## 2.10 Замена итерације формулом

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

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

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

### 2.10.1 Аритметички и геометријски низ

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

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

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

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

### 2.10.2 Збирови степена

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

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

### 2.10.3 Комбинаторика

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

## Задатак: Број подстрингова који почињу и завршавају са 1

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

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

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

**Пример**

| *Улаз* | *Излаз* |
|---|---|
| `010001001` | `3` |

*Објашњење*

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

**Решење**

#### Анализа свих сегмената

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

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

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

#### Бројање јединица

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

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

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

## Задатак: Недостајући број

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

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

**Излаз:** На стандардни излаз исписати елемент који недостаје.

**Пример**

| *Улаз* | *Излаз* |
|---|---|
| `5`<br>`0 4 2 5 1` | `3` |

**Решење**

### Линеарна претрага

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

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

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

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

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

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

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

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

### Збир елемената

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

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

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

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

израчунати бројеве који недостају.

## Задатак: Максимални принос

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

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

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

### Пример 1

| *Улаз* | *Излаз* |
|---|---|
| `5 10 3` | `80` |

*Објашњење*

Димензија након проширења ће бити $8 \times 10$.

### Пример 2

*Улаз*

```
9 10 4
```

*Излаз*

```
132
```

*Објашњење*

Димензија након проширења ће бити $11 \times 12$.

### Пример 3

*Улаз*

```
14 17 5
```

*Излаз*

```
324
```

*Објашњење*

Димензија након проширења ће бити $18 \times 18$.

### Решење

#### Груба сила

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

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

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

### Израчунавање максималног приноса

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

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

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

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

## Задатак: Број дељивих у интервалу

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

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

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

**Излаз:** На стандардни излаз исписати тражени цео број.

### Пример

| *Улаз* | *Излаз* |
|---|---|
| 30<br>53<br>5 | 5 |

*Објашњење*

Бројеви су 30, 35, 40, 45 и 50.

### Решење

### Линеарна претрага

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

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

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

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

### Израчунавање броја дељивих бројева

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

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

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

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

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

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

## Задатак: Растављања на збир узастопних

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

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

**Излаз:** На стандардни излаз исписати тражени број начина.

### Пример

*Улаз* *Излаз*
```
15    3
```

*Објашњење*

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

### Решење

### Испробавање свих могућности за први члан, па за дужину

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

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

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

### Испробавање свих могућности за дужину, па за први члан

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

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

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

### Испробавање свих могућности за дужину и израчунавање првог члана

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

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

$m$.

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

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

## Задатак: Највећи заједнички делилац

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

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

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

**Пример**

| *Улаз* | *Излаз* |
|---|---|
| 20<br>30<br>40 | 10 |

**Решење**

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

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

### Еуклидов алгоритам

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

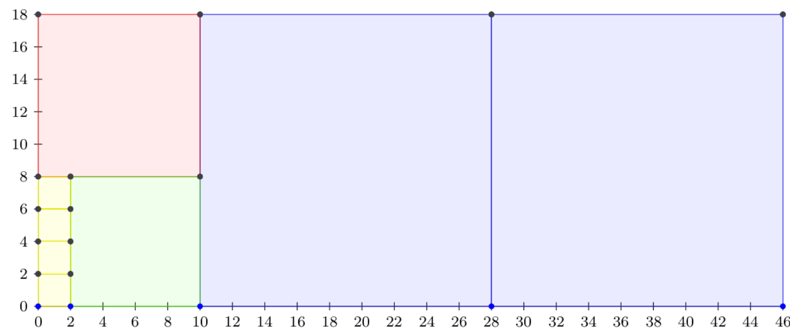
Ако су два броја једнака тј. ако је $a = b$, тада им је и њихов НЗД једнак.

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

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

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

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



Слика 2.21: Поплочавање правоугаоника квадратима

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

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

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

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

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

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

**Поправљање сложености коришћењем дељења**

**Пример.** Размотримо одређивање НЗД на једном примеру.

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

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

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

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

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

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

Еуклидов алгоритам, дакле, допушта веома једноставну рекурзивну карактеризацију.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 2.11 Одсецање

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

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

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

### Задатак: Прост број

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

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

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

**Пример**

| *Улаз* | *Излаз* |
|---|---|
| `17` | `DA` |

**Пример 2**

| *Улаз* | *Излаз* |
|---|---|
| `903543481` | `NE` |

**Решење**

#### Линеарна претрага свих потенцијалних делилаца

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

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

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

#### Одсецање у претрази

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

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

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

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

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

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

Могуће је правити и другачије имплементације истог алгоритма.

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

### Провера само непарних бројева

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

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

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

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

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

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

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

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

## Задатак: Ератостеново сито

Напиши програм који одређује број простих бројева у интервалу $[a, b]$ и њихов збир (ако збир има више од 6 цифара, исписати само 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<br>1000 | 168 76127 |

**Решење**

#### Појединачне провере простих бројева

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

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

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

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

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

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

### Ератостеново сито

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## Задатак: Најдужа серија победа

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

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

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

**Пример**

| *Улаз* | *Излаз* |
|---|---|
| 8<br>1<br>1<br>-1<br>1<br>1<br>1<br>-1<br>-1 | 3 |

**Решење**

### Груба сила

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

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

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

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

  return maxDuzina;
}
```

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

### Најдужа серија за сваки леви крај

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

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

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

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

```
  }

  return maxDuzina;
}
```

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

**Одсецање непотребних израчунавања**

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

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

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

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

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

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

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

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

## Задатак: Број растућих сегмената

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

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

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

**Пример**

| *Улаз* | *Излаз* |
|---|---|
| 5<br>1<br>3<br>4<br>-2<br>10 | 4 |

*Објашњење*

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

**Решење**

### Груба сила

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

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

### Број растућих сегмената за сваки леви крај

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

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

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

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

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

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

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

#### Максимални растући сегменти

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

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

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

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

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

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

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

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

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

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

*Види другачија решења овог задатка.*

## Задатак: Максимални збир сегмента

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

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

**Излаз:** На стандардни излаз испиши тражени збир.

**Пример**

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

**Решење**

**Груба сила**

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

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

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

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

### Одсецање непотребних провера

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

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

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

### Прекид после негативног збира

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

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

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

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

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

**Одсецање провере почетака унутар позитивног сегмента**

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

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

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

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

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

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

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

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

*Види другачија решења овог задатка.*

## 2.12 Инкременталност

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

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

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

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

### 2.12.1 Инкременталност збира и производа

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

### Задатак: Највећи збир префикса

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

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

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

**Пример**

| *Улаз* | *Излаз* |
|---|---|
| 5 | 8 |
| 4 | |
| 2 | |
| -3 | |
| 5 | |
| -4 | |

**Решење**

#### Засебно израчунавање сваког салда

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

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

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

#### Скривена сложеност

За сабирање је могуће употребити и библиотечку функцију.

```
int najveciZbirPrefiksa(const vector<int>& a) {
  // prvog dana je zbir 0
  // najveci zbir prefiksa
  int maxZbir = 0;
  // analiziramo sve pozicije u nizu
  for (auto it = begin(a); it != end(a); it++)
    // racunamo zbir prefiksa od pocetka do ispred pozicije it
    // i azuriramo maksimum ako je potrebno
    maxZbir = max(maxZbir, accumulate(begin(a), it, 0));
```

```
  return maxZbir;
}
```

**Анализа сложености.** Иако се тада у програму види једна петља, због скривене линеарне сложености библиотечких функција укупна сложеност програма остаје квадратна тј. $O(n^2)$.

**Напомена.** Скривена сложеност у задацима овог типа је нарочито чест проблем у језицима у којима се збир дела низа израчунава јако једноставно и елегантно. На пример, у језику Python:

```
maxZbir = 0
for i in range(n):
    maxZbir = max(maxZbir, sum(a[0:i+1]))
```

### Инкрементално израчунавање салда

Много ефикасније решење се може добити ако се примети прилично очигледна чињеница да се салдо у следећем дану може јако једноставно израчунати ако је познат салдо у претходном дану тако што се тај салдо увећа за износ трансакције у следећем дану, што омогућава *инкременталност* израчунавања. Збир (текући салдо) израчунавамо индуктивном конструкцијом. Означимо са $Z_k$ збир трансакција у првих $k$ дана. Важи да је $Z_0 = 0$ и да је $Z_{k+1} = Z_k + a_k$. Збир зато иницијализујемо на нулу, а онда у петљи учитавамо трансакцију по трансакцију, увећавамо тренутни збир за износ трансакције, проверавамо да ли је текући износ већи од до тада највећег и ако јесте, ажурирамо вредност највећег износа.

```
// ucitavamo broj elemenata niza
int n;
cin >> n;

// prvog dana je saldo 0
// najveci zbir prefiksa
int maxZbir = 0;
// tekuci zbir prefiksa
int zbir = 0;
for (int i = 0; i < n; i++) {
  // ucitavamo tekuci element niza
  int x; cin >> x;
  // azuriramo zbir prefiksa
  zbir += x;
  // azuriramo maksimum ako je potrebno
  if (zbir >= maxZbir)
    maxZbir = zbir;
}

// ispisujemo resenje
cout << maxZbir << endl;
```

**Анализа сложености.** У сваком кораку петље вршимо константан број додатних операција (ажурирање збира и минимума) и сложеност овог алгоритма је линеарна тј. $O(n)$. Приметимо и да је меморијска сложеност овог решења $O(1)$, јер током инкременталне обраде нема потребе памтити вредности свих трансакција у низу.

### Задатак: Сегмент датог збира у низу целих бројева

Напиши програм који за дати низ целих бројева одређује број непразних сегмената узастопних елемената низа чији је збир једнак датом броју.

**Улаз:** Са стандардног улаза се у првој линији уноси тражена вредност збира $z$ (цео број $-10000$ и $10000$), затим, у наредној линији димензија низа $n$ ($3 \leq n \leq 50000$) и затим у наредној линији елементи низа (цели бројеви између -100 и 100, раздвојени размаком).

**Излаз:** На стандардни излаз испиши број сегмената чији је збир једнак $z$.

**Пример**

| *Улаз* | *Излаз* |
|---|---|
| 11 | 7 |
| 10 | |
| 1 2 3 5 1 -1 1 5 3 2 | |

**Решење**

### Груба сила

Директно решење грубом силом подразумевало би да се провере сви сегменти узастопних елемената, да се за сваки израчуна збир и да се провери да ли је тај збир једнак траженом. Сви сегменти се могу набројати угнежђеним петљама, где спољна петља пролази кроз леве крајеве сегмента ($i$ узима вредности од $0$ па до $n-1$), а унутрашња петља пролази кроз десне крајеве сегмената (од вредности $i$ па до $n-1$). Збир можемо рачунати алгоритмом сабирања или применом библиотечких функција.

```
int brojSegmenataDatogZbira(const vector<int>& a, int trazeniZbir) {
  // broj elemenata niza
  int n = a.size();
  // broj segmenata trazenog zbira
  int broj = 0;

  // prolazimo sve segmente
  for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
      // izracunavamo zbir segmenta [i, j]
      int zbir = 0;
      for (int k = i; k <= j; k++)
        zbir += a[k];
      // proveravamo da li je zbir tog segmenta jednak trazenom
      if (zbir == trazeniZbir)
        broj++;
    }
  }

  return broj;
}
```

**Анализа сложености.** Сложеност овог приступа је кубна у односу на димензију $n$ тј. $O(n^3)$. Наиме, постоји квадратни број сегмената и за сабирање елемената сваког од њих потребно је линеарно време.

Претходна процена је извршена прилично грубо, јер иако је сложеност сабирања сегмента линеарна, нису сви сегменти дужине $n$, међутим, и прецизнија анализа ће довести до истог резултата. Унутрашња петља се извршава $j-i+1$ пута толико пута се врши операција сабирања), што одговара дужини сегмента. Спољне петље које набрајају све сегменте набрајају један сегмент дужине $n$, два сегмента дужине $n-1$, три сегмента дужине $n-2$, … и $n$ сегмената дужине $1$. Значи да је број корака једнак $1 \cdot n + 2 \cdot (n-1) + 3 \cdot (n-2) + \ldots + (n-1) \cdot 2 + n \cdot 1$. Дакле, сегмената дужине $k$ има $n-k+1$, па важи да је претходни збир једнак

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

Ово је реда величине $\frac{n^3}{2} - \frac{2n^3}{6} = \frac{n^3}{6}$, што је $O(n^3)$.

### Инкрементално рачунање збира сегмената

Алгоритам грубе силе који посебно рачуна збир сваког сегмента се може унапредити ако се збирови рачунају инкрементално тј. ако се искористи чињеница да се збир сваког сегмента који се добија проширивањем претходног сегмента једним елементом може лако израчунати на основу збира претходног сегмента, тако што се збир претходног сегмента увећа за текући елемент низа. Ту идеју смо видели, на пример, у задатку Највећи збир префикса.

Дакле, опет можемо набрајати све сегменте угнежђеним петљама, на почетку тела спољашње петље збир иницијализујемо на нулу, у унутрашњој петљи збир увећавамо за елемент $a_j$ и, ако је он једнак траженом, исписујемо индексе интервала $[i, j]$.

```
int brojSegmenataDatogZbira(const vector<int>& a, int trazeniZbir) {
  // broj elemenata niza
  int n = a.size();
  // broj segmenata trazenog zbira
  int broj = 0;

  // prolazimo sve segmente
  for (int i = 0; i < n; i++) {
    // zbir segmenta [i, j]
    int zbir = 0;
    for (int j = i; j < n; j++) {
      // izracunavamo zbir segmenta [i, j] na osnovu zbira segmenta [i, j-1]
      zbir += a[j];
      // proveravamo da li je zbir tog segmenta jednak trazenom
      if (zbir == trazeniZbir)
        broj++;
    }
  }

  return broj;
}
```

**Анализа сложености.** Овим је избегнута линеарна сложеност за израчунавање збира тренутног интервала тј. да се збир сваког наредног интервала добија у константном времену, тако да је сложеност целог алгоритма редукована на квадратну тј. $O(n^2)$.

Опет би се прецизнијом анализом добио исти резултат. Наиме, сви збирови сегмената који почињу на позицији 0 рачунају се помоћу $n$ сабирања, збирови сегмената који почињу на позицији 1 рачунају се помоћу $n-1$ сабирања итд. Укупан број сабирања је, дакле, $1 + \ldots + n = \frac{n(n+1)}{2}$, што је $O(n^2)$.

Задатак се, може решити и доста ефикасније од овога.

*Види другачија решења овог задатка.*

### Задатак: Сума реда

Другарице су кренуле на клизање. Изнајмиле су клизаљке, мало се клизале, а онда одлучиле да се окрепе уз топли чај. Изуле су се, ставиле клизаљке на гомилу, али су заборавиле да обележе које клизаљке су чије. Пошто све имају ногу веома сличне величине, договориле су се да није ни важно, него да свака може да узме било који пар клизаљки са гомиле. Израчунати колика је вероватноћа да ниједна од њих није добила исте клизаљке које је користила пре паузе за чај, ако се зна да се та вероватноћа може израчунати као $1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} \ldots + \frac{(-1)^n}{n!}$, где је $n$ број другарица.

**Улаз:** Са стандардног улаза се учитава број другарица $n$ ($2 \leq n \leq 20$).

**Излаз:** На стандардни излаз исписати тражену вероватноћу, заокружену на 14 децимала.

**Пример 1**

| *Улаз* | *Излаз* |
|---|---|
| 2 | 0.500000000000000 |

**Пример 2**

| *Улаз* | *Излаз* |
|---|---|
| 10 | 0.367879464285714 |

**Решење**

**Израчунавање сваког сабирка засебно**

Директан начин да се израчуна тражена вероватноћа је да се израчуна збир серије бројева која се добија тако што се за свако $k$ од 0 до $n$ израчуна вредност $\frac{(-1)^k}{k!}$. Степен $(-1)^k$ се може израчунати функцијом `pow` или се може одредити гранањем, пошто је $(-1)^k = 1$ за парне вредности $k$ и $(-1)^k = -1$ за непарне вредности $k$. Факторијел $k!$ се може израчунати множењем серије бројева од 1 до $k$.

```
// faktorijel broja n
double faktorijel(int n) {
  double p = 1.0;
  for (int i = 2; i <= n; i++)
    p *= i;
  return p;
}

// verovatnoca da nijedna devojcica nije uzela iste klizaljke:
// zbir 1 - 1/1! + 1/2! + ... + (-1)^n/n!
double verovatnoca(int n) {
  double p = 0.0;
  for (int k = 0; k <= n; k++)
    p += pow(-1, k) / faktorijel(k);
  return p;
}
```

**Анализа сложености.** Израчунавање $k$-тог сабирка захтева $O(k)$ операција, па је за сабирање $n$ сабирака потребно време $O(n^2)$.

**Инкрементално израчунавање сабирака**

Ефикасније решење се може добити ако се уочи да бројеви $1 = \frac{(-1)^0}{0!}$, $-1 = \frac{(-1)^1}{1!}$, $\frac{1}{2} = \frac{(-1)^2}{2!}$, $-\frac{1}{6} = \frac{(-1)^3}{3!}$ итд. чине веома правилну серију у којој се сваки наредни члан може добити инкрементално, множењем претходног члана вредношћу $-\frac{1}{k}$. Дакле, у овом задатку се користи како инкременталност серије парцијалних збирова (у склопу алгоритма сабирања), тако и инкременталност серије самих сабирака, који су заправо парцијални производи правилне серије $-\frac{1}{1}, -\frac{1}{2}, -\frac{1}{3}, -\frac{1}{4}, \ldots$ Ако са $x_k$ обележимо сабирак $k$, тада је $x_0 = \frac{(-1)^0}{0!} = 1$, док је $x_{k+1} = -\frac{1}{k} \cdot x_k$.

Током имплементације ћемо одржавати две променљиве. Прва ће да представља збир до сада сабраних сабирака, а друга текући сабирак. Збир и текући члан иницијализоваћемо на вредност 1 (то је вредност $\frac{(-1)^0}{0!}$). У сваком кораку петље у којој $k$ узима вредности од 1 до $n$ текући члан ћемо ажурирати множењем са вредношћу $-\frac{1}{k}$ и додаваћемо га на збир. Након завршетка петље, збир ће садржати тражену вероватноћу коју ћемо исписати са траженим бројем децимала.

**Напомена.** Добар савет приликом израчунавања збирова овог типа је да се израчуна количник два суседна сабирка и да се провери да ли се он можда веома једноставно израчунава у функцији од $k$ (у овом случају тај количник је $\frac{-1}{k}$). Уместо количника, могуће је користити и разлику два узастопна сабирка.

```
// verovatnoca da nijedna devojcica nije uzela iste klizaljke:
// zbir 1 - 1/1! + 1/2! + ... + (-1)^n/n!
double verovatnoca(int n) {
  double p = 1.0;
  // tekuci element zbira (-1)^k/k!
  double xk = 1.0;
  for (int k = 1; k <= n; k++) {
    // izracunavamo sledeci clan mnozenjem prethodnog sa -1/k
    xk *= -1.0/k;
    // dodajemo ga na z
    p += xk;
  }
  return p;
}
```

**Анализа сложености.** Израчунавање $k$-тог сабирка на основу претходног захтева само $O(1)$ операција, па се израчунавање збира $n$ сабирака врши у времену $O(n)$. Додуше, у овом задатку $n$ је веома мали број, па се на овај начин не постиже значајно убрзање, али у другим сличним задацима оптимизација на основу инременталности може бити веома значајна.

**Доказ коректности. Ко жели да зна више?**

На крају, образложимо и како се дошло до формуле која је у задатку дата. Пермутације у којима ни један

елемент није на свом месту се називају деранжмани. Њихов број је могуће одредити на основу формуле укључивања-искључивања. Размотримо скуп пермутација од 4 елемента. Њега чине пермутације 1234, 1243, 1324, 1342, 1423, 1432, 2134, 2143, 2314, 2341, 2413, 2431, 3124, 3142, 3214, 3241, 3412, 3421, 4123, 4132, 4213, 4231, 4312, 4231 од чега су деранжмани 2143, 2341, 2413, 3142, 3412, 3421, 4123, 4312 и 4321. Укупно пермутација има 4!. Од тог броја треба одузети број пермутација којима се бар један елемент налази на свом месту. Можемо разматрати пермутације у којима се 1 налази на свом месту, а остали бројеви су произвољно испермутовани (то су 1234, 1243, 1324, 1342, 1423, 1432), у којима се број 2 налази на свом месту, а остали бројеви су произвољно испермутовани (то су 1234, 1243, 3214, 3241, 4213, 4231), у којима се број 3 налази на свом месту, а остали бројеви су произвољно испермутовани (то су 1234, 1432, 2134, 2431, 4132, 4231) и оне у којима је 4 на свом месту, а остали бројеви су произвољно испермутовани (то су 1234, 1324, 2134, 2314, 3124, 2314). Њих има $4 \cdot 3!$. Преостале пермутације су деранжмани и број деранжмана се може добити тако што се од укупног броја пермутација одузме број оних пермутација које фиксирају неки елемент. Проблем настаје због тога што су неке пермутације бројане више пута (на пример, пермутација 1243 је бројана и у оквиру пермутација које елемент 1 остављају на свом месту и у оквиру пермутација које остављају број 2 на свом месту). Пермутације које су бројане бар два пута су оне које остављају нека два елемента на свом месту. Ако су то елементи 1 и 2, то су пермутације 1234 и 1243, ако су то елементи 1 и 3 то су пермутације 1234 и 1432, ако су то елементи 1 и 4 то су пермутације 1234 и 1324, ако су то елементи 2 и 3 то су пермутације 1234 и 4231, ако су то елементи 2 и 4 то су пермутације 1234 и ако су то елементи 3 и 4 то су пермутације 1234 и 2134. Парова елемената има $\binom{4}{2} = 6$, а сваки од њих даје две пермутације. Њихов број можемо одузети од броја пермутација са фиксираним једним елементом, али је проблем да се међу њима неке пермутације понављају тако да смо онда одузели више него што је потребно. Пермутације које се понављају више пута су оне које фиксирају 3 елемента. Која год да су три елемента у питању то је пермутација 1234. Три елемента можемо одабрати на $\binom{4}{3} = 4$ начина тако да треба додати 4 пермутације. Међутим, тада је пермутација која фиксира сва 4 елемента урачуната два пута, тако да њу на крају треба одузети. Дакле број деранжмана једнак је $4! - 4 \cdot 3! + 6 \cdot 2! - 4 \cdot 1! + 1 = 24 - 24 + 12 - 4 + 1 = 9$.

Овом логиком долазимо до тога да се број деранжмана може израчунати као

$$n! - \binom{n}{1}(n-1)! + \binom{n}{2}(n-2)! - \ldots + (-1)^{n-1}\binom{n}{n-1}1! + (-1)^n$$

Зато је вероватноћа да се насумично одабере деранжман једнака количнику броја деранжмана и укупног броја пермутација. Пошто је $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, добијамо

$$\frac{n! - \frac{n!}{1!(n-1)!}(n-1)! + \frac{n!}{2!(n-2)!}(n-2)! + \ldots + (-1)^{n-1}\frac{n!}{(n-1)!1!}1! + (-1)^n}{n!}$$

тј.

$1 - \frac{1}{1!} + \frac{1}{2!} - \ldots + \frac{(-1)^{n-1}}{(n-1)!} + \frac{(-1)^n}{n!}$

### 2.12.2 Инкременталност минимума и максимума

И минимум и максимум серије бројева представљају веома важне и често коришћене статистике. Минимум тј. максимум серије префикса тј. серије суфикса низа такође могу да се израчунавају инкрементално, што доводи до ефикаснијих решења.

#### Задатак: Најбољи "сабмит"

Такмичар је током интергалактичког шампионата у програмирању слао на оцењивање један задатак више пута. Број поена које такмичар добија за задатак се рачуна тако што се одреди највећи број поена од свих појединачних слања (гледа се "најбољи сабмит"). Напиши програм који одређује колико је поена за тај задатак ученик имао након сваког слања. Пошто су интергалактички задаци веома тешки, они носе пуно поена и такмичари их често шаљу велики број пута.

**Улаз:** У првој линији стандардног улаза налази се природан број $n$ ($n \leq 50000$). У следећих $n$ линија налазе се редом поени које је ученик добио за свако појединачно слање (број између 0 и 100000).

**Излаз:** На стандардном излазу приказати $n$ линија које приказују колико је поена за тај задатак ученик имао након сваког слања.

**Пример**

| *Улаз* | *Излаз* |
|---|---|
| 5 | 3 |
| 3 | 3 |
| 2 | 4 |
| 4 | 4 |
| 1 | 5 |
| 5 | |

**Решење**

За сваку позицију $i$ у серији бројева је потребно одредити највећи међу свим бројевима од почетка серије, закључно са тренутним елементом $m_i = \max(a_0, \ldots, a_i)$.

### Груба сила

Наивни начин да се то уради је да се резултати свих слања упишу у низ, а затим да се за сваку позицију $i$ од $0$ до $n-1$ одреди и испише максимум $m_i$. Одређивање максимума се може урадити уобичајеним алгоритмом одређивања максимума серије бројева.

**Анализа сложености.** Ово решење је прилично неефикасно (има квадратну сложеност у односу на број слања, тј. сложеност $O(n^2)$).

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

for (int i = 0; i < n; i++) {
  int maxPoena = poeni[0];
  for (int j = 1; j <= i; j++)
    if (poeni[j] > maxPoena)
      maxPoena = poeni[j];

  cout << maxPoena << endl;
}
```

### Библиотечке функције - скривена сложеност

Максимум дела низа можемо одредити и библиотечком функцијом. У језику C++ можемо употребити функцију `max_element`.

**Анализа сложености.** Иако сада програм има само једну петљу, у функцији за израчунавање максимума је скривена линеарна сложеност, па укупна је сложеност $O(n^2)$.

```
for (auto it = next(begin(poeni)); it <= end(poeni); it++)
  cout << *max_element(begin(poeni), it) << endl;
```

### Инкременталност серије парцијалних максимума

Кључни увид којим се може доћи до ефикаснијег решења је да се приликом додавања новог елемента максимум проширене серије $m_{i+1}$ може израчунати крајње једноставно ако је познат максимум $m_i$ серије пре проширења. Наиме, ако је познат број поена такмичара пре текућег слања задатка, када се одреди број поена у том слању, максимум се или увећава (ако је у текућем слању остварен највећи број поена до тада) или се не мења тј. $m_{i+1} = \max(m_i, a_{i+1})$. Почетни максимум може бити постављен или на први елемент низа (важи да је $m_0 = a_0$), или, пошто су сви елементи ненегативни, на нулу (важи да је $m_{-1} = 0$). Дакле, принцип инкременталности који важи за парцијалне збирове, важи и за парцијалне максимуме. Инкрементално израчунавање серије парцијалних збирова (збирова префикса) описано је, на пример, у задатку Највећи збир префикса.

Дакле, у задатку се примењује класичан алгоритам одређивања максимума серије бројева, али се у сваком кораку исписује текућа вредност максимума.

**Анализа сложености.** Иницијализација се врши у времену $O(1)$ и сваки нови максимум се од претходног добија у времену $O(1)$, па се $n$ максимума израчунава у укупном времену $O(n)$. Ово решење не учитава

истовремено све елементе у низ, па је меморијска сложеност $O(1)$. Са друге стране у овом решењу се наизменично учитавају и исписују бројеви, што у језику C++ захтева да се библиотека за улаз и излаз посебно подеси, да би се искључило стално пражњење бафера, које успорава програм (довољно је навести `cin.tie(0)` и уместо `endl` користити `'\n'`.

```
ios_base::sync_with_stdio(false); cin.tie(0);
int n;
cin >> n;
int maxPoena = 0;
for (int i = 0; i < n; i++) {
  int poeni;
  cin >> poeni;
  if (poeni > maxPoena)
    maxPoena = poeni;
  cout << maxPoena << '\n';
}
```

### Задатак: Поглед на реку

У једној улици која иде ка реци налазе се разне куће и зграде. Инвеститор бира локацију на којој би изградио вишеспратницу, такву да се са њеног последњег спрата види река. Зато она мора бити виша или бар једнаке висине од свих постојећих зграда од одабране локације до краја улице. Напиши програм који за сваку локацију у тој улици одређује минималну висину нове вишеспратнице.

**Улаз:** У првој линији стандардног улаза налази се природан број $n$ ($n \leq 50000$). У следећих $n$ линија налазе се редом висине свих зграда и кућа од почетка до краја улице.

**Излаз:** На стандардном излазу приказати $n$ бројева (сваки у посебном реду) који приказују тражене висине.

**Пример**

| *Улаз* | *Излаз* |
|---|---|
| 5 | 23 |
| 13 | 23 |
| 23 | 17 |
| 11 | 17 |
| 17 | 13 |
| 13 | |

**Решење**

Задатак захтева да се за сваку позицију $i$ у низу одреди максимум свих елемената од те позиције па до краја низа (тј. да се одреди $m_i = \max(a_i, \ldots, a_{n-1})$).

Приметимо да се овде захтева рачунање максимума серије суфикса низа, што је заправо веома слично израчунавању максимуми серије префикса низа. Израчунавање серије максимума префикса низа описано је у задатку Најбољи “сабмит”.

**Груба сила**

Директан, наивни приступ да се то уради је да се у спољној петљи по $i$ пролази кроз позиције од $0$ до $n-1$, а да се у унутрашњој петљи која пролази позиције од $i$ до $n-1$ одређује максимум $m_i$ (применом уобичајеног алгоритма одређивања максимума тј. минимума серије елемената). Максимуме ћемо уписивати у низ (то може бити и оригинални низ) и на крају исписати у траженом редоследу.

**Анализа сложености.** Пошто се за сваки суфикс дужине од $1$ до $n$ одређује максимум у линеарној сложености, укупна сложеност овог приступа је $O(n^2)$.

**Инкременталност**

И за серију максимума растућих суфикса низа важи принцип инкременталности. Максимум елемената који почињу на позицији $i$ може се лако одредити ако се већ зна максимум елемената који почињу на позицији $i+1$. Наиме, важи да је $m_i = \max(a_i, m_{i+1})$. Иницијализацију можемо извршити било на последњи елемент серије (важи $m_{n-1} = a_{n-1}$), било на нулу, јер су сви елементи ненегативни (важи $m_n = 0$).

Задатак решавамо тако што у петљи пролазимо кроз низ у обрнутом редоследу, крећући се од последњег члана низа (који је индексиран са $n-1$) до првог члана (који је индексиран са 0). Максимум иницијализујемо на нулу. У сваком пролазу кроз петљу, анализира се вредност текућег елемента низа који се пореди са текућим максимумом. Ако је вредност текућег елемента низа већа од текућег максимума (који у ствари представља максималану вредност елемената десно од текућег), нови текући максимум постаје баш тај члан низа. Један проблем са овим решењем је што се максимуми одређују у супротном редоследу од оног који је потребно исписати. Зато је пре исписа потребно упамтити их. Једно решење би било да се они памте у новом низу. Ако не желимо да ангажујемо додатну меморију, можемо приметити да се након одређивања максимума $m_i$ елемент $a_i$ неће више анализирати, па се $m_i$ може уписати на место елемента $a_i$. На крају, исписаћемо елементе низа у који смо сместили максимуме од почетка.

```
// maksimalni element od pozicije i do kraja niza
int max = 0;
for (int i = n - 1; i >= 0; i--) {
  // ako je tekuci element veci od maksimalnog iza njega, onda je
  // potrebno azurirati maksimum
  if (a[i] > max)
    max = a[i];
  // belezimo vrednost maksimuma (koristimo slobodno prostor niza a,
  // jer nam vrednost a[i] vise nece biti potrebna)
  a[i] = max;
}
```

**Анализа сложености.** Пошто се сада сваки максимум одређује на основу претходног у константној сложености, укупна сложеност одређивања свих $n$ максимума је линеарна тј. $O(n)$.

**Напомена.** Приметимо да и збирови суфикса имају својство инкременталности.

## 2.12.3 Инкременталност - остале статистике

### Задатак: Рутер

Дуж једне улице су равномерно распоређене зграде (растојање између сваке две суседне је једнако). За сваку зграду је познат број корисника које нови добављач интернета треба да повеже. Одредити у коју од зграда треба поставити рутер тако да би укупна дужина оптичких каблова којим се сваки од корисника повезује са рутером била минимална (рачунати само дужину каблова од зграде до зграде и занемарити дужине унутар зграда).

**Улаз:** У првом реду стандардног улаза налази се број $n$ ($1 \leq n \leq 10^5$), а у наредном $n$ природних бројева раздвојених размацима који представљају број корисника у свакој од $n$ зграда.

**Излаз:** На стандардни излаз исписати минималну дужину каблова.

**Пример**

| *Улаз* | *Излаз* |
|---|---|
| 6<br>3 5 1 6 2 4 | 30 |

**Решење**

**Груба сила**

Наивно решење би подразумевало да се израчуна дужина каблова за сваку могућу позицију рутера и да се одабере најмањи. Да бисмо израчунали дужину каблова, ако је рутер у згради на позицији $k$, рачунамо заправо збир

$$\sum_{i=0}^{n-1} |k-i| \cdot a_i,$$

где је $a_i$ број корисника у згради $i$.

```
long long minDuzinaKablova(const vector<int>& broj_stanara) {
  // broj zgrada
  int n = broj_stanara.size();
```

```
  // minimalna duzina kablova
  long long min_duzina_kablova = numeric_limits<long long>::max();
  // obrađujemo sve zgrade od 1 do n-1
  for (int k = 0; k < n; k++) {
    // duzina kablova ako je ruter u zgradi broj k
    long long duzina_kablova = 0;
    for (int i = 0; i < k; i++)
      duzina_kablova += (k - i) * broj_stanara[i];
    for (int i = k+1; i < n; i++)
      duzina_kablova += (i - k) * broj_stanara[i];

    if (duzina_kablova < min_duzina_kablova)
      min_duzina_kablova = duzina_kablova;
  }

  return min_duzina_kablova;
}
```

**Анализа сложености.** Сваки тежински збир можемо израчунати у времену $O(n)$, па пошто се испитује $n$ позиција, алгоритам је сложености $O(n^2)$.

### Решење на основу принципа инкременталности

Много боље решење и линеарни алгоритам можемо добити ако применимо принцип инкременталности и избегнемо рачунање у сваком кораку из почетка. Размотримо како се дужина каблова мења када се рутер помера са зграде $k$ на зграду $k+1$.

Дужину каблова за рутер у згради $k+1$ добијамо од дужине каблова за рутер у згради $k$ тако што ту дужину увећамо за укупан број станара закључно са зградом $k$ и умањимо је за укупан број станара почевши од зграде $k+1$. То је заправо интуитивно прилично јасно и без компликованог математичког извођења. Померањем рутера за дужину једне зграде надесно, сваком станару који живи закључно до зграде $k$ дужина кабла се повећала за једно растојање између зграда, а свим станарима од зграде $k+1$ надесно се та дужина смањује за једно растојање између зграда.

**Доказ коректности.** Формално, математички, то се може показати на следећи начин. Ако је рутер на позицији $k$, тада је дужина каблова једнака

$$d_k = \sum_{i=0}^{k-1}(k-i)\cdot a_i + \sum_{i=k+1}^{n-1}(i-k)\cdot a_i.$$

Ако је рутер на позицији $k+1$, тада је дужина каблова једнака

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

Разлика између те две суме једнака је

$$\begin{aligned}
d_{k+1}-d_k &= \left(\sum_{i=0}^{k}(k+1-i)\cdot a_i - \sum_{i=0}^{k-1}(k-i)\cdot a_i\right) + \left(\sum_{i=k+2}^{n-1}(i-k-1)\cdot a_i - \sum_{i=k+1}^{n-1}(i-k)\cdot a_i\right)\\
&= \left(\sum_{i=0}^{k-1}((k+1-i)-(k-i))\cdot a_i\right) + a_k - a_{k+1} + \left(\sum_{i=k+2}^{n-1}((i-k-1)-(i-k))\cdot a_i\right)\\
&= \sum_{i=0}^{k-1} a_i + a_k - a_{k-1} - \sum_{i=k+2}^{n-1} a_i\\
&= \sum_{i=0}^{k} a_i - \sum_{i=k+1}^{n-1} a_i
\end{aligned}$$

Укупне бројеве станара пре и после дате зграде можемо такође рачунати инкрементално (при преласку на наредну зграду, први број се увећава, а други умањује за број станара текуће зграде).

Дакле, у програму можемо да памтимо три ствари: дужину каблова $d_k$ ако је рутер на позицији $k$, укупан број станара $pre_k$ пре зграде $k$ (не укључујући њу) и укупан број станара $posle_k$ од зграде $k$ (укључујући њу) до краја. На почетку, када је $k = 0$, први број $d_0$ морамо експлицитно израчунати као $\sum_{i=1}^{n-1} i \cdot a_i$, други број треба иницијализовати на нулу $pre_0 = 0$, а трећи на укупан број свих станара $posle_k = \sum_{i=0}^{n-1} a_i$. Затим за свако $k$ од $1$ до $n-1$ рачунамо $pre_k = pre_{k-1} + a_{k-1}$, $posle_k = posle_{k-1} - a_{k-1}$ и затим $d_k = d_{k-1} + pre_k - posle_k$.

**Пример.** Илуструјмо извршавање овог алгоритма на примеру зграда у којима живи редом $3, 5, 1, 6, 2, 4$ станара.

```
k    dk  pre_k   posle_k
0    53   0       21
1    38   3       18
2    33   8       13
3    30   9       12
4    39  15        6
5    52  17        4
```

Приметимо да је могуће извршити и малу оптимизацију (додуше која неће поправити асимптотску сложеност) на основу монотоности низа $d_k$ и петљу прекинути чим се број $d_k$ први пут повећа. Наиме, вредности у том низу ће опадати до тражене минималне вредности, након чега ће кренути да расту. У претходном примеру, могли смо закључити да је минимална дужина каблова 30, чим се у наредном кораку та вредност повећала на 39.

```
long long minDuzinaKablova(const vector<int>& broj_stanara) {
  // broj zgrada
  int n = broj_stanara.size();
  // krećemo od zgrade 0
  // ukupna dužina kablova ako je ruter u tekućoj zgradi
  long long duzina_kablova = 0;
  for (int i = 0; i < n; i++)
    duzina_kablova += broj_stanara[i] * i;
  // broj stanara pre tekuće zgrade
  long long stanara_pre = 0;
  // broj stanara od tekuće zgrade do kraja
  long long stanara_posle = 0;
  for (int i = 0; i < n; i++)
    stanara_posle += broj_stanara[i];

  // minimalna duzina kablova
  long long min_duzina_kablova = duzina_kablova;

  // obrađujemo sve zgrade od 1 do n-1
  for (int k = 1; k < n; k++) {
    // ažuriramo brojeve stanara
    stanara_pre += broj_stanara[k-1];
    stanara_posle -= broj_stanara[k-1];
    // ažuriramo duz
    duzina_kablova += stanara_pre - stanara_posle;
    if (duzina_kablova < min_duzina_kablova)
      min_duzina_kablova = duzina_kablova;
  }

  return min_duzina_kablova;
}
```

**Анализа сложености.** Пошто је и за једну и за другу фазу потребно време $O(n)$, то је уједно сложеност овог алгоритма.

Интересантно, укупну почетну дужину каблова можемо израчунати ефикасно и без множења, тако што на збир додамо број станара у последњој згради, затим број станара у последње две зграде, затим број станара у последње три зграде и тако даље, све док не додамо број станара у свим зградама осим прве.

### Задатак: Највећи тежински збир после цикличног померања

Дат је низ $a$ целих бројева дужине $n$. Дозвољена је операција цикличног померања тј. ротације низа улево за једно место, операцију можемо понављати произвољан број пута. Написати програм којим се највећа вредност тежинског збира

$$0 \cdot a_0 + 1 \cdot a_1 + 2 \cdot a_2 + 3 \cdot a_3 + \ldots + (n-1) \cdot a_{n-1},$$

по модулу 1234567 у трансформисаном низу.

**Улаз:** У првој линији стандардног улаза налази се природан број $n$ ($1 \leq n \leq 50000$). У следећих $n$ линија налазе се редом елементи низа $a$ (цели бројеви из интервала $[0, 100]$).

**Излаз:** На стандардном излазу у једној линији приказати највећу вредност тежинског збира.

**Пример**

| *Улаз* | *Излаз* |
|---|---|
| 3<br>5<br>4<br>1 | 13 |

*Објашњење*

Највећи збир се добија након ротације за два места улево (низ је тада `154`).

**Решење**

**Груба сила**

Задатак можемо решити тако што $n-1$ пут низ ефективно ротирамо за по једно место улево, израчунавајући сваки пут тежински збир изнова, тражећи максимум и позицију максимума тако добијених тежинских збирова. Пошто се тежински збирови у нашем задатку рачунају по датом модулу $mod$, све операције у претходним формулама треба заменити одговарајућим операцијама по модулу (њих можемо реализовати у засебним функцијама).

**Анализа сложености.** Пошто број операција потребан за израчунавање тежинског збира линеарно зависи од дужине низа $n$ (сваки тежински збир се израчунава у сложености $O(n)$), овај приступ доводи до квадратне сложености алгоритма (сложености $O(n^2)$). Чак иако бисмо оптимизовали број израчунавања остатака, програм ће бити неефикасан услед квадратне сложености алгоритма и пуно померања елемената низа.

```
// modul dat u tekstu zadatka
const int mod = 1234567;

// zbir brojeva x i y po modulu mod
int zm(int x, int y, int mod) {
  return (x % mod + y % mod) % mod;
}

// ciklicno pomeranje (rotiranje) niza za jedno mesto ulevo
void rotirajUlevo(vector<int>& a, int n) {
  int pom = a[0];
  for (int i = 0; i < n - 1; i++)
    a[i] = a[i + 1];
  a[n - 1] = pom;
  // moglo bi i rotate(begin(a), next(begin(a)), end(a));
}

// suma brojeva a[i]*i, za i od 0 do n-1, po modulu mod
```

```
int tezinskiZbir(const vector<int>& a, int n, int mod) {
  int s = 0;
  for (int i = 0; i < n; i++)
    s = zm(s, i * a[i], mod);
  return s;
}

// najveci tezinski zbir 0*a[0] + ... + (n-1)*a[n-1] posle ciklicnog
// pomeranja niza ulevo
int maksTezinskiZbirNakonRotacije(const vector<int>& a) {
  // kopiramo niz da bismo mogli da ga menjamo
  auto acopy = a;
  // broj elemenata niza
  int n = acopy.size();
  // maksimum inicijalizujemo na tezinsku sumu pocetnog niza
  int maxTezinskiZbir = tezinskiZbir(acopy, n, mod);
  for (int i = 1; i < n; i++) {
    // rotiramo elemente niza a za jedno mesto ulevo
    rotirajUlevo(acopy, n);
    // izracunavamo novu tezinsku sumu
    int tekSuma = tezinskiZbir(acopy, n, mod);
    // azuriramo podatke o maksumumu, ako je to potrebno
    if (tekSuma > maxTezinskiZbir)
      maxTezinskiZbir = tekSuma;
  }
  return maxTezinskiZbir;
}
```

**Избегавање ротација низа**

Уместо ефективне ротације свих елемената низа, ефекат обиласка низа који је ротиран за $k$ места улево можемо постићи тако што обилазак крећемо од позиције $k$, а затим у петљи која има $n$ итерација увећавамо бројач за 1, али овај пут по модулу $n$ (када бројач постане $n$ вредност му се враћа на нулу што можемо постићи било експлицитним испитивањем вредности након сваког увећања бројача, било израчунавањем остатка при дељењу са $n$, што је спорије од гранања, јер је израчунавање остатка при дељењу обично релативно скупа операција).

На основу ограничења датих у тексту задатка, максимална вредност тежинског збира се постиже када низ има 50000 елемената чија је вредност 100. Тада се са 100 множи сваки елемент од 0 до 49999 и максимални тежински збир је једнак 100 пута вредност збира свих природних бројева до 49999 што је једанко $100 \cdot \frac{49999 \cdot (49999+1)}{2}$, што је око $1.25 \cdot 10^{11}$ и стаје у опсег 64-битног типа (у језику C++ то је тип `long long`). Ако тај тип употребимо за чување збира, онда остатак при дељењу можемо израчунати само једном, након целокупног израчунавања збира, чиме се добија на ефикасности.

**Анализа сложености.** Иако избегавамо ротације, свака од $n$ тежинских сума се израчунава засебно у времену $O(n)$, па сложеност остаје $O(n^2)$.

```
// modul dat u tekstu zadatka
const int mod = 1234567;

// najveci tezinski zbir 0*a[0] + ... + (n-1)*a[n-1] posle ciklicnog
// pomeranja niza ulevo
int maksTezinskiZbirNakonRotacije(const vector<int>& a) {
  int n = a.size();
  long long maxTezinskiZbir = 0;
  for (int i = 0; i < n; i++) {
    // koristimo tip long long, da ne bismo morali da izracunavamo
    // ostatak posle svakog sabiranja, vec samo nakon izracunavanja
    // celog zbira
    long long tezinskiZbir = 0;
    int k = i;
```

```
    for (int j = 0; j < n; j++) {
      tezinskiZbir += a[k] * j;
      if (++k == n) k = 0;
    }
    tezinskiZbir %= mod;
    // azuriramo podatke o maksumumu, ako je to potrebno
    if (tezinskiZbir > maxTezinskiZbir)
      maxTezinskiZbir = tezinskiZbir;
  }
  return maxTezinskiZbir;
}
```

**Оптимизација на основу инкременталности**

Задатак је могуће решити и много ефикасније ако применимо принцип инкременталности тј. пронађемо начин да тежински збир после ротације ефикасно израчунамо на основу познатог тежинског збира пре ротације. Можемо уочити да је у тежинском збиру после померања улево сваки елемент низа, изузев првог елемента $a_0$, један пут мање укључен него пре померања, а први елемент је укључен $n-1$ пут. Обележимо са $z_i$ тежински збир добијен приликом померања полазног низа улево $i$ пута, а са $z$ класичан збир свих елемената низа. Према томе важе следеће једнакости:

$$\begin{array}{rcl} z_0 & = & 0 \cdot a_0 + 1 \cdot a_1 + 2 \cdot a_2 + ... + (n-2) \cdot a_{n-2} + (n-1) \cdot a_{n-1} \\ z_1 & = & 0 \cdot a_1 + 1 \cdot a_2 + 2 \cdot a_3 + ... + (n-2) \cdot a_{n-1} + (n-1) \cdot a_0 \\ z_2 & = & 0 \cdot a_2 + 1 \cdot a_3 + 2 \cdot a_4 + ... + (n-2) \cdot a_0 + (n-1) \cdot a_1 \\ & \ldots & \\ z_{n-1} & = & 0 \cdot a_{n-1} + 1 \cdot a_0 + 2 \cdot a_1 + ... + (n-2) \cdot a_{n-3} + (n-1) \cdot a_{n-2} \end{array}$$

$$z = a_0 + a_1 + ... + a_{n-2} + a_{n-1}$$

Приметимо да важи

$$\begin{array}{rcl} z_0 - z_1 & = & a_1 + a_2 + \ldots + a_{n-1} - (n-1) \cdot a_0 = z - n \cdot a_0 \\ z_1 - z_2 & = & a_2 + a_3 + \ldots + a_0 - (n-1) \cdot a_1 = z - n \cdot a_1 \\ & \ldots & \\ z_{n-2} - z_{n-1} & = & a_{n-1} + a_0 + \ldots + a_{n-3} - (n-1) \cdot a_{n-2} = z - n \cdot a_{n-2} \end{array}$$

итд.

Према томе $z_{i-1} - z_i = z - n \cdot a_{i-1}$, тј.

$$z_0 = z, \qquad z_i = z_{i-1} - z + n \cdot a_{i-1}, \text{ за } i > 0.$$

Дакле, тежински збир после померања за једно место улево можемо једноставно израчунати без померања низа на основу тежинског збира низа пре померања и збира свих елемената низа.

Имплементацију можемо извршити на следећи начин. Израчунамо тежински збир полазног низа $z_0 = \sum_{i=0}^{n-1} i \cdot a_i$ и класични збир свих елемената низа $z = \sum_{i=0}^{n-1} a_i$ (једноставним алгоритмом сабирања елемената низа). Од свих збирова треба израчунати највећи и одредити после колико ротација се та највећи збир постиже. Максимум ћемо иницијализовати на први израчунати тежински збир, а број померања ћемо иницијализовати на 0. Затим рачунамо тежинске збирове за низове добијене померањем низа за једно место улево $i$ пута, и то редом за $i$ од 1 до $n-1$. Тежински збир $z_i$ за низ добијен након $i$ померања рачунамо тако што претходни тежински збир $z_{i-1}$ умањимо за збир свих елемената $z$ и увећамо за $n \cdot a_{i-1}$. Проверавамо да ли је добијени тражени збир већи од дотадашњег максимума и ако јесте коригујемо максимум.

Наравно, све аритметичке операције вршимо по датом модулу.

**Анализа сложености.** Приметимо да време потребно за израчунавање почетних збирова линеарно зависи од дужине низа $n$, док је за израчунавање сваког наредног збира довољан константан број операција, тако да је укупна временска сложеност алгоритма линеарна тј. $O(n)$.

```
// zbir x i y po modulu mod
int zm(int x, int y, int mod) {
  return (x % mod + y % mod) % mod;
}

// razlika x i y po modulu mod
int rm(int x, int y, int mod) {
  return (x % mod - y % mod + mod) % mod;
}

// najveci tezinski zbir posle ciklicnog pomeranja
int maksTezinskiZbirNakonRotacije(const vector<int>& a) {
  // broj elemenata niza
  int n = a.size();
  // izracunavamo tezinsku sumu i*a[i] i obicnu sumu elemenata a[i] po
  // modulu mod
  int tezinskiZbir = 0;
  int suma = 0;
  for (int i = 0; i < n; i++) {
    tezinskiZbir = zm(tezinskiZbir, i * a[i], mod);
    suma = zm(suma, a[i], mod);
  }

  // najveca do sada vidjena tezinska suma
  int maxTezinskiZbir = tezinskiZbir;
  for (int i = 1; i < n; i++) {
    // rotacija za jedno mesto ulevo na sledeci nacin menja tezinsku sumu
    tezinskiZbir = zm(rm(tezinskiZbir, suma, mod), n * a[i - 1], mod);

    // proveravamo da li je dobijena tezinska suma veca od do tada
    // najvece
    if (tezinskiZbir > maxTezinskiZbir)
      maxTezinskiZbir = tezinskiZbir;
  }
  return maxTezinskiZbir;
}
```

### 2.12.4 Покретни прозор

У многим ситуацијама потребно је израчунати одређену статистику свих сегмената низа фиксиране дужине. Тих сегмената је мање него свих сегмената (њихов број линеарно зависи од дужине низа), али ако се статистика сваког сегмента рачуна из почетка, добија се неефикасан алгоритам (квадратне сложености). Сваки наредни сегмент се разликује од претходног за тачно два елемента: приликом преласка са неког сегмента на наредни уклања се први елемент старог сегмента и додаје се последњи елемент новог сегмента. Ово омогућава инкрементално и ефикасно израчунавање многих статистика (оних заснованих на операцијама које имају инверзне, попут сабирања).

#### Задатак: Сегмент највећег просека

Дат је низ $a$ реалних бројева дужине $n$ и природан број $k$. Написати програм којим се у низу $a$ одређује позиција почетка сегмента (подниза узастопних елемената) дужине $k$ са највећим просеком (ако више сегмената има исти просек, пријавити последњи од њих).

**Улаз:** У првој линији стандардног улаза налази се природан број $k$ ($k \leq 5 \cdot 10^3$). У другој линији налази се природан број $n$ ($n \leq 5 \cdot 10^5$). У следећих $n$ линија налазе се по један реалан број (ти бројеви представљају редом елементе низа $a$).

**Излаз:** На стандардном излазу приказати позицију почетка последњег сегмента дужине $k$ низа $a$ чији је просек највећи (позиције у низу се броје од нуле).

**Пример**

| *Улаз* | *Излаз* |
|---|---|
| 3 | 2 |
| 5 | |
| 1.0 | |
| 5.0 | |
| 8.0 | |
| 2.0 | |
| 7.0 | |

**Решење**

Приметимо да је проблем налажења сегмента дужине $k$ чији је просек највећи, еквивалентан проблему налажења сегмента дужине $k$ највећег збира. Просек сегмента добијамо дељењем суме сегмента са дужином сегмента, која је у овом задатку константа и износи $k$, тако да је просек највећи када је збир највећи.

### Груба сила

Задатак решавамо тако што анализирамо све сегменте дужине $k$, почев од сегмента који почиње од елемента $a_0$ до сегмента који почиње од елемента $a_{n-k}$. Одређујемо збир сваког сегмента и уврђујемо који је последњи сегмент максималног збира. Одређивање позиције на којој почиње последњи сегмент максималног збира вршимо уобичајеним алгоритмом одређивања позиције максимума тј. минимума, при чему, пошто се тражи последњи сегмент, индекс максимума ажурирамо када год се наиђе на сегмент који има збир који је већи или једнак тренутном максимума (када би се тражио први сегмент, индекс би се ажурирао само када се наиђе на сегмент који има збир који је строго већи од тренутно максималног).

Директан приступ би био да се рачунање збира елемената сегмента који почиње од елемента $a_i$ реализује тако што ће се редом сабирати елементи $a_j$ за $j$ од $i$ до $i + k - 1$ (применом класичног алгоритма сабирања елемената низа).

```
// pronalazi indeks pocetka segmenta duzine k ciji je prosek najveci
int pocetakSegmentaNajvecegProseka(const vector<double>& a, int k) {
  // duzina niza
  int n = a.size();

  // trenutna maksimalna suma segmenta i indeks njenog pocetka
  double maxSuma = numeric_limits<double>::min();
  int maxPocetak = 0;

  for (int i = 0; i <= n - k; i++) {
    // izracunavamo sumu segmenta duzine k koji pocinje na poziciji i
    double suma = 0;
    for (int j = i; j < i+k; j++)
      suma += a[j];
    // double suma = accumulate(next(a, i), next(a, i+k), 0.0);

    // ako je potrebno, azuriramo maksimum
    if (suma >= maxSuma) {
      maxSuma = suma;
      maxPocetak = i;
    }
  }

  // vracamo pocetak poslednjeg segmenta sa maksimalnom sumom
  // (ujedno i prosekom)
  return maxPocetak;
}
```

**Анализа сложености.** Број операција сабирања у овом алгоритму приступу је око $(n-k)\cdot k$ (јер одређивање сваког од $n-k$ сегмената захтева око $k$ операција сабирања), што даје квадратни број операција када је $k$ око $n/2$. Временску сложеност најгорег случаја, дакле, можемо проценити са $O(n^2)$. Пошто све елементе

памтимо у низу, меморијска сложеност је $O(n)$.

**Инкременталност - покретни прозор**

Међутим, пошто се узастопни сегменти у великој мери преклапају (разликују им се само почетни и завршни елемент) могућ је и бољи приступ, у којем се збир наредног сегмента дужине $k$ рачуна на основу познатог збира претходног сегмента дужине $k$. Ако са $S_i$ обележимо збир сегмента који почиње од елемента $a_i$ и дужине је $k$, $S_i = a_i + a_{i+1} + ... + a_{i+k-1}$, лако се може показати да за $i > 0$ важи једнакост $S_i = S_{i-1} - a_{i-1} + a_{i+k-1}$. Према томе можемо израчунати збир $S_0$ као збир првих $k$ елемената низа (опет уобичајеним алгоритмом сабирања), а затим за свако $i$ од 1 до $n - k$ наредни збир $S_i$ добијамо на основу претходне једнакост. Дакле, ефикасније решење добијамо ако збир рачунамо инкрементално (сваки наредни члан се израчунава на основу претходних).

```
// pronalazi indeks pocetka segmenta duzine k ciji je prosek najveci
int pocetakSegmentaNajvecegProseka(const vector<double>& a, int k) {
  // duzina niza
  int n = a.size();

  // suma pocetnog segmenta duzine k
  double suma = 0;
  for (int i = 0; i < k; i++)
    suma += a[i];

  // trenutna maksimalna suma segmenta i indeks njenog pocetka
  int maxPocetak = 0;
  double maxSuma = suma;

  for (int i = 1; i <= n - k; i++) {
    // izracunavamo sumu segmenta duzine k koji pocinje na poziciji i
    suma = suma - a[i - 1] + a[i + k - 1];

    // ako je potrebno, azuriramo maksimum
    if (suma >= maxSuma) {
      maxSuma = suma;
      maxPocetak = i;
    }
  }

  // vracamo pocetak poslednjeg segmenta sa maksimalnom sumom
  // (ujedno i prosekom)
  return maxPocetak;
}
```

**Анализа сложености.** Оптимизовани приступ је знатно ефикаснији у односу на директни, јер се у сваком кораку петље врши само константан број операција, па је сложеност алгоритма линеарна тј. $O(n)$. Елементи се чувају у низу дужине $n$, па је меморијска сложеност $O(n)$.

*Види другачија решења овог задатка.*

## 2.13 Збирови префикса и разлике суседних елемената низа

Ако је дат низ елемената збир свих елемената у интервалу позиција $[a, b]$ се може израчунати као разлика између збира свих елемената у интервалу позиција $[0, b]$ и збира елемената у интервалу позиција $[0, a - 1]$.

На пример, размотримо како да израчунамо збир елемената на позицијама из интервала [3, 5] (тј. на позицијама 3, 4 и 5) у низу 4, 2, 3, 1, 5, 6, 9, 2. На тим позицијама се налазе елементи 1, 5 и 6 и збир им је $1 + 5 + 6 = 12$. Збир свих елемената из интевала позиција $[0, 5]$ је $4 + 2 + 3 + 1 + 5 + 6 = 21$, док је збир свих елемената из интервала позиција $[0, 2]$ једнак $4 + 2 + 3 = 9$. Разлика $21 - 9$ управо је једнака 12.

Ова наизглед веома једноставна особина сабирања може значајно помоћи убрзању разних алгоритама у којима су нам потребни збирови узастопних елемената низа. Наиме, ако знамо *збирове свих префикса* низа тј.

збирове на свим интервалима $[0, k)$, за $k = 0$ до $n$ (а њих можемо израчунати током фазе претпроцесирања, инкрементално, у линеарној сложености), тада у константној сложености (једним одузимањем) можемо израчунати збир произвољног сегмента низа.

У језику C++ парцијалне збирове је могуће израчунати и коришћењем библиотечке функције `partial_sum`, која, наравно, ради у линеарној сложености. Функцији се прослеђују два итератора на део низа који се сабира, као и итератор на почетак низа у који се смештају резултати (пошто се унапред зна колико ће елемената бити, тај низ се унапред алоцира).

Дакле, од датог низа, низ збирова префикса можемо израчунати у линеарној сложености, али важи и обратно. Од низа префикса, у линеарној сложености можемо израчунати елементе оригиналног низа. Важи чак и јаче тврђење од тога, јер сваки конкретни елемент низа можемо наћи у константној сложености, одузимањем два суседна збира префикса. Дакле, прелазак са низа на збирове његових префикса можемо сматрати променом репрезентације (нема смисла чувати и једно и друго истовремено у меморији).

Приметимо огромну сличност са интегралним и диференцијалним рачуном. Израчунавање збирова префикса одговара одређеном интеграљењу, разлика збирова префикса одговара Њутн-Лајбницовој формули, док израчунавање разлике суседних елемената одговара диференцирању. Интеграљење и диференцирање су međусобно инверзне операције.

Дуалан приступ збировима префикса је промена репрезентације у којој уместо низа чувамо разлике суседних елемената. Повратак на оргинални низ се онда може извршити у линеарној сложености тако што израчунамо збирове префикса низа разлика. Ова репрезентација нам омогућава да веома ефикасно мењамо сегменте низа тако што све елементе из неког задатог сегмента увећамо или умањимо за неку фиксну вредност.

## Задатак: Збирови сегмената

Позната је зарада једног предузећа током одређеног броја дана. Напиши програм који омогућава кориснику да израчунава укупну зараду предузећа у временским периодима одређеним почетним и крајњим даном.

**Улаз:** Са стандардног улаза се уноси број дана $n$ ($1 \leq n \leq 100000$), а затим у наредном реду $n$ целих бројева између $0$ и $100$, раздвојених са по једним размаком, који представљају зараде током $n$ дана. Након тога се уноси број упита $m$ ($1 \leq m \leq 100000$) и у наредних $m$ редова се уносе временски периоди одређени редним бројем почетног дана $a$ и крајњег дана $b$ ($0 \leq a \leq b < n$).

**Излаз:** На стандардни излаз исписати $m$ целих бројева који представљају укупне зараде у сваком од $m$ периода.

**Пример**

| *Улаз* | *Излаз* |
|---|---|
| 5 | 15 |
| 1 2 3 4 5 | 9 |
| 3 | 3 |
| 0 4 | |
| 1 3 | |
| 2 2 | |

**Решење**

### Директно решење

Директно решење подразумева да све бројеве учитамо у низ, а затим да за сваки упит изнова рачунамо збир одговарајућег сегмента низа.

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

// ucitavamo granice segmenata i izracunavamo i ispisujemo njihove zbirove
int m;
cin >> m;
```

```
for (int i = 0; i < m; i++) {
  int a, b;
  cin >> a >> b;
  int zbir = 0;
  for (int j = a; j <= b; j++)
    zbir += brojevi[j];
  cout << zbir << endl;
}
```

**Анализа сложености.** Сложеност оваквог приступа је $O(nm)$.

Пошто се фаза учитавања и исписа података преплићу, учитавање би и испис података би требало додатно убрзати, но то не може поправити неефикасност овог наивног алгоритма.

### Збирови префикса

Једноставно ефикасно решење је засновано на наредној идеји: уместо чувања елемената низа, можемо чувати низ збирова префикса низа. Збир сваког сегмента $[l, d]$ можемо разложити на разлику збира префикса до елемента $d$ и префикса до елемента $l-1$. Сличну техника је употребљена у задатку Аритметички троугао. Ако користимо ознаку $\sum_{k=m}^{n} a_k$ која означава збир $a_m + a_{m+1} + \ldots + a_n$, можемо записати да је

$$\sum_{k=l}^{d} a_k = \sum_{k=0}^{d} a_k - \sum_{k=0}^{l-1} a_k.$$

Збирови свих префикса се могу израчунати и сместити у додатни (а ако је уштеда меморије битна, онда чак и у оригинални) низ. Дакле, током учитавања елемената можемо формирати низ збирова префикса (рачунаћемо их инкрементално, јер се сваки наредни збир префикса добија увећавањем претходног збира префикса за текући елемент низа). Нека $z_i$ означава збир елемената префикса одређеног позицијама из интервала $[0, i)$. Формирамо, дакле, низ $z_i = \sum_{k=0}^{i-1} a_k$ (при чему је $z_0 = 0$, збир празног префикса). Тада збир елемената у сегменту позиција $[l, d]$ израчунавамо као $z_{d+1} - z_l$.

```
ios_base::sync_with_stdio(false); cin.tie(0);
// ucitavamo brojeve i izracunavamo zbirove prefiksa
int n;
cin >> n;
vector<int> zbirovi_prefiksa(n+1);
zbirovi_prefiksa[0] = 0;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  zbirovi_prefiksa[i+1] = zbirovi_prefiksa[i] + x;
}

// ucitavamo granice segmenata i izracunavamo i ispisujemo njihove zbirove
int m;
cin >> m;
for (int i = 0; i < m; i++) {
  int a, b;
  cin >> a >> b;
  cout << zbirovi_prefiksa[b+1] - zbirovi_prefiksa[a] << '\n';
}
```

**Анализа сложености.** За учитавање бројева и формирање низа збирова префикса потребно нам је $O(n)$ корака. Након оваквог претпроцесирања, збир сваког сегмента се може израчунати у времену $O(1)$, па је укупна сложеност $O(n+m)$.

Пошто се у овом задатку преплићу фаза учитавања и фаза исписа података на стандардни улаз и излаз, потребно је обратити пажњу на неефикасност која настаје због честог пражњења излазног бафера. Потребно је развезати `cin` и `cout` коришћењем `cin.tie(0)` и уместо помоћу `endl` у нови ред прелазити помоћу `\n`. На-

равно, ово има смисла само у случају аутоматске примене програма на велике улазе и излазе - овим изменама програм престаје да ради коректно у интерактивном режиму.

## Задатак: Максимални збир сегмента

*Овај задатак је поновљен у циљу увежбавања различитих техника решавања. Види текст задатка.*

*Покушај да задатак урадиш коришћењем техника које се излажу у овом поглављу.*

**Решење**

**Збирови префикса**

Један алгоритам којим можемо ефикасно решити овај задатак се заснива на коришћењу збирова префикса. Сличну техника је описана, на пример, у задатку Збирови сегмената. Ако знамо збир сваког префикса низа, онда збир сваког сегмента можемо добити као разлику збирова два префикса. Збир елемената сегмента $[l, d]$ једнак је разлици збира елемената сегмента $[0, d+1)$ и збира елемената сегмента $[0, l)$, тј. важи да је

$$\sum_{i=l}^{d} a_i = \sum_{i=0}^{d} a_i - \sum_{i=0}^{l-1} a_i$$

Рачунамо да је збир празног сегмента $[0, 0)$ по дефиницији једнак нули.

За сваку позицију у низу одређујемо максимални збир суфикса који се на тој позицији завршава. Највећи од свих максималних збирова суфикса је највећи збир неког сегмента у низу (јер је сваки сегмент заправо суфикс дела низа до оне позиције на којој се тај сегмент завршава).

Користимо индуктивну конструкцију и низ проширујемо једним по једним елементом. Крећемо од празног низа чији је максимални збир сегмента једнак нули. Приликом сваког проширивања низа новим елементом, претпостављамо да знамо максимум збирова сегмента у непроширеном делу низа и да израчунамо максимални збир суфикса проширеног низа. Максимум збирова сегмента у проширеном низу је већи од та два броја.

Максимални збир суфикса проширеног низа, на основу разлагања на збирове префикса, добија се као разлика збира целог проширеног низа (тј. збира префикса до текуће позиције) и збира неког префикса непроширеног низа (празан суфикс не морамо анализирати, јер је празан сегмент већ обрађен у склопу инцијализације). Пошто је умањеник константан, да бисмо максимизовали разлику потребно да знамо најмањи могући умањилац, тј. да знамо најмањи збир префикса који се завршава на некој позицији испред текуће. И текући и минимални збир префикса можемо одржавати инкрементално. Инкрементално израчунавање збирова префикса већ је описано у задатку Највећи збир префикса.

Када год низ проширимо неким елементом, збир префикса увећавамо за тај елемент, поредимо га са дотадашњим минималним збиром префикса и ако је мањи, ажурирамо минимални збир префикса. Наравно, одржавамо и глобални максимални збир сегмента који ажурирамо сваки пут када наиђемо на суфикс чији је збир већи од дотадашњег максимума.

Приметимо да је у овом решењу је индуктивна хипотеза појачана и претпостављамо да поред сегмента највећег збира у обрађеном делу низа умемо да одредимо и минимални збир префикса обрађеног дела низа.

**Анализа сложености.** Сложеност овог решења је $O(n)$, па је ово решење оптималне сложености.

```
int zbir_prefiksa = 0;
int min_zbir_prefiksa = zbir_prefiksa;
int max = 0;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  zbir_prefiksa += x;
  int zbir_segmenta = zbir_prefiksa - min_zbir_prefiksa;
  if (zbir_segmenta > max)
    max = zbir_segmenta;
  if (zbir_prefiksa < min_zbir_prefiksa)
    min_zbir_prefiksa = zbir_prefiksa;
```

```
}
cout << max << endl;
```

*Види другачија решења овог задатка.*

## Задатак: Број сегмената чији је збир дељив са k

Дат је низ $a$ природних бројева дужине $n$ и природан број $k$. Написати програм који одређује број сегмента низа $a$ (непразних поднизова узастопних елемената) чији је збир дељив са $k$.

**Улаз:** У првој линији стандардног улаза налази се природан број $k$ ($k \leq 10^5$). Друга линија стандардног улаза садржи природан број $n$ ($n \leq 10^5$). У следећој линији се налази $n$ природних бројева (ти бројеви представљају редом елементе низа $a$), раздвојених са по једним размаком.

**Излаз:** На стандардном излазу приказати колико постоји сегмената низа $a$ чији је збир дељив са $k$.

### Пример

| *Улаз* | *Излаз* |
|---|---|
| 3<br>5<br>1 8 2 3 4 | 4 |

*Објашњење*: то су сегменти $(1, 8)$, $(3)$, $(2, 3, 4)$ и $(1, 8, 2, 3, 4)$

### Решење

#### Груба сила

Директан начин да се задатак реши је да се угнежђеним петљама наброје сви сегменти, да се за сваки израчуна сума и да се провери да ли је дељива са $k$, бројећи успут такве сегменте. Број је дељив са $k$ ако и само ако даје остатак $0$ при дељењу са $k$, а знамо да је остатак при дељењу збира са бројем $k$ заправо једнак збиру остатака тј. да важи да је $(a + b) \bmod k = (a \bmod k + b \bmod k) \bmod k$.

Дакле, за сваки сегмент довољно је израчунати збир по модулу $k$, при чему за фиксирани леви крај сегмента, збир сваког наредног сегмента $[i, j]$ добијамо инкрементално, од збира сегмента $[i, j - 1]$, додајући на њега број $a_j$ и израчунавајући остатак добијеног збира при дељењу са $k$. Збир префикса смо рачунали инкрементално, на пример, у задатку Највећи збир префикса.

```
// izracunava broj segmenata niza a ciji je zbir deljiv sa k
int brojSegmenataZbiraDeljivogSaK(const vector<int>& a, int k) {
  // duzina niza
  int n = a.size();
  // broj segmenata deljivih sa k
  int broj = 0;
  // obradjujemo sve segmente [i, j]
  for (int i = 0; i < n; i++) {
    // zbir po modulu k segmenta [i, j] inicijalizujemo na nulu
    int s = 0;
    for (int j = i; j < n; j++) {
      // azuriramo zbir po modulu k segmenta [i, j] na osnovu zbira [i, j-1]
      s = (s + a[j]) % k;
      // ako je zbir po modulu k jednak 0, zbir je deljiv sa k
      if (s == 0)
        broj++;
    }
  }
  return broj;
}
```

**Анализа сложености.** Уз овако инкрементално израчунавање збира, сложеност алгоритма једнака је броју сегмената што је $O(n^2)$.

### Остаци збирова префикса

Тражимо број сегмената $a_p, a_{p+1}, \ldots, a_q$, за $0 \leq p \leq q < n$, таквих да је збир $S_{pq} = a_p + a_{p+1} + ... + a_q$ дељив са $k$.

Обележимо са $S_0 = 0$, $S_1 = a_0$, $S_2 = a_0 + a_1$, итд., тј. обележимо са $S_i$, $0 < i \leq n$ збир првих $i$ елемената низа ($S_i = a_0 + a_1 + ... + a_{i-1}$). Збир $S_{pq}$ можемо изразити као $S_{q+1} - S_p$. Сличну технику користили смо, на пример, у задатку Сегмент датог збира у низу целих бројева.

На основу особина операција по модулу важи да је $S_{pq} \bmod k = (S_{q+1} \bmod k - S_p \bmod k + k) \bmod k$.

Према томе збир $S_{pq}$ је дељив са $k$ (тј. важи $S_{pq} \bmod k = 0$) акко збирови $S_{q+1}$ и $S_p$ имају исти остатак при дељењу са $k$ тј. ако је $S_{q+1} \bmod k = S_p \bmod k$.

Обележимо са $b_r$ број збирова $S_i$ (за $0 \leq i \leq n$) који при дељењу са $k$ дају остатак $r$ (за свако $0 \leq r < k$). Сваки пар (различитих) збирова префикса који дају исти остатак $r$ одређује тачно један сегмент чији је збир елемената дељив са $k$. У скупу од $m$ различитих елемената постоји тачно $\frac{m(m-1)}{2}$ различитих парова. Зато је за свако $r$ број сегмената који се добија комбинујући два збира који дају остатак $r$ једнак $\frac{b_r \cdot (b_r - 1)}{2}$. Према томе укупан број сегмената дељивих са $k$ је $\sum_{r=0}^{k-1} \frac{b_r \cdot (b_r - 1)}{2}$.

Остаје још само питање како избројати збирове префикса за сваки дати остатак тј. како израчунати све бројеве $b_r$. Низом $b$ дужине $k$ памтимо број префикса чији збирови елемената дају остатке редом $0, 1, 2, \ldots, k-1$ тако да је $b_r$ једнак броју префикса чији збир елемената при дељењу са $k$ даје остатак $r$ (што је могуће, с обзиром на дату горњу границу броја $k$). Збирове префикса, наравно, израчунавамо инкрементално. Слично смо радили, на пример, у задатку Највећи збир префикса.

Полазни збир је $S_0 = 0$, тако да све елементе низа $b$ иницијализујемо на 0, осим вредности на позицији 0 коју иницијализујемо на 1. Збир текућег префикса одржавамо у променљивој $S$ коју иницијализујемо на нулу. Учитавамо члан по члан низа $x$, и при томе ажурирамо збир префикса ($S$ постављамо на $(S + x) \bmod k$), увећавајући одговарајући бројач (вредност $b_S$ увећавамо за 1).

**Пример.** Прикажимо рад овог алгоритма на примеру одређивања броја сегмената низа $1, 8, 2, 3, 4$ који су дељиви са 3. Могући остаци су 0, 1 и 2.

```
i  ai Si  b0 b1 b2
      0   1  0  0
0  1  1   1  1  0
1  8  0   2  1  0
2  2  2   2  1  1
3  3  2   2  1  2
4  4  0   3  1  2
```

Зато је коначан резултат $\frac{b_0(b_0-1)}{2} + \frac{b_1(b_1-1)}{2} + \frac{b_1(b_1-1)}{2} = 3 + 0 + 1 = 4$.

```
// izracunava broj segmenata niza a ciji je zbir deljiv sa k
int brojSegmenataZbiraDeljivogSaK(const vector<int>& a, int k) {
  // duzina niza
  int n = a.size();

  // na mestu i u nizu br čuvamo broj segmenata čiji zbir pri
  // deljenju sa k daje ocmamak i
  vector<int> br(k, 0);
  br[0] = 1;

  // zbir tekućeg segmenata
  int s = 0;
  for (int i = 0; i < n; i++) {
    // ažuriramo zbir elemenata tekućeg segmenta po modulu k
    s = (s + a[i]) % k;
    br[s]++;
  }

  // izračunavamo ukupan broj segmenata deljivih sa k
```

```
  int broj = 0;
  for(int i = 0; i < k; i++)
    broj += br[i]*(br[i]-1)/2;

  return broj;
}
```

**Анализа сложености.** Временска сложеност овог алгоритма је, јасно, $O(n)$. Приметимо да у овом решењу није било потребно користити низ за бројеве које уносимо, јер при уносу обрадимо сваки елемент, међутим, користимо помоћни низ дужине $k$, па је меморијска сложеност $O(k)$. Обратимо пажњу на то да ово може бити ограничавајући фактор, ако $k$ може бити јако велики број (што у овом задатку није случај).

## Задатак: Увећавање сегмената

Камион превози терет током $N$ километара пута. На пут креће празан и током пута утоварује и истоварује пакете. Ако се за сваки пакет зна на ком је километру пута утоварен, на ком је километру пута истоварен и колика му је маса, напиши програм који одређује колико је оптерећење камиона на сваком километру пута. Сматрати да се предмет утоварује на почетку, а истоварује на крају датог километра.

**Улаз:** Са стандардног улаза се уноси број километара $N$ $(10 \leq N 10000)$, затим, у наредном реду, број предмета $M$ $(0 \leq M \leq 10000)$, а након тога, у наредних $M$ редова по три цела броја раздвојена размацима који представљају број километра на чијем је почетку утоварен предмет (цео број између 0 и $N-1$), број километра на чијем крају је истоварен (цео број између 0 и $N-1$) и на крају маса предмета (цео број између 1 и 10).

**Излаз:** На стандардни излаз исписати масу терета у килограмима на сваком километру пута (иза сваке масе написати по један размак).

**Пример**

| *Улаз* | *Излаз* |
|---|---|
| `10`<br>`3`<br>`1 5 10`<br>`3 7 10`<br>`2 8 15` | `0 10 25 35 35 35 25 25 15 0` |

*Објашњење*

```
       km   0   1  2  3  4  5  9  7  8  9
            0   0  0  0  0  0  0  0  0  0
1 5 10      0  10 10 10 10 10  0  0  0  0
3 7 10      0  10 10 20 20 20 10 10  0  0
2 8 15      0  10 25 35 35 35 25 25 15  0
```

**Решење**

**Директно решење**

Директан начин је да се одржава низ $M$ у којем се памти маса на камиону током сваког километра пута. Након учитавања сваког податка о предмету (почетног километра $a$, завршног километра $b$ и масе $m$), све вредности у низу $M$ на позицијама од $a$ до $b$ (укључујући и њих) се увећавају за $m$.

**Анализа сложености.** Проблем са овим решењем је то што предмети могу путовати велики број километара па се у сваком кораку врши ажурирање великог броја чланова низа (сложеност је у најгорем случају $O(n \cdot m)$).

```
int n;
cin >> n;
vector<int> mase(n, 0);
int m;
cin >> m;
for (int i = 0; i < m; i++) {
  int km_od, km_do, masa;
  cin >> km_od >> km_do >> masa;
```

```
  for (int km = km_od; km <= km_do; km++)
    mase[km] += masa;
}

for (int masa : mase)
  cout << masa << " ";
```

### Разлике суседних елемената низа

Задатак можемо решити ефикасније ако уместо да у низу $M$ одржавамо масу у камиону у километру $i$, одржавамо разлику између масе у километру $i$ и $i-1$ (на позицији 0 се чува маса у камиону у нултом километру). Дакле, уводимо низ $R_i$ такав да је $R_0 = M_0$, а $R_i = M_i - M_{i-1}$, за $1 \leq i < n$. Посматрајмо шта се дешава са низом $R$ када се у низу $M$ сви елементи на позицијама $a$ до $b$ увећају за $m$.

- Вредност $R_a$ једнака је разлици $M_a - M_{a-1}$ (или евентуално $M_0$ ако је $a = 0$) и она се увећава за $m$, јер је $M_a$ увећан за $m$, док се $M_{a-1}$ не мења.
- Све вредности од $R_{a+1}$ до $R_b$ остају не промењене. Наиме, за све њих важи да је $R_i = M_i - M_{i-1}$, а да су и $M_i$ и $M_{i-1}$ увећани за $m$.
- На крају, вредност $R_{b+1}$ се умањује за $m$. Наиме важи да је $R_{b+1} = M_{b+1} - M_b$, да се $M_b$ увећава за $m$, док се $M_{b+1}$ не мења.

Рецимо да ако је $b = n - 1$, тада не морамо разматрати вредност $R_{b+1} = R_n$ (мада, униформности ради, можемо, што захтева да низ $R$ садржи $n + 1$ елемент). Дакле, приликом сваког учитавања бројева $a$, $b$ и $m$ потребно је само да увећавамо елемент $R_a$ за $m$, а да елемент $R_{b+1}$ умањимо за $m$.

Када знамо елементе низа $R$ елементе низа $M$ можемо једноставно реконструисати сабирањем елемената низа $R$. Наиме, важи да је $M_0 = R_0$, док је $M_i = M_{i-1} + R_i$, тако да сваки наредни елемент низа $M$ можемо израчунати као збир претходног елемента низа $M$ и њему одговарајућег елемента низа $R$. Приметимо да је заправо елемент $M_i$ једнак збиру свих елемената од $R_0$ до $R_i$, јер је $R_0 + R_1 + \ldots + R_i = M_0 + (M_1 - M_0) + \ldots + (M_i - M_{i-1}) = M_i$.

**Анализа сложености.** Укупна сложеност овог приступа је $O(n + m)$.

```
int n;
cin >> n;
vector<int> razlika(n+1, 0);
int m;
cin >> m;
for (int i = 0; i < m; i++) {
  int km_od, km_do, masa;
  cin >> km_od >> km_do >> masa;
  razlika[km_od] += masa;
  razlika[km_do+1] -= masa;
}

int masa_km = 0;
for (int km = 0; km < n; km++) {
  masa_km += razlika[km];
  cout << masa_km << " ";
}
```

**Напомена.** Идеја коришћена у овом задатку донекле је слична (заправо инверзна) техници одређивања збира сегмената као разлике два збира префикса. Ту технику смо, на пример, применили у задатку Збирови сегмената. Може се приметити да се реконструкција низа врши заправо израчунавањем префиксних збирова низа разлика суседних елемената, што указује на дубоку везу између ове две технике. Заправо, разлике суседних елемената представљају одређени дискретни аналогон извода функције, док префиксни збирови представљају аналогију одређеног интеграла. Израчунавање збира сегмента као разлике два збира префикса одговара Њутн-Лајбницовој формули.