само потребно на почетак низа (или на крај) довести првих k елемената, а иза њих (или испред) поставити остале елементе тј. раздвојити те две групе елемената, при чему је редослед елемената унутар сваке од група ирелевантан.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <functional>  // zbog greater<int>() u pozivu funkcije sort
using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo ulazne podatke
  int n, k;
  cin >> n >> k;

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

  // sortiramo niz nerastuce
  sort(begin(a), end(a), greater<int>());

  // sabiramo prvih k elemenata niza
  int s = accumulate(begin(a), next(begin(a), k), 0);

  // ispisujemo rezultat
  cout << s << endl;
}
```

Модификовани алгоритам сортирања селекцијом

Једна могућа идеја која избегава сортирање елемената који су мањи од првих k је да се сортирање врши алгоритмом селекције (види задатак Сортирање бројева) који, подсетимо се, у сваком кораку на текуће место у низу доводи најмањи од преосталих елемената низа (у првом кораку се на прво место доводи највећи (или најмањи) елемент целог низа, у другом кораку се на друго место доводи највећи од свих елемената низа осим оног постављеног на прво место итд.). Приметимо да ће се након k корака на почетку низа наћи тачно k највећих елемената и ту се алгоритам може зауставити-

ти. Пошто је за налажење најмањег елемента у низу потребно $O(n)$ операција, и то се ради k пута, временска сложеност овог алгоритма је $O(n \cdot k)$, док је меморијска сложеност $O(n)$.

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

const int N_MAX = 100000;
int a[N_MAX];

int main() {
  ios_base::sync_with_stdio(false);
  // ucitavamo ulazne podatke
  int n, k;
  cin >> n >> k;
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // sortiramo niz primenjujuci k rundi algoritam sortiranja
  // selekcijom
  for (int i = 0; i < k; i++) {
    int minPoz = i;
    for (int j = i + 1; j < n; j++)
      if (a[j] > a[minPoz])
    minPoz = j;
    swap(a[i], a[minPoz]);
  }

  // izracunavamo i ispisujemo zbir elemenata niza
  int s = 0;
  for (int i = 0; i < k; i++)
    s += a[i];
  cout << s << endl;

  return 0;
}
```

QuickSelect

Алгоритам брзе селекције (QuickSelect) представља модификацију алгоритма брзог сортирања који се може употребити да се низ подели тако да на првих k највећих (или најмањих) елемената низа, а након тога елементи мањи (или већи) од њих, при чему је редослед елемената у те две групе произвољан.

Алгоритам брзе селекције се може имплементирати и ручно и заснива на кораку партиционисања који у линеарној сложености елементе низа уређује тако да се прво у низу нађу елементи који су већи од неког датог елемента (тзв. пивота), затим пивот и након тога елементи који су мањи од пивота. Редослед елемената у свакој од ових

група је потпуно произвољан. Ако се пивот јавља више пута, остала појављивања пивота могу бити било лево, било десно од пивота (често се узима да се лево од пивота налазе елементи већи или једнаки од њега, а десно елементи строго мањи од њега или обратно). За партиционисање се могу користити поступци описани у задацима Двобојка или Тробојка. Осовни алгоритам је рекурзиван и параметар рекурзије су границе l и d и број k, и задатак алгоритма је да део низа на позицијама $[l, d]$ реорганизује тако да на почетку буде k најмањих елемената тог дела низа. Излаз из рекурзије је када је интервал $[l, d]$ једночлан или празан (када је $l \geq d$) или када је $k \leq 0$. Након избора пивота и партиционисања претпоставимо да се пивот нашао на месту m. Ако је број елемената лево од пивота (вредност $m - l$) већа од k онда је довољно наћи k највећих елемената левог дела низа (јер су сви елементи у левом мањи од елемената у десном делу, јер их након партиционисања раздваја пивот) тј. извршити рекурзивни позив за интервал $[l, m - 1]$ и број k. У супротном се закључно са пивотом налази $m - l + 1$ од k највећих елемената низа и зато је потребно у десном делу одредити још $k - (m - l + 1)$ највећих елемената из тог дела, тако да се рекурзивни позив врши за интервал $[m + 1, d]$ и број $k - m + l - 1$. Приметимо да у функцији постоји само један рекурзивни позив и то репни, тако да се он може једноставно елиминисати.

Под претпоставком да ће пивот делити низ на делове који су отприлике једнаке величине, сложеност овог алгоритма је линеарна $O(n)$. Пошто се цео низ учитава и чува у меморији и просторна сложеност је $O(n)$.

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

// QuickSelect - odredjujemo najvecih k elemenata niza a tj. niz permutujemo
// tako da se najvecih k elemenata nadju na prvih k pozicija (u proizvoljnom
// redosledu)
void qsortK(vector<int>& a, int l, int d, int k) {
  if (k <= 0 || l >= d)
    return;

  // niz particionisemo tako da se pivot (element a[l]) dovede na
  // svoje mesto, da ispred njega budu svi elementi koji su veci ili
  // jednaki od njega, a da iza njega budu svi elementi veci od njega
  int m = l;
  for (int t = l+1; t <= d; t++)
    if (a[t] >= a[l])
      swap(a[++m], a[t]);
  swap(a[m], a[l]);

  if (k < m - l)
    // svih k elemenata su levo od pivota - obradjujemo deo ispred pivota
    qsortK(a, l, m - 1, k);
  else
    // neki kod k najvecih su iza pivota - obradjujemo deo iza pivota
```

```
    qsortK(a, m+1, d, k - (m - l + 1));
}

// QuickSelect - pomocna funkcija zbog lepseg interfejsa
void qsortK(vector<int>& a, int k) {
  qsortK(a, 0, a.size() - 1, k);
}

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo ulazne podatke
  int n, k;
  cin >> n >> k;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
      cin >> a[i];

  // odredjujemo prvih k najvecih elemenata niza
  qsortK(a, k);

  // sabiramo prvih k elemenata niza i ispisujemo rezultat
  int s = 0;
  for (int i = 0; i < k; i++)
    s += a[i];

  cout << s << endl;
  return 0;
}
```

Рачунање збира током партиционисања

Уместо да елементе прво распоредимо тако да k највећих елемената буде на почетку, па тек затим да их сабирамо, функција може бити дефинисана тако да истовремено распоређује елементе и рачуна њихов збир.

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

// QuickSelect - odredjujemo zbir k najvecih elemenata dela niza [l, d]
int zbirKNajvecih(vector<int>& a, int l, int d, int k) {
  if (k == 0)
    return 0;

  // niz particionisemo tako da se pivot (element a[l]) dovede na
  // svoje mesto, da ispred njega budu svi elementi koji su veci ili
  // jednaki od njega, a da iza njega budu svi elementi veci od njega
  int m = l;
```

```
  for (int t = l+1; t <= d; t++)
    if (a[t] >= a[l])
      swap(a[++m], a[t]);
  swap(a[m], a[l]);

  if (k < m - l + 1)
    // svih k elemenata su levo od pivota - obradjujemo deo ispred pivota
    return zbirKNajvecih(a, l, m - 1, k);
  else {
    int zbir = 0;
    for (int i = l; i <= m; i++)
      zbir += a[i];
    // neki kod k najvecih su iza pivota - obradjujemo deo iza pivota
    return zbir + zbirKNajvecih(a, m+1, d, k - (m - l + 1));
  }
}

// QuickSelect - pomocna funkcija zbog lepseg interfejsa
int zbirKNajvecih(vector<int>& a, int k) {
  return zbirKNajvecih(a, 0, a.size() - 1, k);
}

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo ulazne podatke
  int n, k;
  cin >> n >> k;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
      cin >> a[i];

  // odredjujemo zbir prvih k najvecih elemenata niza
  int zbir = zbirKNajvecih(a, k);

  cout << zbir << endl;
  return 0;
}
```

Пошто је рекурзија репна, она се лако елиминише.

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

// QuickSelect - zbir k najvecih elemenata niza
int zbirKNajvecih(vector<int>& a, int k) {
  int l = 0, d = a.size() - 1;
```

```
  int zbir = 0;
  while (k != 0) {
    // niz particionisemo tako da se pivot (element a[l]) dovede na
    // svoje mesto, da ispred njega budu svi elementi koji su veci ili
    // jednaki od njega, a da iza njega budu svi elementi veci od njega
    int m = l;
    for (int t = l+1; t <= d; t++)
      if (a[t] >= a[l])
        swap(a[++m], a[t]);
    swap(a[m], a[l]);

    if (k < m - l + 1)
      // svih k elemenata su levo od pivota - obradjujemo deo ispred pivota
      d = m - 1;
    else {
      for (int i = l; i <= m; i++)
        zbir += a[i];
      // neki kod k najvecih su iza pivota - obradjujemo deo iza pivota
      k -= m - l + 1;
      l = m + 1;
    }
  }
  return zbir;
}

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo ulazne podatke
  int n, k;
  cin >> n >> k;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // odredjujemo zbir prvih k najvecih elemenata niza
  int zbir = zbirKNajvecih(a, k);

  cout << zbir << endl;
  return 0;
}
```

Библиотечка функција У језику C++ библиотечка функција `nth_element` врши поделу низа тако да се на позицији n нађе елемент који ту и припада у сортираном редоследу, да се испред те позиције нађу елементи који су сви мањи или једнаки од њега, а да се иза те позиције нађу елементи који су сви већи или једнаки од њега. Функцији се прослеђује итератор на почетак дела низа (вектора) који се обрађује (обично добијен помоћу `begin`), итератор на неку позицију на средини низа и итератор који указује

непосредно иза краја низа (обично добијен помоћу `end`). Ако средишњи итератор указује на n-ту позицију у низу након примене функције на тој позицији ће се наћи n-ти по величини елемент, док ће сви елементи лево од њега бити мањи или једнаки од свих елемената десно од њега. Рецимо и да постоји функција `partial_sort` која је слична претходној али уједно елементе испред дате позиције уређује (соритра) по величини, међутим, у овом случају то нам није потребно и тиме би се само непотребно губило време.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <numeric>
using namespace std;

int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo ulazne podatke
  int n, k;
  cin >> n >> k;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
      cin >> a[i];

  // niz particionisemo tako da je  n-ti element na svom mestu i da su
  // svi elementi ispred njega manji ili jednaki od svih elemenata iza
  nth_element(a.begin(), next(a.begin(), k), a.end(), greater<int>());

  // odredjujemo i ispisujemo zbir prvih k elemenata transformisanog niza
  cout << accumulate(a.begin(), next(a.begin(), k), 0) << endl;

  return 0;
}
```

Уметање

Једно решење је да се k највећих елемената низа чувају у нерастуће сортираном низу и да се примењује техника сортирања уметањем (види задатак Сортирање бројева). Ради једноставности имплементације чуваћемо низ од $k+1$ елемената. Када учитамо сваки следећи број поена, стављаћемо га иза последњег постављеног елемента у том низу док се низ не попуни, односно на последњу позицију (ону са индексом k), и затим применом алгоритма уметања померати елемент улево док не дође на своју позицију у том низу (ако је тај елемент није међу највећих k у до сада учитаним елементма, он ће остати на позицији $k+1$ и у следећем учитавању бити замењен). Уметање можемо реализовати тако што у привремену промењиву учитамо тај последњи елемент низа, затим све елементе који су мањи од њега померимо за једно место удесно и на крају њега упишемо на место елемента који је последњи померен. На крају, сабирамо елементе на првих k позиција тог низа ($k+1$-ви елемент на позицији k није релевантан).

Пошто се у сваком кораку елементи могу померати k пута, а имамо укупно n корака и временска сложеност овог алгоритма је лоша и износи $O(n \cdot k)$, при чему је просторна сложеност боља и износи $O(k)$.

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

int main() {
  ios_base::sync_with_stdio(false);

  int n, k;
  cin >> n >> k;

  vector<int> a(k + 1);
  cin >> a[0];
  for (int i = 1; i < n; i++) {
    int x;
    cin >> x;
    // umece x na odgovarajuce mesto u prvih min(i, k) nerastuce
    // uredjenih elemenata tako da je niz i dalje u nerastucem poretku
    int j;
    for (j = min(i, k) - 1; j >= 0 && x > a[j]; j--)
      a[j + 1] = a[j];
    a[j + 1] = x;
  }

  // izracunavamo i ispisujemo zbir elemenata niza
  int s = 0;
  for (int i = 0; i < k; i++)
    s += a[i];

  cout << s << endl;

  return 0;
}
```

Ред са приоритетом

Највећих k до сада виђених елемената низа можемо чувати у структури података која нам омогућава да пронађемо најмањи елемент у њој и да га евентуално заменимо оним који је тренутно учитан (ако је тренутно учитани елемент већи од њега). Идеална структура за то је хип тј. ред са приоритетом.

Ред са приоритетом у језику C++ можемо добити помоћу `priority_queue`. Елементе у ред можемо убацити методом `push`. Елемент који је најмањи можемо очитати методом `top` и избацити методом `pop`.

На почетку ред попуњавамо са k првих учитаних елемената, а затим сваки наредни

учитани елемент поредимо са најмањим у реду и ако је већи од њега, најмањи избацујемо, а учитани елемент убацујемо. Пошто је сложеност метода за убацивање и избацивање из реда са приоритетом логаритамска, а методе за очитавање најмањег елемента константна, временска сложеност овог алгоритма је $O(n \cdot \log(k))$, док је просторна сложеност $O(k)$. Приметимо да је овај алгоритам донекле сличан алгоритму сортирања уз помоћ хипа тј. алгоритма Хип-сорт (HeapSort).

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

int main() {
  ios_base::sync_with_stdio(false);
  int n, k;
  cin >> n >> k;

  // red sa prioritetom koji cuva k najvecih elemenata koristi se
  // min-hip, koji omogucava brzo uklanjanje najmanjeg elementa
  priority_queue<int, vector<int>, greater<int>> pq;

  // ucitavamo prvih k elemenata i ubacujemo ih u red
  for (int i = 0; i < k; i++) {
    int x;
    cin >> x;
    pq.push(x);
  }

  // ucitavamo preostale elemente
  for (int i = k; i < n; i++) {
    int x;
    cin >> x;
    // ako je ucitani element veci od najmanjeg trenutno u redu
    // izbacujemo taj najmanji i menjamo ga ucitanim
    if (x > pq.top()) {
      pq.pop();
      pq.push(x);
    }
  }

  // izbacujemo elemente iz reda racunajuci njihov zbir i ispisujemo ga
  int s = 0;
  while (!pq.empty()) {
    s += pq.top();
    pq.pop();
  }
  cout << s << endl;

  return 0;
```

```
}
```

Задатак: Број инверзија

Поставка: Напиши програм који одређује колико у низу има инверзија (позиција $0 \leq i < j < n$, таквих да је $a_i > a_j$.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 50000$) и затим n целих бројева, сваки у посебном реду.

Излаз: На стандардни излаз исписати само тражени број инверзија.

Пример

Улаз	*Излаз*
5 3 1 4 2 5	3

Решење:

Груба сила

Грубом силом се задатак решава тако што се помоћу угнежђених петљи испитају сви парови позиција $0 \leq i < j < n$ и преброје сви случајеви када је $a_i > a_j$ (бројимо елементе филтриране серије). Сложеност овог алгоритма одговара броју парова, а то је $O(n^2)$.

```
#include <iostream>
#include <vector>

using namespace std;

int broj_inverzija(const vector<int>& a) {
  int n = a.size();
  int broj = 0;
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++)
      if (a[j] < a[i])
        broj++;
  return broj;
}

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

```
  cout << broj_inverzija(a) << endl;;
  return 0;
}
```

Подели па владај - модификација алгоритма MergeSort

Размотримо како бисмо проблем решили декомпозицијом. Празан и једночлан низ немају инверзија. Ако је низ подељен на две половине, укупан број инверзија једнак је збиру броја инверзија међу елементима прве половине, броја инверзија међу елементима друге половине и броја парова елемената где први елемент припада првој, други елемент припада другој половини и први је већи од другог. Прва два броја можемо одредити рекурзивно и остаје само питање како ефикасно одредити трећи број. Да бисмо добили укупну сложеност $O(n \log n)$ тај проблем је потребно решити у сложености $O(n)$ тако да испитивање свих парова елемената из прве и друге половине не долази у обзир. Задатак би се могао лакше решити ако би прва и друга половина биле сортиране (кључни увид је да сортирање елемената тих половина не мења трећи број). Тада можемо применити технику два показивача и веома слично као у случају обједињавања два сортирана низа одредити жељени трећи број. Уместо да сортирамо половине засебно, можемо алгоритам сортирања интегрисати са бројањем инверзија и проширити инваријанту наше функције тако да током бројања инверзија уједно сортира низ. На основу инваријанте, рекурзивни позиви ће сортирати леву и десну половину, а да бисмо је одржали, током одређивања трећег броја вршићемо обједињавање сортираних низова (исто као у алгоритму MergeSort).

```
#include <iostream>
#include <vector>

using namespace std;

int broj_inverzija(vector<int>& a, int l, int d, vector<int>& b) {
  if (l >= d)
    return 0;
  int s = l + (d - l) / 2;
  int broj = 0;
  broj += broj_inverzija(a, l, s, b);
  broj += broj_inverzija(a, s+1, d, b);
  int pl = l, pd = s+1, pb = 0;
  while (pl <= s && pd <= d) {
    if (a[pl] <= a[pd])
      b[pb++] = a[pl++];
    else {
      broj += s - pl + 1;
      b[pb++] = a[pd++];
    }
  }
  while (pl <= s)
    b[pb++] = a[pl++];
  while (pd <= d)
```

```
    b[pb++] = a[pd++];

  copy(begin(b), next(begin(b), d - l + 1), next(begin(a), l));

  return broj;
}

int broj_inverzija(const vector<int>& a) {
  vector<int> tmp1(a.size()), tmp2(a.size());
  copy(begin(a), end(a), begin(tmp1));
  return broj_inverzija(tmp1, 0, a.size()-1, tmp2);
}

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

  cout << broj_inverzija(a) << endl;;
  return 0;
}
```