# Садржај

# Глава 1

# 1. круг квалификација

## Задатак: Средња брзина

*Аутор: Филип Марић*

Тело је први део пута прешло крећући се равномерно, једном брзином, а други део пута такође равномерно, крећући се неком другом брзином напиши програм који одређује средњу брзину овог кретања.

*Напомена*: средња брзина се израчунава тако што се укупан пређени пут подели са укупним протеклим временом.

**Опис улаза**

Са стандардног улаза се учитава цео број $s_1$, затим цео број $t_1$, затим цео број $s_2$ и цео број $t_2$ (сваки број је у посебном реду). Вредности $s_1$ и $s_2$ представљају број метара које је возило прешло у првом и у другом делу пута. Бројеви $t_1$ и $t_2$ представљају број секунди које је тело провело крећући се у првом и у другом делу пута.

**Опис излаза**

На стандардни излаз испсиати средњу брзину у метрима у секунди.

**Пример 1**

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| 4<br>2<br>8<br>4 | 2 | $v_{sr} = \frac{s_1 + s_2}{t_1 + t_2} = \frac{4+8}{2+4} = \frac{12}{6} = 2$ |

**Пример 2**

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| 50<br>8<br>35<br>2 | 8.5 | $v_{sr} = \frac{s_1 + s_2}{t_1 + t_2} = \frac{50+35}{8+2} = \frac{85}{10} = 8.5$ |

### Решење

Улазне податке ћемо учитати у 4 променљиве (`s1`, `t1`, `s2` и `t2`). Затим ћемо применом формуле израчунати средњу брзину. На крају ћемо ту средњу брзину и исписати.

```
#include <iostream>

using namespace std;

int main() {
  int s1, s2, t1, t2;
  cin >> s1 >> t1 >> s2 >> t2;
  double vsr = (double)(s1 + s2) / (double)(t1 + t2);
  cout << vsr << endl;
  return 0;
}
```

## Задатак: Атлетичари

*Аутор: Филип Марић*

У зависности од трка које трче, атлетичаре делимо на краткопругаше, средњепругаше и дугопругаше. Краткопругаши (или спринтери) трче трке дужине највише 400 метара, средњепругаши трче дуже трке од њих, али највише дужине једне миље, док дугопругаши трче трке дуже од тога. Написати програм којим се на основу дужине трке у којој атлетичар учествује одређује категорију којој он припада.

**Опис улаза**

Дужина трке у метрима - цео број од 60 до 42195.

**Опис излаза**

На стандардни излаз исписати једну од следећих речи: `kratko`, `srednje`, `dugo`.

**Пример 1**

| *Улаз* | *Излаз* |
|---|---|
| 400 | kratko |

**Пример 2**

| *Улаз* | *Излаз* |
|---|---|
| 1500 | srednje |

### Решење

Из формулације задатка произилази да имамо три категорије, које су одређене следећим интервалима:

- ако дужина у метрима припадају интервалу $[60, 400]$, тркач је краткопругаш;
- ако дужина у метрима припада интервалу $(400, 1609]$, тркач је средњепругаш (једна миља има 1609 метара);
- ако дужина у метрима припада интервалу $(1609, 42195]$, тркач је дугопругаш.

Одавде произилази кôд са три међусобно независна услова, којима се у произвољном редоследу проверава припадност температуре једном од три интервала $[60, 400]$, $(400, 1609]$ и $(1609, 42195]$. Пошто знамо да је унети број метара увек у интервалу $[60, 42195]$, довољно је испитивати припадност интервалима $(-\infty, 400]$, $(400, 1609]$ и $(1609, +\infty)$.

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

int main() {
  int m; // Duzina trke u metrima
  cin >> m;

  if (m <= 400)
    cout << "kratko" << endl;
  if (m > 400 && m <= 1609)
    cout << "srednje" << endl;
  if (m > 1609)
    cout << "dugo" << endl;

  return 0;
}
```

Међутим, до решења се може доћи и уз коришћење такозване *конструкције* `else-if`, следећим поступком:

- ако број метара није већи од $400$, тркач је краткопругаш;
- у супротном (број метара јесте већи од $400$): ако је број метара није већи од $1609$ (припада другом интервалу), тркач је средњепругач;
- у супротном (број метара је већи од $1609$), тркач је дугопругаш.

Овим се избегавају провере које нису заиста неопходне.

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

int main() {
  int m; // Duzina trke u metrima
  cin >> m;

  if (m <= 400)
    cout << "kratko" << endl;
  else if (m <= 1609)
    cout << "srednje" << endl;
  else
    cout << "dugo" << endl;

  return 0;
}
```

Аналогно претходном решењу, можемо проверавати припадност броја метара идући здесна од интервала $(1609, \infty)$ наниже.

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

int main() {
  int m; // Duzina trke u metrima
```

```
  cin >> m;

  if (m > 1609)
    cout << "dugo" << endl;
  else if (m > 400)
    cout << "srednje" << endl;
  else
    cout << "kratko" << endl;

  return 0;
}
```

# Задатак: Гпс

*Аутор: Владимир Кузмановић*

Перица је за рођендан добио паметни спортски сат који омугућава праћење кретања и пређеног растојања уз помоћ ГПС-а (глобални систем за позиционирање). Перица је отишао са школом на екскурзију са богатим културним програмом и жели да тестира свој нови сат. Током екскурзије ученици треба да посете $n$ локација. Перица жели да премери укупно растојање које ће прећи уз помоћ свог новог сата и замолио нас је да напишемо програм који ће му у томе помоћи.

**Опис улаза**

У првој линији стандардног улаза се учитава природан број $n$, $1 \leq n \leq 10^6$, и затим у следећих $n+1$ линија стандардног учитава се по један природан број који редом представљају растојања између две узастопне деонице пута. Растојање између две деонице је број у интервалу $[50, 250]$

**Опис излаза**

На стандардни излаз исписати један природан број који представља укупан пређени пут.

**Пример**

| *Улаз* | *Излаз* |
|---|---|
| 5<br>17<br>25<br>48<br>37<br>14<br>67 | 208 |

## Решење

Основна идеја решења задатка може бити да прво учитамо свих $n+1$ реалних бројева у помоћни низ. Након што смо сачували бројеве у помоћном низу, потребно је да их све саберемо и тако одредимо укупни пређени пут.

Иако је идеја једноставна, задатак се на вишим нивоима такмичења не може у потпуности решити исправно на овај начин. Разлог зашто не може, најчешће се крије у постављеним границама за број $n$ и временским и меморијским ограничењима програма. На пример, да је у услову задатка ограничење за број $n$ било $1 \leq n \leq 10^9$, тада би нам за горњу границу вредности броја

$n$, тј. број 1000000000, било потребно приближно 3.8 гигабајта меморије за смештање свих $10^9$ бројева, јер сваки цео број заузима 4 бајта у меморији. Поред огромне количине меморије, биће нам потребно и много више времена од дозвољених 2 секунде само за учитавање тих бројева. Јасно је да са доступним ресурсима то не можемо да остваримо.

Са доступним меморијским ограничењем могли бисмо да учитамо 15ак милиона целих бројева, што би био само мали подскуп могућих димензија улаза у проширеној верзији задатка. Дакле, потребно је да смислимо решење које неће користити помоћне низове, ако желимо у потпуности исправно да решимо задатак.

*Напомена:* На општинском нивоу такмичења се не гледа меморијска сложеност програма, али то што ученици могу да користе додатну количину меморије у својим решењима не значи да је то у потпуности иправно нити да то треба да раде. На нижим нивоима такмичења висока меморијска сложеност програма не доноси негативне поене, али то чини на вишим нивоима. Са тим у вези, ученици од старта треба да уче да је обазрива употреба ресурса у програму заправо срж такмичарског програмирања.

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

using namespace std;

int main() {

   int n = 0;
   cin >> n;

   vector<int> rastojanja;
   for (int i = 1; i <= n + 1; i++) {
        int x;
        std::cin >> x;
        rastojanja.push_back(x);
   }

   int ukupno = 0;
   for (int i = 0; i < rastojanja.size(); i++) {
        ukupno += rastojanja[i];
   }

   cout << ukupno << endl;

   return 0;
}
```

Алтернатива првом решењу је да збир свих елемената низа израчунамо уграђеном функцијом. С обзиром да користимо помоћни низ, ни ово решење није у потпуности исправно.

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

```
using namespace std;

int main() {

   int n = 0;
   cin >> n;

   vector<int> rastojanja;
   for (int i = 1; i <= n + 1; i++) {
        int x;
        std::cin >> x;
        rastojanja.push_back(x);
   }

   int ukupno = accumulate(rastojanja.begin(), rastojanja.end(), 0);

   cout << ukupno << endl;

   return 0;
}
```

Решење које је у потпуности исправно, користи инкременталност да би одредило збир елемената. Потребно је да током учитавања истовремено рачунамо збир елемената. На тај начин постижемо да меморијска сложеност нашег програма буде константа, тј. да не зависи од броја $n$.

Инкременталност је лако уочити. Ако треба да израчунамо збир $k$ бројева, довољно је да чувамо само збир претходних $k-1$ бројева, али не и саме бројеве. Тада, тражени збир $k$ бројева добијамо тако што познати збир $k-1$ бројева увећамо за вредност $k$-тог броја.

У строго типизираном језику као што је Ц++ морамо да водимо рачуна о типовима које користимо. У нашем програму у најгорем случају можемо да учитамо $10^6$ бројева који редом представљају растојање једне деоницу. Једна деоница може имати највећу дужину од 250 мерних јединица. То значи да ако саберемо свих $10^6$ најдужих деоница, укупан збир у најгорем случају може износити највише $250 * 10^6$ што се и даље може представити `int` типом у програмском језику Ц++. У случају да је збир у најгорем случају већи од *2.1*10^9*, морали бисмо да користимо неки шири тип података попут `long long`.

```
#include <iostream>

using namespace std;

int main() {

   int n = 0;
   cin >> n;

   int ukupno = 0;
   for (int i = 1; i <= n + 1; i++) {
      int x;
      cin >> x;
```

```
      ukupno += x;
   }

   cout << ukupno << endl;

   return 0;
}
```

# Задатак: Најуспешнија партија

*Аутор: Владимир Кузмановић*

Перица се спрема за међународно такмичење у електронским спортовима и одлучио је да води статистику о броју поена које осваја након сваке партије своје омиљене игрице. Припреме за такмичење трају укупно $n$ дана. С обзиром да Перица има и разне друге обавеза током дана, одлучио је да сваког дана игра тачно једну партију и да на крају сваке партије запише број освојених поена. Потребно је да напишемо програм који ће помоћи Перици да одреди редни број дана када је освојио највише поена. У случају да постоји више дана у којима је освојио максимални број поена, као резултат се узима последњи такав дан. *Напомена: Редни бројеви дана почињу од 1.*

### Опис улаза

У првој линији стандардног улаза се учитава природан број $n$, $1 \leq n \leq 10^6$, и затим у следећих $n$ линија стандардног учитава се по један природан број $x_i$, $1 \leq x_i \leq 10^9$, који редом представљају број освојених поена тога дана.

### Опис излаза

На стандардни излаз исписати редни број последњег дана у којем је освојен највећи број поена.

### Пример

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| 10<br>250<br>300<br>180<br>400<br>260<br>400<br>380<br>400<br>370<br>240 | 8 | Перица је освојио највише 400 поена и то 4., 6. и 8. дана. Према услову задатка треба да испишемо редни број последњег дана када је Перица освојио највише поена, па ће резултат бити 8. |

## Решење

Основна идеја решења задатка може бити да прво учитамо свих $n+1$ реалних бројева у помоћни низ. Након што смо сачували бројеве у помоћном низу, потребно је да одредимо позицију последњег појављивања максимума.

Иако је идеја једноставна, задатак се на вишим нивоима такмичења не може у потпуности решити исправно на овај начин. Разлог зашто не може, најчешће се крије у постављеним границама за број $n$ и временским и меморијским ограничењима програма. На пример, да је у услову задатка ограничење за број $n$ било $1 \leq n \leq 10^9$, тада би нам за горњу границу вредности броја $n$, тј. број $1000000000$, било потребно приближно $3.8$ гигабајта меморије за смештање свих $10^9$ бројева, јер сваки цео број заузима 4 бајта у меморији. Поред огромне количине меморије, биће нам потребно и много више времена од дозвољених 2 секунде само за учитавање тих бројева. Јасно је да са доступним ресурсима то не можемо да остваримо.

Са доступним меморијским ограничењем могли бисмо да учитамо $15$ак милиона целих бројева, што би био само мали подскуп могућих димензија улаза у проширеној верзији задатка. Дакле, потребно је да смислимо решење које неће користити помоћне низове, ако желимо у потпуности исправно да решимо задатак.

*Напомена:* На општинском нивоу такмичења се не гледа меморијска сложеност програма, али то што ученици могу да користе додатну количину меморије у својим решењима не значи да је то у потпуности иправно нити да то треба да раде. На нижим нивоима такмичења висока меморијска сложеност програма не доноси негативне поене, али то чини на вишим нивоима. Са тим у вези, ученици од старта треба да уче да је обазрива употреба ресурса у програму заправо срж такмичарског програмирања.

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

using namespace std;

int main() {

    int n;
    cin >> n;

    std::vector<int> poeni;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        poeni.push_back(x);
    }

    int maxInd = 0;
    for (int i = 1; i < poeni.size(); i++) {
        if (poeni[i] >= poeni[maxInd]) {
            maxInd = i;
        }
    }

    cout << (maxInd + 1) << endl;

    return 0;
}
```

Алтернатива првом решењу је да позицију последњег појављивања максимума израчунамо угра-

ђеним функцијама. Иако су решења са уграђеним функцијама углавном кратка и елегантна, често се иза уграђених функција крије скривена временска сложеност. Наиме, уграђене функције су често ефикасније од онога што ми можемо да напишемо ако се ради о уобичајеним алгоритмима, али нису магично решење свих проблема, јер нису тако лако прилагодљиве различитим случајевима. Сложеност се крије управо у том прилагођавању програма ономе што нам нуде уграђене функције.

У програмском језику Ц++ постоји уграђена функција која може да одреди максимални елемент набројиве колекције (низ, вектор, ...) и постоји уграђена функција која може да одреди прво појављивање неког елемента $x$ у набројивој колекцији. Обе функције се налазе у заглављу *algorihm*, део су *std* простора имена и редом се зову *max* и *find*. Међутим, не постоји уграђена функција која може истовремено да одреди максимални елемент и његову позицију у низу.

Због тога, морамо прво да одредимо максимални елемент колекције уз помоћ функције *max*, затим да одредимо његово прво појављивање полазећи од краја низа уз помоћ функције *find*. Обе функције нам као свој резултат враћају итераторе. Ово је место на којем се крије сложеност, јер ћемо описаним поступком два пута проћи кроз цео низ. У првом пролазу ћемо одредити максимум, а у другом његову позицију.

На крају, потребно је да одредимо разлику итератора уз помоћ функције *distance* и тиме редни број позиције на којој се максимум последљи пут јавља.

С обзиром да користимо помоћни низ, ни ово решење није у потпуности исправно. Поред тога, скривена сложеност је још једна замка коју треба да избегнемо.

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

using namespace std;

int main() {

    int n;
    cin >> n;

    std::vector<int> poeni;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        poeni.push_back(x);
    }

    std::vector<int>::reverse_iterator maxElement = max_element(poeni.rbegin(), poeni.rend());
    std::vector<int>::reverse_iterator maxPos = find(poeni.rbegin(), poeni.rend(), *maxElement);
    int maxInd = distance(maxPos, poeni.rend()) - 1;

    cout << (maxInd + 1) << endl;

    return 0;
}
```

Решење које је у потпуности исправно, користи инкременталност да би одредило последњу позицију максимума. Потребно је да током учитавања истовремено рачунамо позицију максимума. На тај начин постижемо да меморијска сложеност нашег програма буде константа, тј. да не зависи од броја $n$, као и одређивање резултата у тачно једном пролазу кроз учитане бројеве.

Инкременталност је лако уочити. Ако треба да израчунамо максимум $k$ бројева, довољно је да чувамо само максималну вредност претходних $k-1$ бројева, али не и саме бројеве. Тада, тражену позицију максимума $k$ бројева добијамо тако што упоредимо познати максимум $k-1$ бројева са вредности $k$-тог броја и уколико је потребно ажурирамо раније упамћену позицију максимума.

```
#include <iostream>

using namespace std;

int main() {

   int n;
   cin >> n;

   int max = -1;
   int redniBroj = -1;
   for (int i = 1; i <= n; i++) {
      int x;
      cin >> x;
      if (x >= max) {
         max = x;
         redniBroj = i;
      }
   }

   cout << redniBroj << endl;

   return 0;
}
```

## Задатак: Имена презимена

*Аутор: Душан Попадић*

У одељењу има $n$ дечака и $m$ девојчица. За потребе позоришне представе коју спремају у одељењу, потребно је изабрати једну девојчицу и једног дечака за главне улоге. Исписати све могуће парове које наставник може изабрати за главне улоге.

### Опис улаза

У првом реду стандардног улаза се уносе бројеви $n$ и $m$ ($1 \leq n, m \leq 30$) раздвојени размаком. У $n$ редова се налазе се имена дечака. У наредних $m$ редова налазе се имена девојчица. Свако име је једна реч која има највише 10 слова.

### Опис излаза

На излаз исписати све могуће парове, сваки пар у посебном реду. У оквиру једног пара треба да се испише прво име дечака, па девојчице, а имена треба да буду раздвојена размаком. Парове исписати у произвољном редоследу.

**Пример 1**

| *Улаз* | *Излаз* |
|---|---|
| `2 3` | `nikola katarina` |
| `nikola` | `nikola saska` |
| `marko` | `nikola nevena` |
| `katarina` | `marko katarina` |
| `saska` | `marko saska` |
| `nevena` | `marko nevena` |

**Пример 2**

| *Улаз* | *Излаз* |
|---|---|
| `3 2` | `bosko dunja` |
| `bosko` | `bosko marija` |
| `boris` | `boris dunja` |
| `bosko` | `boris marija` |
| `dunja` | `bosko dunja` |
| `marija` | `bosko marija` |

## Решење

## Опис главног решења

Потребно је унети све дечаке и девојчице у два низа, а затим итерирати по њиховим елементима са две угњеждене петље: једна петља итерира по дечацима, док ће друга по девојчицама. Овако састављен пар дечака и девојчице је потребно исписати.

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

using namespace std;

int main() {

  int n, m;
  cin >> n >> m;
  cin.ignore();

  vector<string> decaci(n);
  vector<string> devojcice(m);

  for(int i = 0; i < n; i++)
    getline(cin, decaci[i]);

  for(int j = 0; j < m; j++)
    getline(cin, devojcice[j]);

  for(int i = 0; i < n; i++)
    for(int j = 0; j < m; j++)
      cout << decaci[i] << " " << devojcice[j] << endl;

  return 0;
}
```

# Задатак: Куповина скија

*Аутор: Душан Пођадић*

Дуња је наградној игри освојила ваучер од $k$ динара за куповину у једној познатој радњи спортске опреме. Она је одлучила да га искористи како би себи купила нове скије и нове панцерице (ципеле за скијање). У понуди је $n$ врста скија и $m$ врста панцерица и за сваку врсту се зна цена за један пар. Дуња не жели да плаћа додатно у односу на оно што има на ваучеру, али такође не би волела да јој пропадне превише новца са ваучера. Зато је замолила вас да за њу изаберете један пар скија и један пар панцерица тако да укупна цена не пређе $k$, али да искористи што више новца са ваучера (тј. да цена буде што приближнија $k$).

### Опис улаза

У првом реду стандардног улаза се уносе три броја $k(1000 \leq k \leq 10^9)$, $n$ и $m(1 \leq n, m \leq 10^5)$. У наредних $n$ редова се уноси по један природан број $s_i(1 \leq s_i \leq 10^9)$ који представља цену $i$-тог пара скија. У наредних $m$ редова се уноси по један природан број $p_i(1 \leq p_i \leq 10^9)$ који представља цену $i$-тог пара панцерица.

Гарантује се да ће Дуња моћи да купи барем један пар скија и један пар панцерица.

*Додатна ограничења*

Тест примери су подељени у групе, тј. подзадатке:

- у тест примерима вредним $10$ поена додатно важи $s_i, p_i \leq 100$
- у тест примерима вредним $40$ поена додатно важи $n, m \leq 10^3$
- у тест примерима вредним $50$ поена нема додатних ограничења

### Опис излаза

У једином реду стандардног излаза исписати један број који представља колико ће Дуњи остати новца на ваучеру после куповине.

### Пример 1

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| `35000 3 2`<br>`22500`<br>`17300`<br>`5000`<br>`14000`<br>`7000` | `3700` | Могуће комбинације цена за пар скија и панцерица су 22500+14000=36500, 22500+7000=29500, 17300+14000=31300, 17300+7000=24300, 5000+14000=19000 и 5000+7000=12000 динара. Дакле, Дуњи се највише исплати да купи скије и панцерице за 31300 динара јер је то највећи број за који не мора да додаје новац (тј. довољан јој је само ваучер). Дуњи пропада 35000-31300=3700 динара. |

### Пример 2

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| `20000 3 3`<br>`16000`<br>`12000`<br>`3800`<br>`8000`<br>`3000`<br>`5000` | `0` | Дуњи се највише исплати да купи скије за 12000 и панцерице за 8000. Дакле укупно ће платити 20000 динара што је и вредност ваучера, па пропада 0 динара. |

## Решење

## Решење бирањем најскупљих скија и панцерица

У првом подзадатку који доноси 10 поена су све цене мање од 100 одакле се може закључити да ће цена скија и панцерица бити највише 200 динара, што је сигурно мање од буџета (буџет је по тексту задатка сигурно макар 1000 динара). Дакле бирамо најскупље панцерице и најскупље скије.

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

using namespace std;

int main() {

  int k, n, m;

  cin >> k >> n >> m;

  int maxSkije = 0;
  int maxPancerice = 0;

  for(int i = 0; i < n; i++){
    int cena;
    cin >> cena;
    maxSkije = max(maxSkije, cena);
  }

  for(int i = 0; i < m; i++){
    int cena;
    cin >> cena;
    maxPancerice = max(maxPancerice, cena);
  }

  int najCena = maxSkije + maxPancerice;

  cout << k - najCena;

  return 0;
}
```

## Решење генерисањем свих парова

Задатак је могуће решити тако што се за сваки пар скија провери збир цена са сваким паром панцерица и онда изаберемо највећи збир цена који је мањи или једнак буџету.

**Анализа сложености**

Како је потребно проћи кроз све парове цена скија и панцерица којих има $n$ односно $m$, сложеност је $O(n \cdot m)$. Ово решење доноси 40 поена.

Могуће је комбиновати ово решење са решењем првог подзадатка тако да се добије 50 поена. Потребно је одредити максимуме оба низа и уколико су у буџету изабрати најскупље скије и панцерице, иначе проверити све парове.

```
// naivno resenje: sve kombinacije O(n^2)

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

using namespace std;

int main() {

  int k, n, m;

  cin >> k >> n >> m;

  vector<int> skije(n);
  vector<int> pancerice(m);

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

  for(int i = 0; i < m; i++)
    cin >> pancerice[i];

  int najCena = 0;

  for(int i = 0; i < n; i++)
    for(int j = 0; j < m; j++)
    {
      int cena = skije[i] + pancerice[j];
      if(cena <= k)
        najCena = max(cena, najCena);
    }

  cout << k - najCena;

  return 0;
}
```

## Решење коришћењем два показивача

Задатак је могуће решити применом технике два показивача. Прво сортирамо оба низа и затим крећемо од најјефтинијих скија и најскупљих панцерица. Уколико им је цена већа од буџета

тада бирамо јефтиније панцерице (померамо показивач у низу панцерица улево). Уколико им је цена у оквиру буџета онда је памтимо као потенцијално решење и бирамо скупље скије (померамо показивач у низу панцерица удесно).

### Анализа сложености

Сортирање низова је сложености $O(n \cdot \log(n))$, односно $O(m \cdot \log(m))$. Након што се низови сортирају потребно је још по једном проћи кроз сваки ($O(n)$ односно $O(m)$) што не утиче на укупну сложеност. Дакле, укупна сложеност је $O(n \cdot \log(n) + m \cdot \log(m))$. Ово решење доноси 100 поена.

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

using namespace std;

int main() {

  int k, n, m;

  cin >> k >> n >> m;

  vector<int> skije(n);
  vector<int> pancerice(m);

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

  for(int i = 0; i < m; i++)
    cin >> pancerice[i];

  // sortiranje oba niza neopadajuce
  sort(skije.begin(), skije.end());
  sort(pancerice.begin(), pancerice.end());

  // najbolja cena koju smo nasli do sada
  int najCena = 0;

  for(int i = 0, j = m - 1; i < n && j >= 0;){

    int cena = skije[i] + pancerice[j];

    if(cena > k) j--;
    else{
      najCena = max(najCena, cena);
      i++;
    }
  }
```

```
  cout << k - najCena;

  return 0;
}
```

## Решење коришћењем бинарне претраге

Задатак је могуће у потпуности решити и применом бинарне претраге. Потребно је сортирати низ панцерица и затим проћи кроз све цене скија, одредити колико остаје пара у буџету (зваћемо то буџет за панцерице) након што се те скије купе и онда у сортираном низу цена панцерица бинарном претрагом тражити највећу вредност која је мања или једнака од буџета за панцерице. Наравно, током ове претраге памтимо највећу укупну цену коју смо успели да постигнемо.

Алтернативно, може се радити са сортираним низом скија, а онда пролазити кроз све цене панцерица.

### Анализа сложености

Сортирање низа је сложености $O(m \cdot \log(m))$, а бинарне претраге кроз тај низ је $O(\log(m)$. Како се бинарна претрага у низу панцерица по једном позива за сваку цену скија, сложеност претраживања је $O(n \log(m))$, па је укупна сложеност $O(m \log(m) + n \log(m))$. Ово решење доноси 100 поена.

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

using namespace std;

int main() {

  int k, n, m;

  cin >> k >> n >> m;

  vector<int> skije(n);
  vector<int> pancerice(m);

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

  for(int i = 0; i < m; i++)
    cin >> pancerice[i];

  sort(pancerice.begin(), pancerice.end());

  int najCena = 0;

  for(int i = 0; i < n; i++){
```

```
    int cenaSkije = skije[i];
    int budzetPancerice = k - cenaSkije;
    // koristeci binarnu pretragu trazimo najvecu cenu pancerica koja je u okviru budzeta. Ako ne postoji cena ce biti -
    int indexPancerice = upper_bound(pancerice.begin(), pancerice.end(), budzetPancerice) - pancerice.begin() - 1;
    if(indexPancerice >= 0){
      int cenaPancerice = pancerice[indexPancerice];
      najCena = max(najCena, cenaSkije + cenaPancerice);
    }
  }

  cout << k - najCena;

  return 0;
}
```

## Задатак: Такси промо-кôд

*Аутор: Филип Марић*

Корисник је добио промо-кôд са којим плаћа вожњу таксијем 500 динара мање, али цена коју плаћа не сме бити нижа од цене поласка која је 300 динара. Написати програм који за унету цену вожње без попуста одређује колико корисник треба да плати након урачунавања овог попуста.

### Опис улаза

Са стандардног улаза се уноси цена без попуста. Цена је природан број мањи од 10000.

### Опис излаза

На стандардни излаз се исписује цена са попустом.

| **Пример 1** | | **Пример 2** | |
|---|---|---|---|
| *Улаз* | *Излаз* | *Улаз* | *Излаз* |
| 1300 | 800 | 600 | 300 |

## Решење

Нека је цена вожње без попуста $c$. Ако је вредност $c - 500$ мања од 300 динара, цена после попуста је 300, а у супротном је цена после попуста $c - 500$. Дакле, цена после попуста је максимум вредности 300 и $c - 500$.

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

using namespace std;

int main() {
  int cena_bez_popusta;
  cin >> cena_bez_popusta;
  cout << max(300, cena_bez_popusta - 500) << endl;
  return 0;
}
```

Задатак се може решити и експлицитним гранањем.

```
#include <iostream>

using namespace std;

int main() {
  int cena_bez_popusta;
  cin >> cena_bez_popusta;
  if (cena_bez_popusta > 800)
    cout << cena_bez_popusta - 500 << endl;
  else
    cout << 300 << endl;
  return 0;
}
```

# Задатак: Број сугласника

*Аутор: Оџељен Тешић*

На улазу је дата реч $s$ дужине $n$ (дужина речи је не више од 1000 карактера) која садржи мала слова **енглеске абецеде**. Избројати колико сугласника постоји у датој речи (бројати и понављања). *Напомена:* Ради једноставности сматрамо да је `r` увек сугласник.

### Опис улаза

У првом реду стандардног улаза је дата дужина речи $n$ ($1 \leq n \leq 1000$). У другом реду стандардног улаза је дата речи $s$.

### Опис излаза

У једином реду стандардног излаза исписати колико сугласника постоји у датој речи.

**Пример 1**

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| `10`<br>`ponedeljak` | `6` | У речи се јављају сугласници `p`, `n`, `d`, `l`, `j` и `k` (приметите да се `lj` не рачуна као један сугласник, већ као два - `l` и `j`). |

**Пример 2**

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| `9`<br>`radijator` | `5` | У речи се јављају сугласници `r`, `d`, `j`, `t` и `r`. |

## Решење

Проћи ћемо кроз целу реч карактер по карактер и проверити да ли је одређени карактер самогласник. Сваки пут када нађемо карактер који није самогласник, повећаћемо бројач сугласника. На крају, исписаћемо укупан број сугласника.

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

using namespace std;
```

```cpp
// Funkcija koja proverava da li je dati karakter samoglasnik.
bool jeSamoglasnik(char c) {
    return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
}

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

    int broj_suglasnika = 0;

    // Prolazak kroz svaki karakter u reci.
    for (char c : s) {
        if (!jeSamoglasnik(c)) {
            broj_suglasnika++;
        }
    }

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

# Задатак: Пресек интервала

*Аутор: Оџьен Тешић*

Дат је природан број $n$ ($2 \leq n \leq 1000$). Колико целих бројева се налази у пресеку затворених интервала $[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]$?

### Опис улаза

На стандардном улазу се уноси природан број $n$.

У наредних $n$ редова се уносе по два цела броја $a_i$ и $b_i$ ($a_i \leq b_i$). Сви бројеви који се уносе су по апсолутној вредности не већи од $1000$.

### Опис излаза

На стандардни излаз исписати тачно један број - број целих бројева који се налазе у пресеку.

### Пример 1

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| 4<br>-32 70<br>5 8<br>4 9<br>6 75 | 3 | Бројеви 6, 7 и 8 се налазе у сва четири затворена интервала. |

**Пример 2**

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| 2<br>1 5<br>5 2024 | 1 | Једини број који припада у оба затворена интервала је број 5. |

**Пример 3**

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| 2<br>5 9<br>10 15 | 0 | Не постоји број који се налази у оба интервала. |

## Решење

Решење задатка се своди на проналажење пресека затворених интервала. Да бисмо то одредили, потребно је да нађемо највећи број који је леви крај свих интервала, у ознаци $max\_a$, и најмањи број који је десни крај свих интервала, у ознаци $min\_b$.

Када добијемо те две вредности, можемо закључити следеће о пресеку: ако је $max\_a \leqslant min\_b$, онда је број целих бројева у том интервалу $min\_b - max\_a + 1$. Међутим, ако је $max\_a > min\_b$, то значи да интервали имају празан пресек, те је одговор 0.

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

int main() {
    int n;
    cin >> n;

    // Највећи почетак интервала постављамо на минималну вредност (-1000)
    int max_a = -1000;
    // Најмањи крај интервала постављамо на максималну вредност (1000)
    int min_b = 1000;

    for (int i = 0; i < n; i++) {
        int a, b;
        cin >> a >> b;
        max_a = max(max_a, a);
        min_b = min(min_b, b);
    }

    // Ако постоји пресек (почетна тачка <= крајња тачка)
    if (max_a <= min_b) {
        // Број целих бројева у пресеку је (min_b - max_a + 1)
        cout << (min_b - max_a + 1) << endl;
    } else {
        // Ако нема пресека, исписујемо 0
        cout << 0 << endl;
    }
```

```
    return 0;
}
```

# Задатак: Најдужи подниз са 3 парна броја

*Аутор: Душан Пођадић*

Дат је низ природних бројева дужине $n$. Одредити дужину најдужег подниза (узастопних елемената) у коме се налази највише три парна броја.

### Опис улаза

У првом реду стандардног улаза уноси се цео број $n$ ($1 \leq n \leq 5 \cdot 10^5$), дужина низа. У другом реду уноси се $n$ целих бројева $a_i$ ($1 \leq a_i \leq 10^9$), елементи низа. Елементи низа су раздвојени размацима.

### Опис излаза

Исписати на стандардни излаз дужину најдужег подниза у коме се налази највише три парна броја.

### Подзадаци

- $n \leq 5 \cdot 10^3$ (40 поена)
- без додатних ограничења (60 поена)

### Пример 1

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| `11`<br>`1 2 4 6 1 3 8 10 5 7 2` | `7` | Најдужи подниз са највише три парна броја је $[1, 3, 8, 10, 5, 7, 2]$ |

### Пример 2

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| `10`<br>`2 7 6 8 10 1 3 5 3 12` | `8` | Најдужи подниз је $[7, 6, 8, 10, 1, 3, 5, 3]$. |

### Пример 3

| *Улаз* | *Излаз* |
|---|---|
| `6`<br>`2 1 4 3 13 8` | `6` |

## Решење

Идеја решења је да применимо технику покретног прозора (енгл. *sliding window*) да бисмо пронашли најдужи узастопни подниз који садржи највише три парна броја.

Ова техника функционише тако што се користе два показивача, леви (`l`) и десни (`d`), који иницијално показују на почетак низа. Показивач `d` се помера надесно, додајући нове елементе у прозор. При сваком кораку, ако је нови елемент парни број, бројимо га у променљивој `broj_parnih`.

Док је `broj_parnih` мањи или једнак 3, настављамо са ширењем прозора увећавањем показивача `d`. Међутим, када `broj_parnih` пређе 3, што значи да тренутни подниз садржи више парних бројева него што је дозвољено, померамо леви показивач `l` надесно све док поново не задовољимо услов са највише три парна броја у прозору. При сваком померању `l`, ако из прозора избацимо парни број, умањујемо `broj_parnih` за 1.

Током проласка кроз низ пратимо највећу дужину прозора који задовољава услове и чувамо ту вредност као `max_duzina`. Сложеност овог приступа је $\mathcal{O}(n)$, јер се сваки показивач кроз низ помера највише једном.

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

using namespace std;

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

  int max_duzina = 0;
  int l = 0, d = 0;
  int broj_parnih = 0;

  while (d < n) {
    if (elementi[d] % 2 == 0)
      broj_parnih++;
    while (broj_parnih > 3) {
      if(elementi[l++] % 2 == 0)
        broj_parnih--;
    }
    max_duzina = max(max_duzina, d - l + 1);
    d++;
  }

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

# Задатак: Програмски језици

*Аутор: Владимир Кузмановић*

Перица се запослио као млади статистичар и као први задатак је добио да одреди резултате анкете о популарности програмских језика. У анкети је учествовало $n$ испитаника и од сваког испитаника је захтевано да наведе програмски језик који најчешће користи, као и оцену коју би дао том језику.

Број програмских језика које испитаници могу навести у анкети није унапред познат, али је познато да се језици у анкети могу јављати више од једанпут. Након обраде свих одговора испитаника, Перица треба да одреди просечну оцену заокружену на две децимале за сваки програмски језик који су испитаници навели и да направи извештај у којем ће резултати бити сортирани азбучно по називу програмског језика.

**Опис улаза**

У првој линији стандардног улаза учитава се број испитаника $n$, $1 \leq n \leq 10^6$. Затим се у следећих $n$ линија учитавају одговори испитаника. Сваки одговор садржи назив програмског језика и оцену раздвојене једним размаком. Назив програмског језика је реч дужине највише 7 слова, а оцена је цео број у интервалу $[0, 100]$.

**Опис излаза**

На стандардни излаз исписати резултате анкете сортиране азбучно по програмским језицима. У сваком реду стандардног излаза штампају се информације о једном програмском језику. За сваки програмски језик у истом реду исписати његов назив и просечну оцену заокружену на две децимале раздвојене једним размаком.

**Пример**

| *Улаз* | *Излаз* |
|---|---|
| `10` | `C 85.00` |
| `Python 87` | `C# 87.00` |
| `C++ 89` | `C++ 92.00` |
| `Java 83` | `Java 86.00` |
| `C++ 95` | `Python 93.00` |
| `Python 100` | |
| `C 85` | |
| `Python 92` | |
| `Java 89` | |
| `C++ 92` | |
| `C# 87` | |

## Решење

Наивно, у програму бисмо могли да чувамо низ програмских језика и да сваком језику придружимо низ оцена које су му испитаници доделили. Имајући у виду да је сваки одговор облика *језик оцена*, да се језици могу понављати и да их корисник не мора уносити у сортираном поретку, лако се уочава да за сваки одговор морамо да проверимо да ли је неки претходни испитаник већ навео тај програмски језик или не. У случају да јесте, тада у низ оцена тог програмског језика треба да додамо нову оцену. У случају да није, тада у низ програмских језика треба да додамо тај нови језик и да му придружимо једноелементни низ са унетом оценом.

Иако једноставан, овакав начин попуњавања низа је врло неефикасан. Наиме, са сваким уносом морамо да претражујемо постојећи низ. Претраживање морамо да вршимо обичном линеарном претрагом, јер немамо гаранцију да ће наш низ бити сортиран. Попуњавање низа на овај начин је квадратне сложености и није погодно за велике димензије улаза.

С обзиром да наш низ није сортиран, пре исписивања резултата морамо да га сортирамо лексикографски по називима програмских језика и тек након тога да испишемо садржај низа. Сортирање можемо да извршимо неким ефикасним алгоритмом у логлинеарном времену, па сложеношћу програма доминира попуњавање низа. Након сортирања, просечне оцене можемо да рачунамо добро познатим алгоритмом пре самог штампања података о конкретном језику. Дакле, ако желимо ефикасно решење, морамо ефикасније да попуњавамо наш низ приликом учитавања резултата.

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

```cpp
#include <iomanip>
#include <algorithm>

using namespace std;

int main() {

    /* učitavamo broj ispitanika */
    int n;
    cin >> n;

    /* svakom programskom jeziku pridružujemo vektor ocena,
     * pa nam zbog toga treba vektor parova oblika
     * <naziv_jezika, vektor_ocena>
     */
    vector<pair<string, vector<double>>> prog_jezici;
    /* učitavamo odgovore n ispitanika */
    for (int i = 0; i < n; i++) {
        string s;
        double o;
        cin >> s >> o;
        /* proveravamo da li smo uneti programski jezik
         * ranije već učitali
         */
        int indeks = -1;
        for (int j = 0; j < prog_jezici.size(); j++) {
            if (prog_jezici[j].first == s) {
                indeks = j;
                break;
            }
        }
        /* ako smo jezik, već ranije učitali */
        if (indeks != -1) {
            /* ocenu dodajemo u vektor ocena */
            prog_jezici[indeks].second.push_back(o);
        }
        /* inače dodajemo novi par u vektor programskih jezika */
        else {
            vector<double> vec;
            vec.push_back(o);
            prog_jezici.push_back(make_pair(s, vec));
        }
    }

    /* sortiramo vektor programskih jezika po nazivu jezika */
    sort(prog_jezici.begin(), prog_jezici.end(),
        [](pair<string, vector<double>> a, pair<string,vector<double>> b) {

        return a.first < b.first;
```

```
    });

    /* fiksiramo prikaz realnih brojava na dve decimale */
    cout << fixed;
    cout << setprecision(2);
    /* štampamo rezultate u traženom obliku */
    for (int i = 0; i < prog_jezici.size(); i++) {
        /* za svaki programski jezik, prvo računamo prosečnu ocenu */
        double sum = 0;
        for (int j = 0; j < prog_jezici[i].second.size(); j++) {
            sum += prog_jezici[i].second[j];
        }
        sum /= prog_jezici[i].second.size();
        /* zatim prikazujemo rezultate ispitivanje */
        cout << prog_jezici[i].first << " " << sum << endl;
    }

    return 0;
}
```

Из анализе неефикасног решења, јасно је да попуњавање низа треба да учинимо ефикаснијим. Поред тога, рачунање просечне оцене можемо такође да учинимо меморијски ефикаснијим. Уместо да чувамо читав низ оцена, довољно је да за сваки програмски језик одржавамо текући збир оцена које су доделили корисници, као и укупан број оцена тог језика. На крају ћемо просечну оцену лако добити као количник та два броја.

Да бисмо попуњавање низа учинили ефикаснијим потребно је да пре свега учинимо претрагу раније унетих језика ефикаснијом и да по могућству избацимо накнадно сортирање. У програмском језику Ц++ можемо да користимо колекцију *std::map* која представља речник сортиран по кључу. Као кључ у речнику ћемо користити назив програмског језика, а као вредност ћемо да користимо уређени пар чија ће прва координата бити текући збир оцена, а друга координата ће бити број оцена додељених програмском језику.

Интерно, колекција *std::map* је самобалансирајуће бинарно стабло чији је кључ у чвору управо кључ речника. Самобалансирањем се гарантују логаритамска времена за све уобичајене операције: претрага, додавање, ажурирање и брисање, што је значајно ефикасније од линеарног времена у наивном решењу. Идејно, попуњавање речника се уопште не разликује од попуњавања у наивном случају. Правилним избором структуре података којом представљамо информације у програму добили смо неупоредиво ефикасније решење. Укупна временска сложеност изградње речника у овом случају биће логлинеарна.

С обзиром да је речник сортирана колекција, да бисмо приказали резултат потребно је само да редом одштампамо елементе који су у речнику.

```
#include <iostream>
#include <map>
#include <iomanip>

using namespace std;

int main() {
```

```
    /* učitavamo broj ispitanika */
    int n;
    cin >> n;

    /* informacije o jezicima možemo odmah
     * da čuvamo u sortiranoj kolekciji
     * kao ključ koristimo naziv jezika, a kao vrednost
     * možemo da čuvamo tekući zbir ocena i broj ocena
     */
    map<string, pair<double, int>> prog_jezici;
    /* učitavamo odgovore n ispitanika */
    for (int i = 0; i < n; i++) {
        string s;
        double o;
        cin >> s >> o;
        /* proveravamo da li smo uneti programski jezik
         * ranije već učitali
         */
        auto it = prog_jezici.find(s);
        /* ako nismo */
        if (it == prog_jezici.end()) {
            /* dodajemo jezik u mapu */
            prog_jezici[s] = make_pair(o, 1);
        }
        /* inače, ažuriramo dosadašnji zbir i broj ocena */
        else {
            prog_jezici[s] = make_pair(it->second.first + o, it->second.second + 1);
        }
    }

    /* fiksiramo prikaz realnih brojava na dve decimale */
    cout << fixed;
    cout << setprecision(2);
    /* i prikazujemo rezultate */
    for (auto it = prog_jezici.begin(); it != prog_jezici.end(); it++) {
        cout << it->first << " "  << (it->second.first/it->second.second) << endl;
    }

    return 0;
}
```

## Задатак: Наелектрисање

*Аутор: Љубомир Бановић*

Сва материја у универзуму (па самим тим и у Периној соби) има свој коефицијент наелектрисања. Пера је недавно купио електрометар (уређај за мерење наелектрисања) којим је измерио наелектрисање сваког предмета у својој соби. Наелектрисање може бити позитивно, негативно

или неутрално. Пера је одлучио да укупно наелектрисање његове собе треба да буде неутрално јер се то слаже уз неутралне боје зидова собе. Занима га на колико различитих начина може да одабере предмете које ће избацити из собе тако да укупно наелектрисање собе буде неутрално. Напиши програм који за дате коефицијенте наелектрисања сваког предмета одређује тај број начина.

**Опис улаза**

Са стандардног улаза се учитава број предмета у Периној соби $n$ ($1 \leqslant n \leqslant 20$). Затим се у наредном реду учитава $n$ целих бројева одвојених размаком који представљају коефицијенте наелектрисања предмета $c_i$ ($-10^6 \leqslant c_i \leqslant 10^6$).

**Опис излаза**

На стандардни излаз исписат један број који представља на колико начина је могуће одабрати предмете које треба избацити из собе тако да наелектрисање у соби буде неутрално.

**Подзадаци**

- $n \leqslant 5$ (20 поена)
- без додатних ограничења (80 поена)

**Пример 1**

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| `5` | `6` | Могућа избацивања су: |
| `1 -2 3 0 -1` | | `Избачено [1, -2, 3, 0, -1], остаје {}` |
| | | `Избачено [1, -2, 3, _, -1], остаје {0}` |
| | | `Избачено [_, -2, 3, 0, _], остаје {1, -1}` |
| | | `Избачено [_, -2, 3, _, _], остаје {1, 0, -1}` |
| | | `Избачено [1, _, _, 0, _], остаје {-2, 3, -1}` |
| | | `Избачено [1, _, _, _, _], остаје {-2, 3, 0, -1}` |

**Пример 2**

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| `4` | `4` | Могућа избацивања су: |
| `-2 4 -2 -2` | | `Избачено [-2, 4, -2, -2], остаје {}` |
| | | `Избачено [-2, _, _, _], остаје {4, -2, -2}` |
| | | `Избачено [ _, _, -2, _], остаје {-2, 4, -2}` |
| | | `Избачено [ _, _, _, -2], остаје {-2, 4, -2}` |

## Решење

Задатак се своди на то да се одреди број подскупова предмета у Периној соби чији збир наелектрисања износи $0$. Решење користи рекурзивно набрајање да би се одредио број могућих подскупова чији збир наелектрисања износи $0$. Функција `brojPodskupovaSaZbirom0` рекурзивно пролази кроз све предмете у соби и проверава сваки могући подскуп. За сваки предмет постоје две опције: додати га у подскуп (што утиче на тренутни збир) или га изоставити. Када се достигне крај списка предмета (услов `indeks == naelektrisanja.size()`), функција проверава да ли је збир тренутног подскупа једнак нули и враћа $1$ ако јесте, односно $0$ ако није. На овај начин, функција враћа укупан број подскупова са неутралним наелектрисањем.

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

```
using namespace std;

// rekurzivna funkcija kojom se nabrajaju svi podskupovi niza naelektrisanja
// i broje oni ciji je zbir jednak 0
int brojPodskupovaSaZbirom0(const vector<int>& naelektrisanja,
                            int indeks, int zbir) {
  // generisan je kompletan podskup
  if (indeks == naelektrisanja.size()) {
    // proveravamo da li mu je zbir 0
    if (zbir == 0)
      return 1;
    else
      return 0;
  }

  return
    // trenutno naelektrisanje ukljucujemo u podskup
    brojPodskupovaSaZbirom0(naelektrisanja, indeks + 1,
                            zbir + naelektrisanja[indeks]) +
    // trenutno naelektrisanje ne ukljucujemo u podskup
    brojPodskupovaSaZbirom0(naelektrisanja, indeks + 1, zbir);
}


int brojPodskupovaSaZbirom0(const vector<int>& naelektrisanja) {
  int indeks = 0;
  int zbir = 0;
  return brojPodskupovaSaZbirom0(naelektrisanja, indeks, zbir);
}

int main() {
  int n;
  cin >> n;

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

  cout << brojPodskupovaSaZbirom0(naelektrisanja) << endl;
  return 0;
}
```

## Задатак: ПИН апликације од ПИН-а телефона

*Аутор: Филип Марић*

Студент треба да одреди четвороцифрени ПИН за нову апликацију, али није сигуран да ће га добро запамтити. Зато је смислио да тај ПИН израчуна на основу њему добро познатог ПИН-а телефона. Сабраће ПИН свог телефона, са бројем који добије када том ПИН-у замени прву и

последњу цифру и са бројем који добије када том ПИН-у замени две средишње цифре. Нови ПИН ће бити последње 4 цифре тако добијеног збира. На пример, ако је ПИН телефона 1234, сабраће бројеве 1234 + 4231 + 1324 и добиће збир 6789, који је уједно нови ПИН.

**Опис улаза**

Са стандардног улаза се уноси четвороцифрени ПИН телефона (може да има и водеће нуле).

**Опис излаза**

На стандардни излаз исписати четвороцифрени ПИН апликације (укључујући и водеће нуле ако их има).

*Напомена*: приказ водећих нула у броју `p` се у језику C++ може постићи наредбом

```
cout << setw(4) << setfill('0') << p << endl;
```

а у језику Python наредбом

```
print(str(p).zfill(4))
```

**Пример 1**

| *Улаз* | *Излаз* |
|---|---|
| `1234` | `6789` |

**Пример 2**

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| `9999` | `9997` | Ако је ПИН телефона 9999, тада ће добити збир 9999 + 9999 + 9999 = 29997, па је нови ПИН 9997. Написати програм који на основу унетог четвороцифреног ПИН-а телефона одређује нови четвороцифрени ПИН који ће се користити за апликацију. |

## Решење

## Опис главног решења

Потребно је да из унетог четвороцифреног ПИНа телефона извучемо све четири цифре. Након тога рачунамо два нова броја: један добијен заменом прве и последње цифре, а други заменом две средишње цифре. Збир оригиналног ПИН-а, првог и другог броја се рачуна, а нови ПИН за апликацију представља последње четири цифре тог збира (водећи рачуна о водећим нулама).

```
#include <iostream>
#include <iomanip>

using namespace std;

int main() {
  int pinTelefona;
  cin >> pinTelefona;

  // odredjujemo cifre PIN-a telefona
  int c0 = pinTelefona % 10;
  int c1 = (pinTelefona / 10) % 10;
  int c2 = (pinTelefona / 100) % 10;
  int c3 = (pinTelefona / 1000) % 10;
```

```
  // broj dobijen zamenom cifre jedinica i hiljada
  int zamenaSpoljasnjih = 1000*c0 + 100*c2 + 10*c1 + c3;
  // broj dobijen zamenom cifre desetica i stotica
  int zamenaUnutrasnjih = 1000*c3 + 100*c1 + 10*c2 + c0;

  // PIN aplikacije su poslednje 4 cifre zbira
  int pinAplikacije = (pinTelefona + zamenaSpoljasnjih + zamenaUnutrasnjih) % 10000;

  // ispisujemo cetvorocifreni PIN aplikacije sa eventualnim vodecim nulama
  cout << setw(4) << setfill('0') << pinAplikacije << endl;

  return 0;
}
```

# Задатак: Број јаких лозинки

*Аутор: Филип Марић*

Лозинка је јака ако има бар 8 карактера, бар једно мало, бар једно велико слово, бар једну цифру и бар један специјални карактер (карактер који није ни слово ни цифра). Напиши програм који одређује колико унетих лозинки је јако.

### Опис улаза

Са стандардног улаза се учитава број лозинки $n$ ($1 \leq n \leq 100$), а затим $n$ лозинки (свака у посебном реду).

### Опис излаза

На стандардни излаз исписати број јаких лозинки.

### Пример

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| `5`<br>`Zdravo123!`<br>`3@abC`<br>`petlja17`<br>`GEJMERI_2024`<br>`music_Shop-85` | `2` | Јаке су само прва и последња лозинка. Друга је прекратка, У трећој недостају велико слово и специјални карактер, а у четвртој недостаје мало слово. |

## Решење

Можемо дефинисати посебну функцију којом се проверава да ли је лозинка јака. Пролазимо кроз карактере дате ниске и за сваки одређујемо да ли је мало слово, велико слово, цифра или ништа од тога. Користимо четири логичке променљиве којима региструјемо да ли се појавио неки крактер одговарајуће врсте. Њих на почетку постављамо на вредност нетачно, а ако се појави одговарајући карактер, мењамо им вредност на тачно. Лозинка је јака ако и само ако након проласка кроз све карактере све промењљиве имају вредност тачно и ако је лозинка довољно дугачка (проверу дужине је могуће урадити и на почетку функције).

У главном програму учитавамо $n$ лозинки, за сваку проверавамо да ли је јака и ако јесте увећавамо вредност бројача (који је на почетку иницијализован на нулу). Након обраде свих лозинки исписујемо вредност бројача.

```
#include <iostream>
#include <cctype>

using namespace std;

bool jakaLozinka(const string& lozinka) {
  if (lozinka.size() < 8)
    return false;
  bool malo = false, veliko = false, cifra = false, specijalni = false;
  for (char c : lozinka)
    if (islower(c)) malo = true;
    else if (isupper(c)) veliko = true;
    else if (isdigit(c)) cifra = true;
    else specijalni = true;
  return malo && veliko && cifra && specijalni;
}

int main() {
  int n;
  cin >> n;
  int brojJakih = 0;
  for (int i = 0; i < n; i++) {
    string lozinka;
    cin >> lozinka;
    if (jakaLozinka(lozinka))
      brojJakih++;
  }
  cout << brojJakih << endl;
  return 0;
}
```

# Задатак: Чудна осмосмерка

*Аутор: Душан Попадић, Филип Марић*

Осмосмерка је игра у којој је дата табела димензија $n \times n$ попуњена великим словима енглеске абецеде и још $k$ речи са стране. Циљ игре је да се задате речи пронађу у табели. Реч у табели може бити у био ком од осам смерова: нагоре, надоле, улево, удесно или дијагонално на све 4 стране (ка горе лево, ка горе десно, ка доле лево и ка доле десно). Реч може почети од било ког слова у табели и не мора попунити цео ред, колону или дијагоналу.

Деда Васа је купио часопис у коме се налази чудна осмосмерка у којој је могуће да постоје речи које се не налазе у табели, чак и да се нека реч налази на више места у табели. Помозите деда Васи тако што ћете пребројати колико се задатих речи може наћи у табели поштујући правила осмосмерке.

**Опис улаза**

У првом реду стандардног улаза се налазе два природна броја, не већа од 100, раздвојена размацима: $n$ и $k$. У наредних $n$ редова се налази по једна реч од $n$ великих латиничних слова енглеске абецеде. Свака реч представља по један ред табеле. У наредних $k$ редова се налази по једна реч састављена од великих слова енглеске абецеде. За ове речи је потребно проверити да ли се налазе у табели.

*Ограничења*

У тест примерима вредним 50 поена важи да ако се реч појављује у табели, сигурно се појављује водоравно или усправно - односно у овим тест примерима није потребно проверавати дијагонале.

**Опис излаза**

У једином реду стандардног излаза исписати један природан број који представља колико се речи (од $k$ задатих) налази у табели.

**Пример 1**

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| `5 6`<br>`SMERZ`<br>`QUXZX`<br>`XNPXQ`<br>`UJKAX`<br>`XAWMR`<br>`UJKA`<br>`PERA`<br>`MUNJA`<br>`MIRA`<br>`SMER`<br>`SUPA` | `4` | У табели се налази 4 речи: UJKA (4. ред од 1. слова удесно), MUNJA (2. колона, од 1. слова надоле), SMER (1. ред од 1. слова улево) и SUPA (1. ред од 1. слова дијагонално доле десно). |

**Пример 2**

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| `4 7`<br>`AMQE`<br>`MROO`<br>`XTOP`<br>`VSAZ`<br>`STRM`<br>`KOSA`<br>`AT`<br>`TOP`<br>`KRV`<br>`ZORA`<br>`TRUBADUR` | `4` | У табели се налазе 4 речи: STRM (4. ред, од 2. слова нагоре), TOP (3. ред од 2. слова удесно), ZORA (4. ред, од 4. слова дијагонално горе лево) и AT (4. ред, од 3. слова дијагонално горе лево). |

## Решење

С обзиром на то да су улазни подаци релативно мали, у овом задатку није потребно посебно водити рачуна о сложености, већ можемо имплементирати једноставно решење. За сваку реч

пролазимо кроз целу матрицу и из сваког поља крећемо у сваком од 8 смерова и проверавамо да ли постоји поклапање са нашом речи.

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

using namespace std;

int smerR[8] = {0, 0, 1, -1, 1, -1, 1, -1};
int smerK[8] = {1, -1, 0, 0, 1, -1, -1, 1};

bool granice(int i, int j, int n, int m) {
  return 0 <= i && i < n && 0 <= j && j < m;
}

int provera(vector<string>& tabela, string& rec, int n){
  for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
      for(int s = 0; s < 8; s++){
        int x = i, y = j, r = 0;
        while(granice(x, y, n, n) && tabela[x][y] == rec[r]){
          x += smerR[s];
          y += smerK[s];
          r++;
          if(r == rec.length()) return 1;
        }
      }
    }
  }
  return 0;
}

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

  vector<string> tabela(n);

  for(int i = 0; i < n; i++)
    getline(cin, tabela[i]);

  int brReci = 0;

  for (int i = 0; i < k; i++) {
    string rec;
    getline(cin, rec);
    brReci += provera(tabela, rec, n);
```

```
  }

  cout << brReci << endl;

  return 0;
}
```

# Задатак: Астроном

*Аутори: Немања Мајски, Оīњен Тешић*

Лука је астроном аматер и страствено прати кретање небеских тела на свом телескопу. Недавно је приметио низ чудних промена у сјају једне звезде. Наиме, забележио је низ вредности који представљају промене у сјају звезде током ноћи. Свака вредност може бити позитивна (повећање сјаја), негативна (смањење сјаја) или нула (нема промене).

Међутим, Лука је приметио да се у неким деловима ноћи дешава нешто занимљиво. Током одређених периода, све промене у сјају се међусобно пониште, што значи да се сјај звезде врати на исти ниво као и на почетку тог периода. Он верује да управо ти периоди крију посебне тајне о понашању звезда.

Ваш задатак је да помогнете Луки да преброји све могуће интервале у којима се промене сјаја међусобно поништавају, тј. враћају на исту вредност као на почетку тог периода.

**Опис улаза**

Први ред стандардног улаза садржи природан број $N \leqslant 2 \cdot 10^5$ - број забележених промена у сјају звезде током ноћи.

Други ред стандардног улаза садржи $N$ размаком раздвојених целих бројева који представљају промене у сјају звезде (сви бројеви су по апсолутној вредности не већи од $10^9$).

**Опис излаза**

На стандардни излаз треба исписати тачно један цео број - број тражених интервала.

**Подзадаци**

- $N \leqslant 100$ - 20 поена;
- $N \leqslant 5000$ - 30 поена;
- без додатних ограничења - 50 поена.

**Пример 1**

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| `7`<br>`3 4 -7 1 2 -1 6` | `2` | Интервали за које ово важи су $[3, 4, -7]$ и $[4, -7, 1, 2]$. |

**Пример 2**

| *Улаз* | *Излаз* | *Објашњење* |
|---|---|---|
| `3`<br>`1 0 1` | `1` | Једини интервал за који ово важи је $[0]$. |

## Решење

Задатак је да пронађемо број интервала у низу у којима је збир елемената једнак нули. Задатак ћемо решити користећи префиксне суме. Префиксна сума на индексу $i$ представља збир свих елемената у низу од почетка до тог индекса. Ако два елемента у низу имају исту префиксну суму, онда збир елемената између њих мора бити нула. Префиксне суме се лако рачунају на следећи начин: за сваки елемент у низу, додајемо његову вредност на тренутну префиксну суму `current_sum`.

Чување броја појављивања префиксних сума у мапи/речнику: користимо мапу `prefix_sume` која прати колико пута је нека префиксна сума већ виђена. На почетку, иницијализујемо `prefix_sume[0] = 1` јер се префиксна сума $0$ подразумевано појављује једном (пре почетка низа). То нам омогућава да рачунамо интервале који почињу од првог елемента и имају збир $0$.

Бројање интервала са збиром $0$: ако се `current_sum` већ налази у мапи/речнику `prefix_sume`, то значи да постоје поднизеви који имају збир $0$ и завршавају се на тренутној позицији. Тиме повећавамо `broj_intervala` за број појављивања те префиксне суме (што је у мапи `prefix_sume[current_sum]`).

Ажурирање мапе/речника са новом појавом префиксне суме: на крају сваке итерације, повећавамо број појављивања `current_sum` у мапи `prefix_sume` за $1$.

```cpp
//Ljuba, kod sa obicnom mapom
#include <iostream>
#include <map>
#include <vector>
using namespace std;

long long broj_intervala(int n, const vector<int>& niz) {
    map<long long, long long> prefix_sume;
    prefix_sume[0] = 1; // Prefiks suma 0 se pojavljuje jednom na početku

    long long current_sum = 0;
    long long broj_intervala = 0;

    for (int x : niz) {
        current_sum += x;

        if (prefix_sume.find(current_sum) != prefix_sume.end())
            broj_intervala += prefix_sume[current_sum];

        prefix_sume[current_sum]++;
    }

    return broj_intervala;
}

int main() {
    int n;
    cin >> n;
    vector<int> niz(n);
```

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

    cout << broj_intervala(n, niz);
    return 0;
}
```