- 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 $R_0 + R_1 + \ldots + R_i = M_0 + (M_1 - M_0) + \ldots + (M_i - M_{i-1}) = M_i$.

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

Идеја коришћена у овом задатку донекле је слична (заправо инверзна) техници виђеној 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.

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

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  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 << " ";
  }

  return 0;
}
```

2.14 Сортирање

Због својих примена, сортирање је један од најзначајнијих и најпроучаванијих проблема у рачунарству. Током година развијен је велики број алгоритама за сортирање. Најпознатији су *QuickSort*, *MergeSort*, *HeapSort*, *SelectionSort*, *InsertionSort*, *BubbleSort* и *ShellSort*. Неки од њих су једноставнији, али спорије раде (такви су *Selection*, *Insertion* и *Bubble*), док су неки ефикаснији, али компликованији за разумевање и реализацију (такви су *Quick*, *Merge*, *Heap* и *Shell*). Њихова ручна имплементација поучна је као одличан пример за приказ различитих техника конструкције алгоритама.

Ипак, по правилу све библиотеке савремених (а и старијих) програмских језика нуде готове функције за сортирање низова. Коришћење ових функција је увек пожељнија опција у односу на ручну имплементацију

(добијају се краћи и разумљивији програми, функције су пажљиво тестиране и скоро извесно потпуно коректне, и имплементиране су на најјефикаснији могући начин). Можемо слободно претпоставити да је сложеност и најгорег и просечног случаја код библиотечких функција сортирања квазилинеарна тј. једнака $O(n \log n)$ - у делу посвећеном рекурзији и техници "подели па владај", биће приказани најзначајнији алгоритми сортирања (који се користе и у библиотечким функцијама сортирања) и њихова анализа сложености.

У наставку овог поглавља кроз неколико веома елементарних примера ћемо приказати употребу бибиотечких функција сортирања, као и пар елементарних алгоритама сортирања (који се једноставно имплементирају, али су квадратне сложености и стога их никада у пракси не треба употребљавати).

Сортирање подразумева да се елементи ређају у односу на неки поредак. Бројеви се подразумевано ређају по њиховој нумеричкој вредности, ниске се подразумевано ређају лексикографски абецедно (као у речнику), парови и торке се поново ређају лексикографски, по својим компонентама, међутим, често је потребно сортирати низове елемената над којима није дефинисан подразумевани поредак (на пример, низове структура). Библиотечке функције сортирања допуштају навођење различитих критеријума сортирања (задавањем функција поређења), што ће такође бити илустровано кроз неколико елементарних задатака.

Након тога, приказаћемо употребу сортирања као технике претпроцесирања која омогућава да се снизи сложеност решења задатака. Сортирање је толико корисна техника, да у свету конструкције алгоритама често важи правило "Ако не знаш одакле да кренеш да конструишеш алгоритам, ти прво покушај да сортираш податке". Једна од најзначајнијих примена сортирања је то што сортирани подаци могу брзо претраживати (применом алгоритма бинарне претраге). Стога ће у наредном поглављу посебна пажња бити посвећена управо томе.

2.14.1 Обрада дупликата (поновљених вредности у низу)

У неким задацима је потребно на неки начин обрадити све поновљене вредности у низу (дупликате). Ефикасна решења се обично добијају након што се низ претпроцесира коришћењем сортирања. Након сортирања низа сви поновљени елементи се налазе један иза другога, што значајно онда олакшава њихову обраду (за сваки елемент је веома једноставно проверити колико пута се јавио у низу, па је самим тим једноставно проверити и да ли је дупликат, уклонити дупликате и слично). Осим сортирањем, обрада дупликата се може вршити и помоћу библиотечких колекција (скупова, мултискупова и мапа тј. речника), о чему ће више речи бити касније.

Задатак: Дупликати

Претпоставимо да су интернет адресе представљене природним бројевима (IP адресе се, на пример, чувају у облику неозначених 32-битних бројева). Претраживач чува списак свих адреса које је корисник посетио током неког претходног периода. Корисник је многе адресе посећивао и више пута. Напиши програм који одређује број различитих адреса које је корисник посетио.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 10^5$), а затим и n природних бројева (мањих од 100000), сваки у посебном реду.

Излаз: На стандардни излаз исписати тражени број.

Пример

Улаз	*Излаз*
6 17 9 17 8 3 9	4

Решење

Претрага

Наиван начин да се детектују дупликати се може засновати на алгоритму линеарне претраге. Од свих учитаних елемената низа бројаћемо само оне који се први пут појављују (када се неки елемент појави други, трећи итд. пут, нећемо га бројати). За сваки елемент низа на позицији i (од нула до $n-1$) провераваћемо да ли се тај елемент јавља на некој позицији од 0 до $i-1$. То можемо урадити алгоритмом линеарне претраге, који смо увели у задатку Негативан број.

Ово решење је нефикасно и сложеност му је квадратна у односу на димензију низа (сложеност му је $O(n^2)$, где је n број елемената низа).

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

using namespace std;

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

  int brojRazlicitih = 0;
  // za svaki element niza a
  for (int i = 0; i < n; i++) {
    // proveravamo da li se a[i] javlja pre pozicije i
    bool sadrzi = false;
    for (int j = 0; j < i; j++)
      if (a[i] == a[j]) {
        sadrzi = true;
        break;
      }
    // ako se ne pojavljuje, ispisujemo ga
    if (!sadrzi)
      brojRazlicitih++;
  }
  cout << brojRazlicitih << endl;
  return 0;
}
```

Сортирање

Један од најчешћих начина уклањања дупликата из низа је заснован на сортирању, јер се након сортирања дупликати нађу један до другог. Сортирање можемо урадити на неки од начина приказаних у задатку Сортирање бројева — најбоље позивом библиотечке функције. Након сортирања пролазимо редом кроз низ и бројимо први елемент, а затим и све елементе који су различити од њима претходног (то су прва појављивања елемената у сортираном низу).

Сложеношћу овог приступа доминира сложеност поступка сортирања. Пролаз након сортирања је линеарне сложености, а сортирање се може остварити у сложености $O(n \log n)$, где је n број елемената низа.

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

using namespace std;

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

```
  // sortiramo niz
  sort(a.begin(), a.end());
  // brojimo prvi element i sve elemente koje su razliciti od
  // svojih prethodnika
  int brojRazlicitih = 1;
  for (int i = 1; i < a.size(); i++)
    if (a[i] != a[i-1])
      brojRazlicitih++;
  cout << brojRazlicitih << endl;

  return 0;
}
```

Библиотечка функција за избацивање дупликата

Избацивање дупликата у сортираном низу може се вршити и библиотечким функцијама. Функција `unique` у језику C++ прима два итератора који ограничавају сегмент сортираног низа тј. вектора (обично `a` и `next(a, n)` тј. `a+n` или `a.begin()` и `a.end()`) и организује тај низ тако да на његов почетак смешта елементе који се не понављају и враћа итератор на позицију иза краја тог дела. На пример, ако је низ $1, 1, 2, 2, 5, 5$, након позива функције `unique` он ће бити реорганизован тако што ће му садржај бити $1, 2, 5, x, x, x$, а итератор ће указивати на позицију три тј. непосредно иза елемента 5. Саржај на позицијама обележеним са x није релевантан. Ако се жели избацивање дупликата, реп низа се може одстранити тако што се позове функција `erase` којој се наведу два итератора - први је онај који је вратила функција `unique` а други је итератор на крај низа (који се обично добија са `next(a, n)` `a + n` или `a.end()`). Ако је потребно израчунати само број јединствених елемената он се може добити тако што се израчуна растојање између итератора на почетак низа и итератора којег врати функција `unique` (на пример, `distance(a, unique(a, next(a, n)))`).

Библиотечке функције за уклањање дупликата из сортиране колекције сложености су $O(n)$, па укупном сложеношћу доминира сортирање сложености $O(n \log n)$.

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

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  // ucitavamo niz
  int n;
  cin >> n;
  vector<unsigned> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  // sortiramo niz
  sort(a.begin(), a.end());
  // uklanjamo duplikate
  a.erase(unique(a.begin(), a.end()), a.end());
  // ispisujemo rezultat
  cout << a.size() << endl;

  // dovoljno je i:
  // cout << distance(a.begin(), unique(a.begin(), a.end())) << endl;
  return 0;
}
```

Задатак: Неупарени елемент

Домаћица матица је на журку позвала своје пријатеље, брачне парове пчеле и трутове. Пошто је гостију пуно, свако је добио број столице. Брачни парови су добили исте бројеве. Који број је добила матица?

Улаз: Са стандардног улаза уноси се број n, а затим и n природних бројева, сваки у посебном реду, од којих се сви осим једног јављају тачно два пута.

Излаз: На стандардни излаз исписати један број - онај који се на улазу јавио тачно једном.

Пример

Улаз	*Излаз*
9 3 2 1 4 2 5 5 3 1	4

Решење

Груба сила

Директно решење задатка подразумева да се за сваки елемент учитаног низа провери да ли у низу постоји још неко његово појављивање. Дакле, користимо угнежђене петље и у спољној петљи пролазимо кроз све позиције i у низу, док у унутрашњој петљи користимо алгоритам линеарне претраге (слично као у задатку Негативан број) да бисмо проверили да ли у низу постоји позиција j таква да је различита од i, а да је $a_i = a_j$. Први пут када се унутрашња петља неуспешно заврши, пријављујемо да је неупарени елемент a_i и прекидамо и спољашњу петљу. Наравно, претрагу је могуће издвојити и у помоћну функцију.

Пошто се суштински упоређују сви парови елемената, а њих у низу дужине n има $\frac{n(n-1)}{2}$, сложеност овог решења је $O(n^2)$.

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

using namespace std;

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

  for (int i = 0; i < n; i++) {
    bool uparen = false;
    for (int j = 0; j < n; j++)
      if (i != j && a[i] == a[j]) {
        uparen = true;
        break;
      }
    if (!uparen) {
      cout << a[i] << endl;
      break;
    }
  }

  return 0;
}
```

Сортирање

Задатак се ефикасније може решити ако се низ претходно сортира (слично као у задацима Дупликати и Дво-

струки студент). Сортирање можемо урадити на разне начине (слично као у задатку Сортирање бројева), али најбољи је да се употреби бибилотечка функција за сортирање. Након тога се сви парови налазе један иза другога и потребно је пролазити низ два-по-два елемента проверавајући да ли је други различит од првог (ако јесте, онда је први елемент тај који се јавља само један пут). Посебну пажњу треба обратити на случај када је тражени елемент последњи у сортираном низу.

Сложеношћу доминира сортирање, које је сложености $O(n \log n)$. Провера једнакости узастопних елемената која наступа након сортирања врши се у једном пролазу кроз низ и линеарне је сложености тј. $O(n)$.

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

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  sort(begin(a), end(a));
  int i;
  for (i = 0; i < n-1; i += 2)
    if (a[i] != a[i+1]) {
      cout << a[i] << endl;
      break;
    }
  if (i == n-1)
    cout << a[n-1] << endl;
  return 0;
}
```

2.14.2 Груписање блиских вредности

Још једна примена сортирања долази од тога што се након сортирања низа, елементи блиски по вредности наћи ће један близу другог. Ово нам омогућава да у низу налазимо што ближе елементе као и групе што блискијих елемената (са што мањом разликом између најмањег и највећег елемента у групи).

Задатак: Најближе собе

Два госта су дошла у хотел и желе да одседну у собама које су што ближе једна другој, да би током вечери могли да заједно раде у једној од тих соба. Ако постоји више таквих соба, они бирају да буду што даље од рецепције, тј. у собама са што већим редним бројевима, како им бука не би сметала. Ако је познат списак слободних соба у том тренутку, напиши програм који одређује бројеве соба које гости треба да добију.

Улаз: У првој линији стандардног улаза налази се број n ($1 \leq n \leq 10^5$), а затим се у наредним линијама налазе бројеви слободних соба (у свакој линији по један) - сви бројеви су различити, али је њихов редослед произвољан.

Излаз: На стандардни излаз исписати бројеве соба гостију (прво мањи број, па већи), раздвојене једним размаком.

Пример

Улаз	*Излаз*
7 18 6 25 11 4 1 16	16 18

Решење

Груба сила

Директан приступ решењу би био да се израчунају растојања између сваке две собе и да се пронађе пар са најмањим растојањима.

Пошто парова има $\frac{n(n-1)}{2}$, сложеност овог приступа би била $O(n^2)$.

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

using namespace std;

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

  // dve sobe sa najmanjim rastojanjem
  int min_i = a[0], min_j = a[1];
  // rastojanje izmedju njih
  int d_min = abs(min_i - min_j);
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++) {
      // rastojanje izmedju soba i i j
      int d_ij = abs(a[i] - a[j]);
      // ako je rastojanje manje od dosada najmanjeg ili je jednako,
      // ali su sobe dalje od recepcije
      if (d_ij < d_min || (d_ij == d_min && min(min_i, min_j) < min(a[i], a[j]))) {
        // azuriramo najblize sobe i rastojanje izmedju njih
        min_i = a[i];
        min_j = a[j];
        d_min = d_ij;
      }
    }
  // ispisujemo sobe u uredjenom redosledu
  cout << min(min_i, min_j) << " " << max(min_i, min_j) << endl;
  return 0;
}
```

Сортирање

Боље решење се може добити ако се низ пре тога сортира. Наиме, најближи елемент сваком елементу у сортираном низу је један од њему суседних. Дакле, ако број a_i учествује у пару најближих соба, онда други елемент тог пара може бити или број a_{i-1} који је непосредно испред a_i у сортираном редоследу или број a_{i+1} који је непосредно иза њега (наравно, не постоји елемент испред првог, нити елемент иза последњег

елемента низа).

Заиста, у неопадајуће сортираном низу важи да из $j' < j < i$ следи $a_i - a_j \leq a_i - a_{j'}$, јер из сортираности и $j < j'$ следи да је $a_j \leq a_{j'}$, као и да из $i < j < j'$ следи да је $a_j - a_i \leq a_{j'} - a_i$, јер из сортираности и $j < j'$ следи $a_j \leq a_{j'}$. Зато је свако $j' < i - 1$ важи да је $a_i - a_{i-1} \leq a_i - a_{j'}$, па елемент на позицији j' није ближи елементу a_i него елемент a_{i-1}. Слично, за свако $j' > i + 1$ важи да је $a_{i+1} - a_i \leq a_{j'} - a_i$, па елемент на позицији j' није ближи елементу a_i него елемент a_{i+1}.

Зато је након сортирања довољно проверити све разлике између суседних елемената и одредити најмању од њих (ако има више истих, одређујемо последњу). За ово користимо алгоритам одређивања најмањег елемента, као у задатку Редни број максимума, док сортирање можемо најлакше ефикасно извршити библиотечком функцијом, као у задатку Сортирање бројева.

Сортирање библиотечком функцијом захтева $O(n \log n)$ операција, док је тражење минимума сложености $O(n)$, тако да је укупно време овог поступка $O(n \log n)$.

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

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  sort(begin(a), end(a));
  int min = 1;
  for (int i = 2; i < n; i++)
    if (a[i] - a[i-1] <= a[min] - a[min-1])
      min = i;
  cout << a[min-1] << " " << a[min] << endl;
  return 0;
}
```

Задатак: Праведна подела чоколадица

Дато је n пакета чоколаде и за сваки од њих је познато колико чоколадица садржи. Сваки од k ученика узима тачно један пакет, при чему је циљ да сви ученици имају што приближнији број чоколадица. Колика је најмања могућа разлика између оног ученика који узме пакет са најмање и оног који узме пакет са највише чоколадица.

Улаз: Са стандардног улаза се уноси природан број n ($1 \leq n \leq 50000$) а затим и n природних бројева (између 1 и 10^6, раздвојене са по једним размаком) који представљају број чоколадица у сваком пакету. У последњем реду се уноси број деце k ($1 \leq k \leq n$).

Излаз: На стандардни излаз исписати вредност најмање разлике.

Пример

Улаз	*Излаз*
8 3 8 1 17 4 7 12 9 4	5

Решење

Сортирање

Најдиректнији начин да се реши задатак је да се испитају сви подскупови од k елемената скупа од n елемената и да се међу њима одабере најбољи. Ово решење је релативно компликовано имплементирати, а уз то је и веома неефикасно (број подскупова је $\binom{n}{k}$, што је $O(n^k)$).

Боље и ефикасније решење се заснива на сортирању. Наиме, када се полазни пакети сортирају по броју чоколадица, ученици треба да узму узастопних k пакета. Претпоставимо да након сортирања имамо низ $a_0, a_1, \ldots, a_{n-1}$. Ученици треба да узму редом пакете од a_i, до a_{i+k-1}, за неко $0 \leq i \leq n-k$.

Докажимо претходну чињеницу и формално. Претпоставимо супротно, да пакети који дају најмањи распон не чине узастопан низ и да сваки узастопни низ пакета дужине k има строго већи распон од распона скупа узетих пакета. Нека је први узети пакет a_i. Тада сигурно постоји неки пакет a_j за $i < j < i+k$ који није узет, а уместо њега је узет неки пакет $a_{j'}$ за неко $i+k \leq j' < n$. Нека је j' последњи пакет који је узет. Међутим, када би ученик који је узео пакет $a_{j'}$ заменио тај пакет за $a_{j'}$ распон би се сигурно смањио или бар остао исти (јер би последњи узети пакет тада био неки пакет $a_{j''}$, за $j'' < j'$, а пошто је низ сортиран неопадајуће, важи да је $a_{j''} \leq a_j$, па и $a_{j''} - a_i \leq a_{j'} - a_i$. Даљим заменама истог типа можемо доћи до тога да су сви узети пакети узастопни, а да је распон мањи или једнак полазном, што је у контрадикцији са тим да је распон полазног скупа узетих пакета строго мањи од распона било којег низа k узастопних пакета.

Разматрање претходног типа је карактеристично за такозване грамзиве алгоритме, о ком ће више речи бити касније.

На основу претходног јасно је да низ бројева чоколадица у пакетима треба најпре сортирати (најбоље помоћу библиотечке функције `sort`, као што је приказано у задатку Сортирање бројева), и затим одредити минимум разлика вредности $a_{i+k-1} - a_i$, за $0 \leq i \leq n-k$ (коришћењем уобичајеног алгоритма за налажење минимума, приказаног, на пример, у задатку Најнижа температура).

Сложеношћу овог алгоритма доминира сложеност корака сортирања, а она је $O(n \cdot \log(n))$, ако се користе библиотечке имплементације. Након сортирања, минимум се одређује $n-k$ корака, тј. у линеарној сложености $O(n)$ (када је k мало број корака може бити веома близак n).

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

using namespace std;

int main() {
  // ucitavamo brojeve cokoladica u paketima
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  // ucitavamo broj dece
  int k;
  cin >> k;

  // sortiramo pakete po broju cokoladica
  sort(begin(a), end(a));

  // odredjujemo i ispisujemo najmanju mogucu razliku za nekih k
  // odabranih paketa
  int min = numeric_limits<int>::max();
  for (int i = 0; i + k - 1 < n; i++) {
    int razlika = a[i + k - 1] - a[i];
    if (razlika < min)
      min = razlika;
  }
  cout << min << endl;

  return 0;
}
```

2.14.3 Свођење на канонски облик

Често имамо потребу да проверимо да ли су два низа елемената једнака, ако се занемари у ком су редоследу елементи наведени. Класичан пример овога је провера да ли се једна реч може добити пермутовањем слова друге (тј. да ли су анаграми). У суштини, ради се о поређењу два мутлискупа елеманата (која су задата низовима). Најбољи начин да се оваква једнакост провери је да се оба низа сведу на неки канонски облик, који неће зависити од редоследа елемената низа. Најједноставнији начин да се такав канонски облик добије је да се елементи низа сортирају пре поређења. Други начин је да се изврши пребројавање свих елемената тј. да се мултискупови представе пресликавањима сваког елемента у његов број појављивања (два таква пресликавања су једнака ако и само ако исти скуп кључева слика у исте вредности).

Задатак: Анаграм

Дате су две ниске сачињене од малих слова енглеске абецеде, интерпукцијских знакова и размака. Напиши програм који проверава да ли су два дата стринга анаграми, тј. да ли се од прве ниске премештањем слова може добити други друга ниска и обрнуто (карактери који нису слова се занемарују).

Улаз: Прва линија станадардног улаза садржи једну ниску, а друга линија садржи другу ниску.

Излаз: На стандардном излазу у једној линији приказати реч `da` ако дате ниске представљају анаграме, у супротном приказати реч `ne`.

Пример1

Улаз	*Излаз*
`panta redovno zakasni` `neopravdan izostanak`	`da`

Пример2

Улаз	*Излаз*
`oni su skrsili vagu` `suvisni kilogrami`	`ne`

Решење

Две ниске су анаграм ако се једна може добити пермутовањем редоследа карактера друге, па је овај задатак веома сличан задатку Провера пермутација и могу се применити све технике које су у њему описане. Једина разлика је у томе што се приликом провере анаграма не узимају у обзир сви карактери, већ само мала слова.

Сортирање

Један начин да се задатак реши је да се обе ниске сортирају и онда упореде. Сортирање ниски може да се изврши библиотечком функцијом. У језику C++ то је функција `sort`. У језику C# то је функција `Array.Sort`. Међутим, пошто се проверавају само слова, пре сортирања и поређења, из ниски је потребно избацити све карактере који нису слова. У језику C++ то је могуће урадити функцијом `copy_if`. Пошто не знамо унапред колико има малих слова, ниску која садржи резултат ћемо проширивати додавањем елемената на крај (за шта користимо `back_inserter`).

Сложеност овог алгоритма зависи од сложености сортирања и ако се користи библиотечка функција сортирања износи $O(n \log n)$. Наиме, копирање ниски (које је неопходно, да се сортирањем не би покварила оригиналне ниске, али и да би се филтрирала само мала слова) и поређење једнакости након сортирања су линеарне сложености, па сортирање узима највише времена.

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

bool anagram(const string& s1, const string& s2) {
  string slova1, slova2;
  copy_if(begin(s1), end(s1), back_inserter(slova1), ::islower);
  copy_if(begin(s2), end(s2), back_inserter(slova2), ::islower);
  sort(begin(slova1), end(slova1));
  sort(begin(slova2), end(slova2));
  return slova1 == slova2;
}

int main() {
  string s1, s2;
  getline(cin, s1);
```

```
  getline(cin, s2);
  if (anagram(s1, s2))
    cout << "da" << endl;
  else
    cout << "ne" << endl;
  return 0;
}
```

2.14.4 Остале примене сортирања

Наведени примери сортирања сигурно не исцрпљују све употребе сортирања. У наставку ће бити приказано још неколико задатака који могу ефикасно бити решени захваљујући примени сортирања.

Задатак: Хиршов h-индекс

Рангирање научника врши се помоћу статистике која се назива *Хиршов индекс* (скр. *h-индекс*). H-индекс научника је највећи број h такав да научник има бар h радова од којих сваки има бар h цитата.

Улаз: Са стандардног улаза се уноси број n ($1 \leq n \leq 5 \cdot 10^4$) који представља број радова научника а затим и n природних бројева који представљају број цитата (између 0 и 10^6) за сваки од тих n радова.

Излаз: На стандардни излаз исписати један природан број који представља h-индекс научника.

Пример

Улаз	*Излаз*
8 3 5 12 7 5 9 0 17	5

Објашњење

Постоји тачно 5 радова са бар 5 цитата (5, 12, 9, 7, 17). Преостали радови имају 3, 5 и 0 цитата, тако да не постоји 6 радова са бар 6 цитата.

Решење

Груба сила

H-индекс се може дефинисати као највећи број h такав да постоји бар h радова који имају бар h цитата. Могуће вредности за h индекс су бројеви између 0 и n. Основни начин да одредимо h индекс је да употребимо линеарну претрагу и да за сваку вредност h између 0 и n проверимо да ли постоји бар h радова са бар h цитата тј. да ли је $B_h \geq h$, где B_h представља broj radova sa h или више цитата. Анализираћемо разне вредности h, на пример, од n па наниже, све док не наиђемо на прву вредност h за коју постоји бар h радова са h цитата тј. за коју је је $B_h \geq h$ и та вредност h представљаће тражени h-индекс.

За сваки број h израчунаћемо вредност B_h. Наиван начин да се то уради је да се да се прође кроз низ бројева цитата c (који не мора бити сортиран) и да се одреди број елемената низа који су већи или једнаки h (алгоритам бројања филтриране серије сличан је оном у задатку Просек одличних).

Пошто се бројање радова за сваку вредност h изводи изнова и захтева линеарну сложеност $O(n)$, а пошто је број различитих вредности h у најгорем случају једнак n, укупна сложеност најгорег случаја овог приступа је $O(n^2)$.

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

using namespace std;

int main() {
  // broj clanaka
  int n;
  cin >> n;

  vector<int> broj_citata(n);
```

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

  // smanjujemo h-indeks sve dok je broj clanaka koji imaju bar
  // h-indeks citata manji od vrednosti h-indeksa
  int h_indeks = n;
  while (count_if(begin(broj_citata), end(broj_citata),
                  [h_indeks](int c) {
                    return c >= h_indeks;
                  }) < h_indeks)
    h_indeks--;

  cout << h_indeks << endl;

  return 0;
}
```

Сортирање

Један ефикасан начин одређивања h-индекса може бити заснован на сортирању. На пример, ако сортирамо низ цитата `3 5 12 7 5 9 0 17` нерастуће, добијамо низ `17 12 9 7 5 5 3 0`. Анализирајмо један по један рад у овом низу. Важи да је $c_0 = 17 \geq 1$, па имамо бар један рад са бар једним цитатом, $c_1 = 12 \geq 2$, па имамо бар два рада са бар два цитата, $c_2 = 9 \geq 3$, па имамо бар 3 рада са бар 3 цитата, $c_3 = 7 \geq 4$, па имамо бар 4 рада са бар 4 цитата, $c_4 = 5 \geq 5$, па имамо бар 5 радова са бар 5 цитата, али не и $c_5 = 5 \geq 6$, па немамо бар 6 радова са бар 6 цитата.

Дакле, ако радове сортирамо у нерастући низ c по броју цитата, пошто се позиције броје од 0, за радове који задовољавају h-услов важи да је $c_h \geq (h+1)$ тј. $c_h > h$. H-индекс можемо израчунати као најмањи број h такав да је $c_h \leq h$.

Заиста ако је неки рад на позицији h у сортираном низу и ако он има c_h цитата, знамо да постоји бар $h+1$ рад који има бар c_h цитата (јер он има бар c_h цитата и испред њега постоји тачно h радова који имају бар онолико цитата колико и он, јер је низ нерастући). У нерастуће сортираном низу проверавамо све индексе h од 0 до $n-1$, све док не наиђемо на први за који важи да је $c_h \leq h$. Када се то деси, знамо да постоји тачно h радова који имају бар h цитата, јер је за претходну позицију важило да је $c_{h-1} > h-1$, тј. $c_{h-1} \geq h$, па је постојало тачно h радова са бар c_h цитата, а то је бар h цитата. Такође, не постоји $h+1$ рад са бар $h+1$ цитатом, јер сви радови од позиције h до краја низа имају строго мање од $h+1$ цитата, а испред њих постоји само h радова.

Сложеност овог алгоритма долази пре свега од корака сортирања и када се сортира библиотечком функцијом једнака је $O(n \cdot \log n)$ (наиме, након сортирања индекс се одређује једним проласком кроз низ, у највише $O(n)$ додатних корака).

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

using namespace std;

int main() {
  // ucitavamo niz brojeva citata za svaki rad
  int n;
  cin >> n;
  vector<int> broj_citata(n);
  for (int i = 0; i < n; i++)
    cin >> broj_citata[i];

  // sortiramo radove na osnovu broju citata, opadajuce
  sort(broj_citata.begin(), broj_citata.end(), greater<int>());
```

```
  // Ako je h-indeks jednak h, tada h-ti po redu casopis ima bar h
  // citata. Trazimo najveci broj koji zadovoljava taj uslov.
  int h_indeks = 0;
  while (h_indeks < n && broj_citata[h_indeks] > h_indeks)
    h_indeks++;

  // ispisujemo rezultat
  cout << h_indeks << endl;

  return 0;
}
```

2.15 Бинарна претрага

Ако немамо никакве информације о редоследу елемената у низу, једини начин да проверимо да ли се у њему налази неки елемент је да употребимо линеарну претрагу и да редом проверавамо један по један елемент низа. У најгорем случају сваки елемент низа мора бити прегледан, па је сложеност таквог приступа $O(n)$, где је n број елемената низа.

Ако је низ елемената сортиран, претрагу је могуће извршити много ефикасније, у сложености $O(\log n)$, коришћењем алгоритма **бинарне претраге**, којим се, услед сортираности низа, одсеца значајан део простора претраге и тако добија на ефикасности. Она се може применити на много сродних проблема.

У основној варијанти, бинарна претрага (БП) служи да се провери да ли сортиарани низ елемената садржи неку дату вредност.

Поред овога, БП се може употребити да се пронађе први елемент у сортираном низу који је (било строго, било нестрого) већи или мањи од датог.

У свом најопштијем облику бинарна претрага се користи да се у низу пронађе преломна тачка тј. први елемент који задовољава неки услов (под претпоставком да су елементи поређани тако да у низу прво иду сви елементи који тај услов не задовољавају, а затим они који тај услов задовољавају).

Бинарну претрагу ћемо употребљавати и за оптимизацију, тј. да пронађемо најмању или највећу вредност, која задовољава одређени услов.

2.15.1 Бинарна претрага елемента у низу

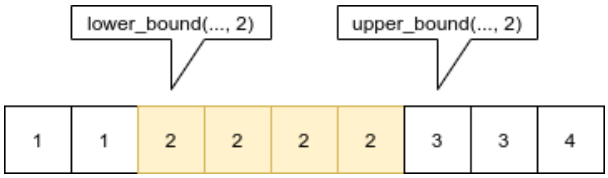
Алгоритам бинарне претраге вредности у низу одговара оном који се може применити у игри погађања непознатог броја и који је заснован на половљењу интервала. Основна идеја је да се тражени елемент пореди са средишњим елементом у низу. Ако је тражени елемент мањи од средишњег, пошто је низ сортиран, знаћемо да је мањи и од свих елемената десно од тог средишњег, па тај део низа можемо елиминисати (можемо начинити одсецање у претрази) и претрагу можемо наставити само у левој половини низа. Симетрично, ако је тражени елемент већи од средишњег, због сортираности је већи и од свих елемената лево од средишњег и лева половина низа може бити елиминисана из даље претраге. На крају, ако елемент није ни мањи ни већи од средишњег, онда му је једнак и пронађен је у низу. Приметимо да у основи алгоритма бинарне претраге лежи техника одсецања, која је оправдана тиме што је низ сортиран.

Пошто се у сваком кораку претраге дужина низа дупло смањује, за претрагу целог низа довољно је $O(\log n)$ корака, где је n дужина низа. Наиме, претрага у најгорем случају траје све док се половљењем низ не испразни. Дужина низа након k корака половљења је отприлике $\frac{n}{2^k}$. Низ ће се испразнити када је $\frac{n}{2^k} < 1$, тј. када је $n < 2^k$, тј. када је $k > \log_2 n$.

Слично као и за сортирање, већина програмских језика пружа готове библиотечке функције за бинарну претрагу.

У језику C++ функција `binary_search` проверава да ли дати распон елемената (задат помоћу два итератора) садржи задату вредност (функција враћа `true` ако и само ако се тражени елемент налази унутар задатог распона). Тако се провера да ли се дати елемент `x` налази унутар сортираног вектора `a` може извршити помоћу `binary_search(begin(a), end(a), x)`.

Поред ове, постоје још три функције које врше одређене варијације бинарне претраге. Ако је потребно пронаћи све елементе једнаке датом, можемо користити функцију `equal_range` (са истим параметрима као `binary_search`). Она враћа пар итератора који ограничавају распон елемената једнаких датом (први итератор указује на први елемент једнак траженој вредности, а други на позицију непосредно иза последњег елемента једнаког траженој вредности). Функција `lower_bound` враћа први од та два итератора (тј. први елемент који је већи или једнак од тражене вредности), а функција `upper_bound` враћа други од њих (тј. први елемент који је строго већи од тражене вредности).



Слика 2.21: lower_bound и upper_bound

О њима и њиховој употреби ће бити више речи у наредним поглављима.

Ако се не зада другачије, подразумева се да је низ сортиран у односу на подразумевани поредак елемената (неопадајући нумерички ако су бројеви у питању, тј. неопадајући абецедни лексикографски ако су ниске у питању). Поредак се може задати или променити на сличан начин као код функција за сортирање (о чему ће бити више речи касније).

Задатак: Провера бар-кодова

У продавници се налази пуно врста производа и познати су њихови бар-кодови. Произвођач жели да сазна колико се врста његових производа продаје у тој продавници. Ако је списак свих кодова производа у продавници дат у сортираном облику, а списак свих кодова производа произвођача је достављен несортиран, напиши програм који одређује тражени број.

Улаз: Са стандардног улаза учитава се број n ($1 \leq n \leq 50000$), а n природних бројева (највише шестоцифрених), раздвојених размацима. Ти бројеви представљају бар-кодове производа у продавници и сортирани су растуће. Након тога се до краја улаза учитавају бар-кодови производа које је произвођач доставио (највише шестоцифрени природни бројеви, сваки у посебном реду).

Излаз: На станадрни излаз исписати број производа произвођача који се већ продају у продавници.

Пример

Улаз	*Излаз*
5 1 3 5 6 7 2 3 4 5 8	2

Решење

Линеарна претрага

Наиван начин да проверимо да ли елемент постоји у низу је примена алгоритма линеарне претраге (попут оног, на пример, у задатку Негативан број). Линеарна претрага може бити било имплементирана ручно, било реализована помоћу библиотечких функција. У језику C++ функцијом `find` можемо проверити да ли низ садржи дати елемент.

Пошто се у најгорем случају линеарном претрагом (било библиотечком, било ручно-имплементираном) анализира сваки елемент низа, сложеност претраге једног елемента је $O(n)$. Ако претпоставимо да се претражује m елемената, укупна сложеност алгоритма је $O(mn)$.

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

```cpp
#include <vector>

using namespace std;

// funkcija proverava da li se u datom sortiranom vektoru a nalazi
// element x
bool sadrzi(const vector<int>& a, int x) {
  // proveravamo sve elemente vektora a
  for (int i = 0; i < a.size(); i++)
    // nasli smo element x na poziciji i
    if (a[i] == x)
      return true;
  // element x nije nadjen
  return false;
}

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

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

  // brojac pojavljivanja elemenata
  int broj = 0;
  // ucitavamo element po element do kraja ulaza
  int x;
  while (cin >> x) {
    // ako je ucitani element sadrzan u nizu a, uvecavamo brojac
    if (sadrzi(a, x))
      broj++;
  }
  // ispisujemo rezultat
  cout << broj << endl;
  return 0;
}
```

Бинарна претрага

Чињеница да је низ сортиран нам даје начин да задатак решимо много ефикасније, применом бинарне претраге.

Низ од n елемената се бинарном претрагом може претражити у сложености $O(\log n)$, па m производа можемо претражити у сложености $O(m \log n)$.

Бибилотечке функције

У језику C++ функција `binary_search` изводи бинарну претрагу неког сегмента (низа или вектора). Аргументи су два итератора (један на први елемент сегмента који се претражује, а други непосредно иза последњег елемента) и елемент који се тражи. Функција враћа `bool` чија је вредност `true` ако и само ако елемент постоји у низу.

Библиотечка функција гарантује да ће сортиран низ од n елемената бити претражен у сложености $O(\log n)$ (па се m елемената независно претражује у $O(m \log n)$ корака).

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

```cpp
using namespace std;

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

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

  // broj onih koji postoje u nizu
  int broj = 0;
  // ucitavamo broj po broj do kraja ulaza
  int x;
  while (cin >> x) {
    // ako je broj sadrzan u nizu, uvecavamo brojac
    if (binary_search(a.begin(), a.end(), x))
      broj++;
  }
  // ispisujemo rezultat
  cout << broj << endl;

  return 0;
}
```

Итеративна имплементација

Алгоритам бинарне претраге вредности у низу одговара оном који се може применити у игри погађања непознатог броја и који је заснован на половљењу интервала.

Претпоставимо да желимо проверити да ли се елемент x налази у низу a између позиција l и d тј. да ли се налази у затвореном интервалу позиција $[l, d]$. Ако је интервал празан, он не садржи елемент x. У супротном пронађимо средину овог интервала s (а то је проблем који смо дискутовали у задатку Средина интервала).

- Ако је $x < a[s]$, пошто је низ сортиран, важи да је x мањи и од свих елемената који су десно од s. Зато претрагу можемо наставити у интервалу $[l, s-1]$.
- Слично, ако је $x > a[s]$, пошто је низ сортиран, важи да је x веће од свих елемената лево од s тако да претрагу можемо наставити само у интервалу $[s+1, d]$.
- На крају, ако је $x = a[s]$ тада смо елемент пронашли у низу и то на позицији s. Претрага се може прекинути и када се интервал који се претражује испразни (што ће се десити ако је $l > d$).

Алгоритам можемо имплементирати итеративно. Промењиве l и d иницијализујемо на 0 и $n-1$, затим у петљи која се извршава док је $l \leq d$ проналазимо s и поредимо $a[s]$ са x. Ако је $x < a[s]$ тада d постављамо на $s-1$, ако је $x > a[s]$ тада l постављамо на $s+1$, а ако је $x = a[s]$ прекидамо функцију (и петљу) информишући позиваоца да је елемент нађен на позицији s. Ако се петља заврши, елемент не постоји у низу.

Докажимо формално коректност овог алгоритма. Инваријанта петље је да су:

- сви елементи у интервалу $[0, l)$ строго мањи од x,
- сви елементи у интервалу (d, n) строго већи од x.

Уз то важи и $0 \leq l \leq d+1 \leq n$.

Иницијализацијом $l = 0$ и $d = n-1$, инваријанта је задовољена.

Претпоставимо да инваријанта важи при уласку у петљу, након провере услова $l \leq d$. Тада израчунавамо средину интервала s (за њу сигурно важи $l \leq s \leq d$).

- Ако је $x < a_s$, тада је $l' = l$, $d' = s-1$. Пошто је $a_s > x$, услед сортираности низа, већи од x су и сви елементи на позицијама из интервала $(d', n) = (s-1, n) = [s, n)$. Пошто се вредност промењиве `l`

није променила, сви елементи на позицијама $[0, l')$ су сигурно строго мањи од x (што знамо да важи на основу претпоставке).

- Ако је $x > a_s$, тада је $l' = s + 1$, $d = d$. Пошто је $a_s < x$, услед сортираности низа, мањи од x су и сви елементи на позицијама из интервала $[0, l') = [0, s + 1) = [0, s]$. Пошто се вредност промењиве `d` није променила, сви елементи на позицијама (d', n) су сигурно строго већи од x (што знамо да важи на основу претпоставке).
- Ако је $x = a_s$, пронађен је тражени елемент и функција коректно потврђује да низ садржи елемент x.

Када се петља заврши важи инваријанта, али услов $l \leq d$ није испуњен. Пошто је $l \leq d+1$, важи да је $l = d+1$. Стога су сви елементи у интервалу $[0, l)$ строго мањи од x а сви елементи у интервалу $(d, n) = [d+1, n) = [l, n)$ строго већи од x. Дакле, сви елементи низа су или строго мањи или строго већи од x и стога низ не садржи елемент x.

Алгоритам се сигурно зауставља, јер се у сваком кораку петље ширина интервала $[l, d]$, која је једнака $d-l+1$ строго смањује, све док не достигне нулу.

Пошто се у сваком кораку интервал $[l, d]$ полови, пошто иницијално крећемо од интервала $[0, n-1]$ и пошто се претрага завршава када се интервал испразни, сложеност претраге једног елемента је $O(\log n)$ (па се m елемената независно претражује у $O(m \log n)$ корака).

```
#include <iostream>

using namespace std;

// funkcija proverava da li se u datom sortiranom nizu a duzine n
// nalazi element x
bool sadrzi(int a[], int n, int x) {
  // petrazujemo da li se element nalazi u intervalu [l, d]
  int l = 0, d = n - 1;
  // dok god taj interval nije prazan
  while (l <= d) {
    // nalazimo sredinu intervala
    int s = l + (d - l) / 2;

    // ako je x manji od srednjeg on se moze nalaziti samo u intevalu
    // [a, s-1] (jer je niz sortiran)
    if (x < a[s])
      d = s - 1;
    // ako je x veci od srednjeg on se moze nalaziti samo u intevalu
    // [s+1, d] (jer je niz sortiran)
    else if (x > a[s])
      l = s + 1;
    else
      // nasli smo element x na poziciji s
      return true;
  }
  // element ne postoji u nizu
  return false;
}

// maksimalni broj elemenata niza dat tekstom zadatka
const int MAX = 50000;

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

  // ucitavamo niz
  int n;
  cin >> n;
```

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

  // broj onih koji postoje u nizu
  int broj = 0;
  // ucitavamo broj po broj do kraja ulaza
  int x;
  while (cin >> x) {
    // ako je broj sadrzan u nizu, uvecavamo brojac
    if (sadrzi(a, n, x))
      broj++;
  }
  // ispisujemo rezultat
  cout << broj << endl;
  return 0;
}
```

Види другачија решења овог задатка.

Задатак: Број парова датог збира

Дат је цео број s и низ различитих целих бројева. Написати програм којим се одређује број парова у низу који имају збир једнак датом броју s.

Улаз: У првој линији стандардног улаза налази се цео број s (број из интервала $[0, 10^6]$), у другој линији налази се број елемената низа n ($1 \leq n \leq 50000$), а у следећих n линија налази се редом елементи низа (бројеви из интервала $[0, 10^6]$).

Излаз: На стандардном излазу приказати број парова различитих елемената низа чији је збир једнак броју s.

Пример

Улаз	*Излаз*
5 6 1 4 3 6 -1 5	2

Решење

Бинарна претрага

Уместо посебне колекције скупа, скуп елемената може бити репрезентован сортираним низом који претражујемо бинарном претрагом (коју смо објаснили у задатку Провера бар-кодова). Да бисмо могли да применимо овај приступ, потребно је да претходно низ a сортирамо у растућем поретку. Елемент $a_j = s - a_i$ можемо тражити у низу почев од позиције $i+1$, јер на тај начин нећемо исти пар бројати два пута. Такође приметимо да је важан услов, дат у задатку, да су елементи низа различити, јер класичном бинарном претрагом налазимо једно појављивање елемента у низу или установимо да тај елемент не постоји, а нама се тражи број парова па је важно колико пута се неки елемент појављује у низу а не само да ли се појављује.

У језику C++ можемо да проверимо да ли тражена вредност постоји у низу помоћу функције `binary_search` (која враћа логичку вредност).

Сложеност бинарне претраге је $O(\log n)$, у зависности од дужине дела низа који се претражује, а пошто ми претрагу вршимо за сваки од n елемената низа, може се показати да ће укупна сложеност бити $O(n \log n)$. Пошто се не претражује увек цео низ, већ само део од текућег елемента до краја низа, сложеност провере ће заправо бити $O(\log(n-1) + \log(n-2) + \ldots + \log 1)$, што је $O(\log(n-1)!)$ и за то је могуће показати да је $O(n \log n)$.

Бинарна претрага би се могла унапредити тако што се би се у сваком наредном кораку вршила претрага за вредношћу $s - a_{i+1}$ само до позиције првог елемента који је био већи или једнак вредности $s - a_i$. Наиме, пошто је $a_{i+1} > a_i$, тај елемент и сви елементи иза њега су сигурно строго већи од вредности $s - a_{i+1}$ (која је строго мања од $s - a_i$). Претрагу можемо прекинути када се лева и десна граница претраге сустигну (тј. када се та десна позиција повуче на лево, све до текуће вредности i).

Овом модификацијом се не би добило асимптотски значајно убрзање, па је нећемо имплементирати.

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

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

  sort(begin(a), end(a));

  int brojParova = 0;
  for (int i = 0; i < n - 1; i++)
    if (binary_search(next(begin(a), i + 1), end(a), s - a[i]))
      brojParova++;

  cout << brojParova << endl;

  return 0;
}
```

Види другачија решења овог задатка.

Задатак: i-ти на месту i

Напиши програм који проверава да ли у строго растућем низу елемената постоји позиција i таква да се на позицији i налази вредност i тј. да важи да је $a_i = i$ (позиције се броје од нуле).

Улаз: Са стандардног улаза се уноси број n ($0 \leq n \leq 10^5$), а затим и строго растући низ од n целих бројева (сваки у посебном реду).

Излаз: На стандардни излаз исписати индекс i такав да је $a_i = i$ или текст `nema` ако такав индекс не постоји у низу. Ако у низу постоји више таквих индекса исписати најмањи од њих.

Пример

Улаз	*Излаз*
6 -3 -1 1 3 5 7	3

Решење

Линеарна претрага

Директан начин да се задатак реши је да се употреби линеарна претрага (попут оне у задатку Негативан број) и да се позиције проверавају редом, од 0 до $n - 1$ све док се не пронађе прва позиција која задовољава услов или док се не дође до краја низа.

Сложеност линеарне претраге је $O(n)$.

```
#include <iostream>

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int n;
  cin >> n;
  int i;
  for (i = 0; i < n; i++) {
    int x;
    cin >> x;
    if (x == i) {
      cout << i << endl;
      break;
    }
  }

  if (i >= n)
    cout << "nema" << endl;

  return 0;
}
```

Бинарна претрага трансформисаног низа

Размотримо низ `-10 -4 1 3 4 9 11`. Елемент `-10` је мањи од своје позиције 0 за 10. Елемент `-4` је мањи од своје позиције 1 за 5, елемент `1` је мањи од своје позиције 2 за 1. Елементи `3` и `4` су једнаки својим позицијама. Елемент `9` је већи од своје позиције 5 за 4 док је елемент `11` већи од своје позиције 6 за 5. Примећујемо одређену монотоност у овом низу, што није случајно. Заиста, ако је $a_i = i$, тада је $a_i - i = 0$. Покажимо да је низ $a_i - i$ неопадајући. Посматарајмо два елемента a_i и a_j на позицијама на којима је $0 \leq i < j$. Пошто је низ a строго растући, важи да је $a_{i+1} > a_i$, па је $a_{i+1} \geq a_i + 1$. Слично је $a_{i+2} > a_{i+1}$, па је $a_{i+2} \geq a_{i+1} + 1 \geq a_i + 2$. Настављањем овог резона важи да је $a_j \geq a_i + j$. Зато је $a_j - j \geq a_i \geq a_i - i$. Решење, дакле, можемо одредити тако што бинарном претрагом проверимо да ли неопадајући низ $a_i - i$ садржи нулу и ако садржи, тада је решење позиција на којој се та нула налази.

Један од најлакших начина да реализујемо бинарну претрагу је да употребимо библиотечку функцију. Пошто нам је потребна прва позиција нуле у трансформисаном низу, не можемо употребити функцију `binary_search`, већ морамо употребити функцију `lower_bound`, слично као у задатку Ранг сваког елемента.

Сложеност бинарне претраге је $O(\log n)$, међутим, временом доминира учитавање и трансформисање низа које захтева $O(n)$ корака.

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

using namespace std;

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

```
  // pripremamo ga za pretragu a[k] = k akko je a[k] - k = 0 tako da
  // umesto niza a, pretrazujemo niz a[i] - i (koji je neopadajuci)
  for (int i = 0; i < n; i++)
    a[i] -= i;

  // trazimo poziciju nule u transformisanom nizu tako sto pronalazimo
  // poziciju prvog elementa koji je >= 0
  auto it = lower_bound(a.begin(), a.end(), 0);

  // ako takav element postoji i ako je jednak nuli
  if (it != a.end() && *it == 0)
    // pronasli smo element i izracunavamo njegovo rastojanje od pocetka niza
    cout << distance(a.begin(), it) << endl;
  else
    // u suprotnom element ne postoji u nizu
    cout << "nema" << endl;

  return 0;
}
```

2.15.2 Бинарна претрага преломне тачке

У свом најопштијем облику, бинарна претрага се може формулисати на следећи начин. Размотримо низ елемената такав да су елементи подељени у две групе, на основу неког својства P. Елементи у почетном делу низа су сви такви да немају то својство P, а елементи у завршном делу низа су сви такви да имају својство P. Низ је, дакле, облика `----++++++++`, где су са `-` означени елементи који немају, а са `+` елементи који имају својство P. Могућа је и ситуација у којој је нека од група празна. Својство P може бити сасвим произвољно. На пример, у сортираном низу бројева можемо посматрати својство *већи је или једнак* X за неку дату вредност X. Тада се у првом делу низа налазе елементи који нису већи или једнаки X, тј. строго су мањи од X, док се иза њих налазе елементи који имају својство, тј. већи су или једнаки X. Даље, на пример, низ ученика може бити организован тако да су прво наведени дечаци, а затим девојчице.

Бинарна претрага нам може помоћи да ефикасано одредимо *преломну тачку*, тј. место где престаје једна и почиње друга група елемената. То може бити или позиција последњег елемента који нема својство P или првог елемента који има својство P. Ако сви елементи низа имају својство P, тада претрага за позицијом последњег елемента низа који нема својство треба да врати -1. Ако ниједан елемент низа нема својство P, тада претрага за позицијом првог елемента низа који има својство P треба да врати дужину низа. Познавање преломне тачке нам омогућава и да ефикасно одговоримо на питање колико је елемената у свакој групи (колико елемената низа нема, а колико елемената низа има својство P).

Класична бинарна претрага се лако формулише као претрага преломне тачке. Ако у низу пронађемо позицију првог елемента који је већи или једнак траженој вредности X, тада можемо проверити да ли је та позиција унутар низа (строго мања од дужине низа) и да ли се на њој налази елемент X - ако је то испуњено елемент постоји у низу, а у супротном не постоји.

У наставку ће кроз низ задатака бити приказан алгоритам бинарне претраге преломне тачке у низу. У првим примерима у низу бројева ћемо тражити позицију првог елемента који је (строго или нестрого) већи или мањи од дате вредности, док ћемо у наредним примерима посматрати и другачије низове, подељене у односу на неко својство P.

Позиција преломне тачке може бити пронађена и уз помоћ инвентивне употребе функција `lower_bound` и `upper_bound`.

Задатак: Провера бар-кодова

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

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

Решење

Бинарна претрага првог елемента који је већи или једнак од траженог

Један начин да се реализује бинарна претрага којом се проверава да ли елемент постоји у низу је да се пронађе позиција првог елемента који је већи или једнак траженом елементу x. Број x се јавља у низу ако и само у низу постоји елемент који је већи или једнак x и први такав елемент је управо x. Овај приступ је описан (у задацима Ранг сваког елемента и i-ти на месту i), у којима је описана употреба функције `lower_bound` која проналази позицију првог елемента који је већи или једнак од датог.

Опишимо имплментацију без употребе библиотечких функција. Први елемент који је већи или једнак од елемента x можемо наћи слично као у задатку Први већи и последњи мањи.

Уведимо променљиве l и d и наметнимо услов (инваријанту) да све време током претраге важи $0 \leq l \leq d+1 \leq n$, и да су

- елементи низа a на позицијама из интервала $[0, l)$ мањи од x,
- елементи на позицијама из интервала (d, n) већи или једнаки x.

Елементима на позицијама из интервала $[l, d]$ статус још није познат. Овај услов ће бити иницијализован ако се променљива l иницијализује на нулу, а променљива d на вредност $n-1$.

Нека s представља средину интервала $[l, d]$. Важи $l \leq s \leq d$.

- Ако је елемент низа a на позицији s већи или једнак вредности x тада су, услед сортираности низа, и сви елементи интервала $[s, n)$ су већи или једнаки x. Зато можемо вредност d поставити на $s-1$ и инваријанта ће остати да важи. Заиста, важи да је $d' = s-1$, па су елементи из интервала $(d', n) = (s-1, n) = [s, n)$ већи или једнаки x, док су сви елементи из интервала $[0, l') = [0, l)$ строго мањи од x, што знамо на основу претпоставке. Услов $0 \leq l' \leq d'+1 \leq n$, еквивалентан је услову $0 \leq l \leq s \leq n$, што сигурно важи, јер је $l \leq s \leq d$ и $0 \leq l \leq d+1 \leq n$.
- Ако је елемент низа a на позицији s строго мањи од вредности x такви су, услед сортираности низа, и сви елементи из интервала $[0, s]$. Зато можемо вредност l поставити на $s+1$ и инваријанта ће остати да важи. Заиста, важи да је $l' = s+1$, па су сви елементи из интервала $[0, l') = [0, s+1) = [0, s]$ строго мањи од x, док су сви елементи из интервала $(d', n) = (d, n)$ већи или једнаки од x, што знамо на основу претпоставке. Услов $0 \leq l' \leq d'+1 \leq n$, еквивалентан је услову $0 \leq s+1 \leq d+1 \leq n$, што сигурно важи, јер је $l \leq s \leq d$ и $0 \leq l \leq d+1 \leq n$.

Претрагу вршимо све док постоје елементи непознатог статуса, тј. док је интервал $[l, d]$ непразан, односно док је $l \leq d$. У тренутку када је $l > d$, када се претрага заврши, важи да је $l = d+1$. На основу инваријанте знамо да се први елемент већи или једнак x налази на позицији $l = d+1$, јер су сви елементи у интервалу $(d, n) = [d+1, n)$ већи или једнаки од x. Изузетак је случај када је $l = n$, када су сви елементи низа a строго мањи од x.

Тражени елемент x постоји у низу ако и само ако је $l = d+1 < n$ и ако је $a_l = x$.

Алгоритам се сигурно зауставља јер се у сваком кораку ширина интервала $[l, d]$ која је једнака $d-l+1$ строго смањује, док не достигне нулу.

Приметимо да овим алгоритмом пролазимо позицију првог појављивања елемента x у низу, док алгоритмом који проверава једнакост током претраге проналазимо било коју позицију. Пошто је често потребно наћи баш прво појављивање елемента, овај алгоритам је јасно у предности.

Пошто се у сваком кораку интервал $[l, d]$ полови, пошто иницијално крећемо од интервала $[0, n-1]$ и пошто се претрага завршава када се интервал испразни, број потребних корака да се то догоди је $O(\log n)$. Пошто се независно претражује m елемената, укупна сложеност је $O(m \log n)$. Приметимо да за разлику од варијанте у којој се унутар тела петље проверава и услов једнакости, за шта су потребна два поређења, у овом алгоритму се врши само једно поређење у телу петље, па је константни фактор мало мањи (што је често занемариво).

```
#include <iostream>

using namespace std;

// funkcija proverava da li se u datom sortiranom nizu a duzine n
// nalazi element x
bool sadrzi(int a[], int n, int x) {
  // trazimo poziciju prvog elementa u nizu a koji je veci ili jednak x

  // [0, l) - elementi strogo manji od x
```

```
  // (d, n) - elementi veci ili jednaki x
  // [l, d] - nepoznati elementi
  int l = 0, d = n - 1;
  // dok god taj interval nije prazan
  while (l <= d) {
    // nalazimo sredinu intervala
    int s = l + (d - l) / 2;

    if (a[s] >= x)
      // posto je niz sortiran svi posle pozicije s-1 su veci ili jednaki x
      d = s - 1;
    else
      // posto je niz sortiran svi pre pozicije s+1 su strogo manji od x
      l = s + 1;
  }
  // prvi veci ili jednak nalazi se na poziciji d+1 = l

  // x je u nizu ako i samo ako u nizu postoji element koji je veci
  // ili jednak x i ako je prvi takav element upravo x
  return l < n && a[l] == x;
}

// maksimalni broj elemenata niza dat tekstom zadatka
const int MAX = 50000;

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

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

  // broj onih koji postoje u nizu
  int broj = 0;
  // ucitavamo broj po broj do kraja ulaza
  int x;
  while (cin >> x) {
    // ako je broj sadrzan u nizu, uvecavamo brojac
    if (sadrzi(a, n, x))
      broj++;
  }
  // ispisujemo rezultat
  cout << broj << endl;
  return 0;
}
```

Задатак: Први који није дељив

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

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

Решење

Линеарна претрага

Наивно решење може бити засновано на примени линеарне претраге (бројању елемената филтриране серије) да би се одредило колико у низу постоји елемената дељивих са учитаним делиоцем.

Ако низ има n елемената, а постоји m делилаца, сложеност овог решења је $O(mn)$. Приметимо да ово решење ни на који начин не користи особину низа да прво иду елементи који су дељиви, а онда елементи који нису дељиви датим делиоцем.

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

using namespace std;

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

  int n;
  cin >> n;
  vector<long long> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  long long d;
  while (cin >> d) {
    auto deljiv = [d](long long x) {return x % d == 0;};
    cout << count_if(begin(a), end(a), deljiv) << '\n';
  }
  return 0;
}
```

Бинарна претрага

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

Ручно имплементирана бинарна претрага

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

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

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

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

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

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

Прикажимо рад алгоритма на једном примеру.

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

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

          ld
1 7 3 5 9 11 2 8 6
          s

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

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

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

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

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

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

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

Пошто се у сваком кораку претраге ширина интервала $[l, d]$ преполови, пошто се иницијално креће од интервала $[0, n-1]$ који има n елемената и пошто се алгоритам завршава када се интервал испразни, сложеност алгоритма је $O(\log n)$. Наиме, дужина интервала после k корака је $\lfloor \frac{n}{2^k} \rfloor$ и важи да је $\lfloor \frac{n}{2^k} \rfloor < 1$ када је $k > \log_2 n$.

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

using namespace std;

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

  int n;
```

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

Исправљање грешака на основу формалне анализе кода

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

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

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

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

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

- Ако је a_s није дељив са k, тада се променљива `d` поставља на вредност $d' = s-1$. На основу инваријанте треба да важи да ниједан елемент у интервалу $[d', n)$ није дељив са k. Међутим, ми то не знамо, јер само знамо да је a_s није дељив са k, али не знамо да a_{s-1} није дељив са k. Дакле, овде се сигурно крије грешка у коду. Ако доделу `d = s-1` заменимо са `d = s`, тада ће инваријанта бити одржана (јер знамо да a_s није дељив са k, па са k неће бити дељив ниједан елемент иза њега).

- Ако a_s јесте дељив са k, тада се променљива `l` поставља на вредност $l' = s+1$. На основу инваријанте треба да важи да су сви елементи у интервалу $[0, l')$ дељиви са k, међутим, то ће овде бити испуњено, јер је a_s дељив са k, па су са k дељиви и сви елементи испред њега. Дакле, у овом случају је кôд коректан и инваријанта остаје одржана.

На крају, када се петља заврши можемо закључити да важи да је $l = d$ (јер све време важи да је $l \leq d$, а након петље не важи да је $l < d$). У коду се за позицију првог елемента који није дељив са k проглашава позиција $d + 1$. Иако је у оригиналној варијанти кода `l` могло без проблема да се замени са `d+1`, у овој варијанти то није могуће. Наиме, ми на основу инваријанте овог кода знамо да се на позицији $l = d$ налази елемент који није дељив са k, а да се на позицији $l - 1$ налази елемент који јесте дељив са k (осим када је $l = 0$ и тада

нема елемената дељивих са k). Зато крајњи резултат није коректан и потребно га је заменити са `d`, јер се први елемент који није дељив са k налази на позицији d (осим када су сви елементи дељиви са k, када је $d = n$, но и тада је d исправна повратна вредност). Дакле, формалном анализом смо открили и исправили две грешке.

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

На крају, скренимо пажњу на још један детаљ исправљеног програма. Парцијална коректност је јасна на основу анализе коју смо спровели, међутим, заустављање може бити доведено у питање, с обзиром на наредбу `d = s`. Заустављање доказујемо тако што показујемо да се у сваком кораку смањује број непознатих елемената, тј. да дужина интервала $[l, d)$ која је једнака $d - l$ у сваком кораку петље опада. Пошто је $l \leq d$ инваријанта, смањивање не може трајати довека, па се у неком тренутку програм зауставља. Поставља се питање да ли се $d - l$ смањује и у измењеном коду у коме се јавља наредба `d=s`. Одговор је потврдан, а образложење је суптилно. Прво, на основу услова петље важи да је $l < d$. Даље, вредност s се израчунава наредбом `s = l + (d - l) / 2` што нам да је $s = \lfloor \frac{l+d}{2} \rfloor$. Због заокруживања наниже, важи да је $s < d$ и зато се након одређивања $d' = s$, $l' = l$ вредност $d' - l'$ смањује у односу на $d - l$. Важи и да је $l \leq s$, али пошто је у другој грани $l' = s + 1$ и $d' = d$, вредност $d' - l'$ се опет смањује у односу на $d - l$. Да је заокруживање којим случајем вршено навише (нпр. `s = l + (d - l + 1) / 2`), програм би могао упасти у бесконачну петљу.

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

using namespace std;

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

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

Библиотечке функције

Задатак можемо решити и помоћу библиотечких функција за бинарну претрагу.

Да бисмо нашли први елемент који задовољава неки услов, у језику C++ можемо употребити функцију `upper_bound`, на мало необичан начин. У случају претраге преломне тачке битни су нам само елементи низа, а не елемент који се тражи (јер заправо не тражимо никакав конкретан елемент унутар низа). Зато као елемент који тражимо можемо навести било шта (на пример, нулу). Кључно је дефинисати функцију поређења (која се прослеђује као последњи аргумент функцији `upper_bound`), тако да враћа информацију о томе да су елементи који не задовољавају услов мањи од траженог, док елементи који задовољавају услов нису мањи од

траженог. Функција поређења, дакле, треба само да анализира свој други елемент и да врати информацију о томе да ли он задовољава услов (у нашем примеру, тражимо први елемент који није дељив бројем d и то је услов који се проверава у склопу функције поређења).

Рецимо да бисмо могли употребити и функцију `lower_bound`, али би тада у функцији поређења било потребно разменити редослед аргумената (њој је тражена вредност увек други аргумент) и негирати услов.

Пошто је сложеност библиотечке функције бинарне претраге $O(\log n)$, сложеност одговора на свих m упита је $O(m \log n)$.

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

using namespace std;

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

  int n;
  cin >> n;
  vector<long long> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
  long long d;
  while (cin >> d) {
    auto it = upper_bound(begin(a), end(a), 0,
                          [d](long long _, long long x) {
                            return x % d != 0;
                          });
    cout << distance(begin(a), it) << '\n';
  }
  return 0;
}
```

Задатак: Минимум ротираног сортираног низа

Сортирани низ целих бројева у коме су сви елементи различити је ротиран за k места улево и тиме је добијен циклични низ који задовољава услов да је $x_k < x_{k+1} < \ldots < x_{n-1} < x_0 < \ldots < x_{k-1}$. Један такав низ је, на пример, `11 13 15 19 24 1 3 8 9`. Напиши програм који проналази најмањи елемент таквог низа. Потруди се да се након учитавања елемената минимум пронађе у временској сложености $O(\log n)$.

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

Излаз: На стандардни излаз исписати најмањи елемент низа.

Пример

Улаз	*Излаз*
9 11 13 15 19 24 1 3 8 9	1

Решење

Линеарна претрага

Задатак можемо решити класичним алгоритмом или библиотечком функцијом за проналажења минимума. У језику C++ можемо употребити функцију `min_element`.

Овај алгоритам захтева пролазак кроз све елементе низа, па је сложености $O(n)$.

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

using namespace std;

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

  cout << *min_element(begin(a), end(a)) << endl;

  return 0;
}
```

Бинарна претрага

Поређење се првим елементом низа

Боље решење се може добити бинарном претрагом. Након ротације сви елементи у почетном делу низа су строго већи од почетног, а онда у завршном делу низа иду сви елементи који су строго мањи од почетног. Најмањи елемент који тражимо је први елемент у низу који је строго мањи од почетног и њега можемо наћи бинарном претрагом. Треба обратити пажњу на специјални случај у коме је низ ротиран за 0 места и тада не постоји елемент који је строго мањи од почетног. Бинарна претрага ће тада вратити позицију иза краја низа и у том случају најмањи елемент у низу је управо први елемент низа.

Дакле, поново тражимо први елемент који задовољава неки дати услов, што можемо урадити на сличан начин као у задатку Први који није дељив. Одржавамо позиције l и d, и инваријанта петље је да су:

- сви елементи испред позиције l (елементи на позицијама из интервала $[0, l)$) већи или једнаки почетном и сортирани су,
- сви елементи иза позиције d (елементи на позицијама из интервала (d, n)) строго мањи од почетног и сортирани су.

Претрага се завршава у тренутку када је $l = d + 1$ и тада важи да су сви елементи иза позиције d тј. сви елементи од позиције $l = d + 1$ па до краја низа строго мањи од почетног елемента низа, док су елементи од почетка низа лево од позиције l већи или једнаки од почетног елемента низа. Ако постоје елементи од позиције l до краја низа, тј. ако је $l < n$, тада су они сигурно мањи од елемената испред позиције l, а пошто су сортирани, најмањи је први од њих, тј. елемент на позицији l. У супротном, ако је $l = n$, тада постоји само леви део низа, тј. сви елементи у низу су већи или једнаки почетном елементу, а пошто је тај део низа сортиран, најмањи елемент је почетни.

Сложеност бинарне претраге је $O(\log n)$, међутим, алгоритмом доминира учитавање елемената у низ, које је сложености $O(n)$, па се предности бинарне претраге не могу ефективно осетити.

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

using namespace std;

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

```
    cin >> a[i];

  int l = 0, d = n-1;
  while (l <= d) {
    int s = l + (d-l)/2;
    if (a[s] < a[0])
      d = s-1;
    else
      l = s+1;
  }

  int min = l < a.size() ? a[l] : a[0];

  cout << min << endl;

  return 0;
}
```

Поређење се последњим елементом низа

Провера специјалног случаја након претраге се може избећи ако се уместо односа са првим, гледа однос са последњим елементом у низу. тражимо први елемент који је строго мањи од последњег.

Инваријанта овог алгоритма је да су:

- сви елементи испред позиције l (елементи на позицијама из интервала $[0, l)$) строго већи од последњег елемента низа и сортирани су,
- сви елемени иза позиције d (елементи на позицијама из интервала (d, n)) мањи или једнаки од последњег елемента низа и сортирани су.

Када се петља заврши важи да је $l = d + 1$. Зато су сви елементи иза позиције l строго већи од елемената на позицији l. Пошто је део од позиције l до краја сортиран, минимум се налази на позицији l, јер је тај део увек непразан. Заиста, мора да важи да је $l < n$, јер би у супротном последњи елемент био лево од позиције l, што је немогуће, јер су у лево од позиције l налазе елементи који су строго већи од последњег.

Сложеност бинарне претраге је $O(\log n)$, међутим, алгоритмом доминира учитавање елемената у низ, које је сложености $O(n)$, па се предности бинарне претраге не могу ефективно осетити.

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

using namespace std;

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

  int l = 0, d = n-1;
  while (l <= d) {
    int s = l + (d-l)/2;
    if (a[s] < a[n-1])
      d = s-1;
    else
      l = s+1;
  }

  cout << a[l] << endl;
```

```
  return 0;
}
```

Имплементацију можемо једноставно реализовати и инвентнивним коришћењем библиотечке функције `upper_bound`, како је описано у задатку Први који није дељив.

Сложеност бинарне претраге библиотечком функцијом је $O(\log n)$, међутим, алгоритмом доминира учитавање елемената у низ, које је сложености $O(n)$, па се предности бинарне претраге не могу ефективно осетити.

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

using namespace std;

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

  int min = *upper_bound(begin(a), end(a), 0,
                         [&a](int _, int x) {
                           return x < a.back();
                         });
  cout << min << endl;

  return 0;
}
```

2.15.3 Проналажење оптималне вредности решења бинарном претрагом

Бинарна претрага се може употребити и у процесу оптимизације, ако се проблем може формулисати као проблем проналажења преломне тачке. Овај облик претраге се понекад назива *Бинарна претрага по решењу*. Идеја је да се проблем оптимизације "наћи најмању вредност која задовољава одређени услов", сведе на проблем одлучивања "да ли дата вредност задовољава одређени услов". Бинарну претрагу је могуће применити ако проблем задовољава својство монотоности, које захтева да ако нека вредност задовољава услов, онда услов задовољавају и све вредности веће од ње, а ако не задовољава, онда услов не задовољавају ни вредности мање од ње. Наравно, сасвим сличан је задатак проналажења највеће вредности која не задовољава услов. Карактеристично за ову употребу бинарне претраге је то што вредности о којима је реч обично нису индекси елемената низа, а често се врши оптимизација и над непрекидним скупом вредности (до на одређену тачност). Такође, провера испуњења услова за сваку конкретну вредност је обично спора и желимо да смањимо број провера испуњења услова колико је могуће. Стога се уместо коришћења библиотечких функција, бинарна претрага ручно имплементира.

Задатак: Дрва

Дрвосеча треба да насече одређену количину дрвета и има тестеру коју може да подешава да сече на било којој целобројној висини (у метрима). Пошто тестера сече само дрво изнад висине на коју је постављена, што је тестера више, насећи ће се мање дрвета. Пошто дрвосеча брине о околини, он не жели да насече више дрвета него што му је потребно. Напиши програм који одређује највишу могућу целобројну висину тестере, тако да дрвосеча добије довољно дрвета (претпостави да увек постоји довољно дрвета).

Улаз: Са стандардног улаза се учитава број дрвећа у шуми n ($1 \leq n \leq 10^5$), а затим низ висина сваког дрвета (низ природних бројева између 1 и 10000, сваки у посебном реду). Након тога учитава се и количина насеченог дрвета (пошто су сва дебла исте дебљине, количина се мери у метрима висине исечених стабала).

Излаз: На стандардни излаз исписати тражену максималну целобројну висину тестере.

Пример

Улаз	*Излаз*
5 24 21 19 14 22 14	18

Решење

Оптимизација бинарном претрагом

Једно решење проблема се може засновати на бинарној претрази по решењу тј. по тражењу оптималне вредности коришћењем бинарне претраге. Постављањем тестере на висину h, код свих дрва која су виша од h биће одсечено $h_i - h$ метара, док од осталих дрва неће бити исечено ништа. На основу тога, за фиксирану висину тестере грубом силом (испитивањем сваког дрвета засебно) у времену $O(n)$ можемо израчунати укупну количину насеченог дрвета. Бинарна претрага је применљива јер знамо да је до одређених висина тестере дрвета довољно, а да је од одређене висине тестере дрвета премало, тако да заправо тражимо преломну тачку, тј. највећу висину тестере за коју је дрвета довољно тј. последњи елемент низа који задовољава услов. Техника како се то може урадити описана је у задатку Први који није дељив. Приметимо да у овом случају немамо вредности смештене у низ, већ их рачунамо по потреби. Зато бинарну претрагу имплементирамо ручно.

Ако је максимална висина дрвета M, тада је сложеност овог приступа $O(n \log M)$. Наиме, бинарном претрагом се претражује интервал $[0, M]$, па се провера да ли је насечено довољно дрвета позива $\log M$ пута. Израчунавање количине насеченог дрвета и провера да ли је она довољна врши се једним пролазак кроз низ дрвета и сложености је $O(n)$.

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

using namespace std;

int testera(const vector<int>& visine, int potrebno) {
  int od_visina = 0;
  int do_visina = *max_element(begin(visine), end(visine));
  while (od_visina <= do_visina) {
    int visina = od_visina + (do_visina - od_visina) / 2;
    long long naseceno = 0;
    for (int v : visine)
      if (v >= visina)
        naseceno += v - visina;

    if (naseceno >= potrebno)
      od_visina = visina + 1;
    else
      do_visina = visina - 1;
  }
  return do_visina;
}

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

```
  cout << testera(visina, potrebno) << endl;

  return 0;
}
```

Задатак: Муцајући подниз

Ако је s ниска, онда нека s^n означава ниску која се добија ако се свако слово понови n пута (нпр. $(xyz)^3$ је $xxxyyyzzz$). Напиши програм који одређује највећи број n такав да је s^n подниз дате ниске t (то значи да се сва слова ниске s^n јављају у ниски t, у истом редоследу као у s^n, али не обавезно узастопно).

Улаз: У првом реду стандардног улаза налази се ниска s, а у другом ниска t.

Излаз: На стандардни излаз напиши тражени број n.

Пример

Улаз	*Излаз*
xyz xaxxybyxyxzyzzb	3

Решење

Дефинисаћемо функцију којом проверавамо да ли је s^n подниз ниске t. Један начин је да експлицитно формирамо ниску s^n и да применимо алгоритам провере подниске (описан у задатку Реч у реч прецртавањем слова). Мало боље решење је да се алгоритам модификује тако да се избегне ефективно креирање ниске s^n. Функција провере прима ниску s, број n и ниску t, а затим сваки од карактера из s тражи унутар ниске t n puta.

Линеарна претрага

Када на располагању имамо функцију за проверу, тада оптималну вредност n можемо наћи линеарном претрагом. Кључна опаска је да ако је s^n подниз t за неко n, онда је s^k подниз t за свако $k \leq n$. Дакле, степен ћемо увећавати кренувши од 1 све док не наиђемо на прву вредност n за коју s^n није подниз t и тада ћемо знати да је оптимална вредност $n - 1$.

Провера да ли је ниска s^n подниз ниске t врши се у линеарном времену у односу на збир дужина ниске. Ако је дужина ниске t, време за једну проверу је $O(T)$. Провере се врше све док се не нађе оптимално n. Оно највише може бити $\lfloor \frac{T}{S} \rfloor$, где је S дужина ниске s па је сложеност $O(\frac{T^2}{S})$.

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

using namespace std;

bool jeMucajuciPodniz(const string& podniz, const string& niz, int n) {
  int i = 0;
  for (char c : podniz) {
    for (int k = 0; k < n; k++) {
      while (i < niz.size() && niz[i] != c)
        i++;
      if (i == niz.size())
        return false;
      i++;
    }
  }
  return true;
}

int najduziMucajuciPodniz(const string& podniz, const string& niz) {
   int d = 1;
   while (jeMucajuciPodniz(podniz, niz, d))
      d++;
```

```
   return d - 1;
}

int main() {
  string podniz;
  string niz;
  cin >> podniz >> niz;
  cout << najduziMucajuciPodniz(podniz, niz) << endl;
  return 0;
}
```

Бинарна претрага

Чињеница да постоји одређени облик монотоности у проблему (са порастом n јављају се прво оне вредности за које услов важи, након који иду вредности за које услов не важи), нам омогућава да тражену оптималну вредност нађемо бинарном претрагом. Сигурни смо да се та вредност налази у интервалу од 0 па до $\left\lfloor \frac{|t|}{|s|} \right\rfloor$. Техником сличном оној описаном у задатку Први који није дељив проналазимо преломну тачку, тј. најмању вредност n такву да s^n није подниз t. Приметимо да у овом случају не претражујемо вредности у неком низу, већ је низ потенцијалних вредности n који се претражује само имплицитан, па бинарну претрагу имплементирамо ручно.

Провера да ли је ниска s^n подниз ниске t врши се у линеарном времену у односу на збир дужина ниске. Ако је дужина ниске t, време за једну проверу је $O(T)$. Интервал који се претражује је $[0, \frac{T}{S}]$, где је S дужина ниске s, па је сложеност $O(\log \frac{T}{S})$.

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

using namespace std;

bool jeMucajuciPodniz(const string& podniz, const string& niz, int n) {
  int i = 0;
  for (char c : podniz) {
    for (int k = 0; k < n; k++) {
      while (i < niz.size() && niz[i] != c)
        i++;
      if (i == niz.size())
        return false;
      i++;
    }
  }
  return true;
}

int najduziMucajuciPodniz(const string& podniz, const string& niz) {
  int l = 0, d = niz.size() / podniz.size();
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (jeMucajuciPodniz(podniz, niz, s))
      l = s + 1;
    else
      d = s - 1;
  }
  return d;
}

int main() {
  string podniz;
  string niz;
  cin >> podniz >> niz;
```

```
  cout << najduziMucajuciPodniz(podniz, niz) << endl;
  return 0;
}
```

2.16 Техника два показивача

Угнежђене петље обично подразумевају постојање две бројачке променљиве од којих спољашња само увећава своју вредност током итерације, док се вредност бројачке петље у унутрашњој увећава до неке горње границе, затим се поново враћа на неку доњу границу и поново увећава и то се понавља више пута, све док спољашња бројачка променљива не достигне своју максималну вредност. Ово по правилу доводи до квадратне сложености (тј. сложености вишег степена у случају угнежђавања већег броја петљи).

Техника два показивача обухвата широку класу ефикасних алгоритама које такође карактерише постојање две или више бројачких променљивих, које се крећу кроз елементе неког низа (често сортираног). Међутим, оно што је карактеристично за њих је то што се, за разлику од унутрашњих променљивих у угнежђеним петљама, ове променљиве стално "крећу у истом смеру", тј. вредност им се или стално повећава или стално смањује (а честа је и комбинација где се "низ обилази са два краја", где се једна променљива стално повећава, а друга стално смањује). Техничка реализација може бити било помоћу једне петље која контролише вредности обе променљиве, било помоћу угнежђених петљи, али тако да се након завршетка тела унутрашње петље, спољашња променљива увећава до места где се унутрашња петља завршила. Пошто се свака променљива може променити највише n пута (где је n неко горње ограничење њихове вредности, обично дужина низа), број промена (па самим тим и извршавања тела петље) је највише $2n$ и линеаран је по n тј. сложеност му је $O(n)$.

Алгоритми засновани на техници два показивача обично могу да се изведу коришћењем одсецања примењених на угнежђене петље, па је, као и код сваке примене одсецања, потребно пажљиво образложити њихову коректност.

Задатак: Обједињавање

У школи малих жутих мрава наставник је прегледао контролни задатак. Прво је прегледао ђаке који су радили групу А, а затим оне који су радили групу Б, средио је резултате за сваку групу и мраве поређао на основу броја поена који су освојили. Напиши програм који му помаже да од уређеног списка ученика који су радили задатке из групе А и од уређеног списка ученика који су радили задатке из групе Б добије јединствен уређен списак свих ученика.

Улаз: Са стандардног улаза се уноси број ђака m који су радили групу А ($5 \leq m \leq 25000$), а затим неопадајуће сортиран низ поена тих ђака (елементи су у једној линији, раздвојени са по једним размаком). Након тога се уноси број n ђака који су радили групу Б ($5 \leq n \leq 25000$), а затим неопадајуће сортиран низ поена тих ђака (елементи су у једној линији, раздвојени са по једним размаком).

Излаз: На стандардни излаз исписати неопадајуће сортирани низ поена свих ђака заједно, раздвојене са по једним размаком.

Пример

Улаз	*Излаз*
4 1 3 5 7 3 2 4 5	1 2 3 4 5 5 7

Решење

Сортирање

Наиван начин да се задатак реши је да се елементи оба учитана низа прекопирају у трећи (било помоћу петље, било библиотечком функцијом `copy`) и да се онда сортирају (на било који од начина приказаних у задатку Сортирање бројева, најбоље библиотечком функцијом `sort`).

Копирање низова дужине m и n захтева $m + n$ операција, а сортирање $O((m + n) \cdot \log(m + n))$. Технички, могли смо одмах елементе учитавати у резултујући низ и тако уштедети меморију и време потребно за копирање, али доминатни фактор, а то је време потребно за сортирање би остао. Приметимо да у овом решењу нисмо уопште употребили чињеницу да су полазни елементи већ сортирани.

Иако ово решење по времену извршавања не заостаје пуно у односу на оптимално (његово време извршавања је квазилинеарно тј. $O((m+n) \cdot \log(m+n))$, а оптимално време је линеарно тј. $O(m+n)$), оно је доста компликованије него што је потребно. То се у овој имплементацији не види, јер је употребљена библиотечка функција сортирања, међутим, имплементација ефикасног алгоритма сортирања захтева захтева напредне технике програмирања. Интересантно, један од популарних алгоритама сортирања је алгоритам сортирања је сортирање обједињавањем (енгл. merge sort) који као свој основни корак захтева извршавање обједињавање два сортирана низа у трећи. Самим тим, донекле је бесмислено проблем обједињавања решавати свођењем на компликованији проблем сортирања.

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

using namespace std;

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

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

  // ucitavamo drugi niz
  int m;
  cin >> m;
  vector<int> b(m);
  for (int i = 0; i < m; i++)
    cin >> b[i];

  // kopiramo dva niza u treci, jedan iza drugog
  vector<int> c(m + n);
  copy(a.begin(), a.end(), c.begin());
  copy(b.begin(), b.end(), next(c.begin(), n));
  // sortiramo treci niz
  sort(c.begin(), c.end());

  // ispisujemo rezultat
  for (int i = 0; i < c.size(); i++)
    cout << c[i] << " ";
  cout << endl;

  return 0;
}
```

Алгоритам обједињавања

Задатак можемо решити ефикасним алгоритом, заснованом на техници два показивача. *Алгоритам обједињавања* (енгл. *merge*) подразумева да су низови који се обједињавају сортирани. Због тога ће први елемент резултата бити мањи од почетних елемената два низа (ако су једнаки, свеједно је који од њих узети). Ако је један од низова празан, резултат обједињавања је други низ и његове елементе је потребно једноставно прекопирати у резултат. Ако низови нису празни, пошто су сортирани, први елемент низа је уједно најмањи у њему. Мањи од два почетна елемента је мањи (или једнак) од почетног елемента другог низа, па је мањи или једнак свим елементима у оба низа и самим тим је најмањи елемент од свих и треба да буде први у резултату. Када се тај елемент уклони из низа, добијамо проблем истог типа као и полазни, који се онда решава на исти начин. Имплементација може бити рекурзивна, међутим, рекурзија је репна и лако се елиминише.

Током итеративне имплементације одржавају се два показивача. Променљива i указује позицију текућег

елемента првог и j која указује на текући елемент другог низа. Док су обе ове променљиве мање од дужине низа по којем се крећу, поредимо елементе на тим позицијама. Ако је елемент на позицији i у првом низу мањи (или једнак) елементу на позицији j у другом низу, тада тај елемент преписујемо у трећи низ (на позицију k коју иницијализујемо на нулу и увећавамо приликом додавања сваког новог елемента) и увећавамо i за 1. У супротном у трећи низ преписујемо елемент из другог низа са позиције j и увећавамо j. Када бар једна од променљивих достигне дужину одговарајућег низа, тада елементе преосталог низа преписујемо у трећи низ. Не морамо експлицитно проверавати да ли у неком од ових низова има преосталих елемената, већ можемо у једној петљи копирати преостале елементе првог, а у другој петљи копирати преостале елементе другог низа (једна од ових петљи ће бити празна).

Прикажимо рад овог алгоритма и на једном примеру.

- Претпоставимо да је потребно објединити наредна два низа.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   _ _ _ _ _ _ _ _ _
i           j         k
```

- У првом кораку је $i = 0$ и $j = 0$, па се пореде елементи на позицијама 0, тј. елементи 1 и 2. Пошто је 1 мањи, он се преписује у резултујући низ и увећава се леви показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 _ _ _ _ _ _ _ _
  i         j           k
```

- Сада је $i = 1$ и $j = 0$, па се пореде елементи на позицијама 1 и 0, тј. 3 и 2. Пошто је 2 мањи, он се преписује у резултујући низ и увећава се десни показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 _ _ _ _ _ _ _
  i           j           k
```

- Сада је $i = 1$ и $j = 1$, па се пореде елементи на позицијама 1 и 1, тј. 3 и 4. Пошто је 3 мањи, он се преписује у резултујући низ и увећава се леви показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 _ _ _ _ _ _
    i         j           k
```

- Сада је $i = 2$ и $j = 1$, па се пореде елементи на позицијама 2 и 1, тј. 6 и 4. Пошто је 4 мањи, он се преписује у резултујући низ и увећава се десни показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 4 _ _ _ _ _
    i           j           k
```

- Сада је $i = 2$ и $j = 2$, па се пореде елементи на позицијама 2 и 2, тј. 6 и 5. Пошто је 5 мањи, он се преписује у резултујући низ и увећава се поново десни показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 4 5 _ _ _ _
    i           j           k
```

- Сада је $i = 2$ и $j = 3$, па се пореде елементи на позицијама 2 и 3, тј. 6 и 8. Пошто је 6 мањи, он се преписује у резултујући низ и увећава се леви показивач.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 4 5 6 _ _ _
      i         j           k
```

- Сада је $i = 3$ и $j = 3$, па се пореде елементи на позицијама 3 и 3, тј. 8 и 8. Пошто су једнаки, било који од њих (на пример десни) може бити преписан у резултујући низ и одговарајући показивач се увећава.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 4 5 6 8 _ _
      i             j               k
```

- Пошто у другом низу више нема елемената, преостала два елемента левог низа (8 и 9) се преписују на крај резултата.

```
a:          b:        c:
0 1 2 3 4   0 1 2 3   0 1 2 3 4 5 6 7 8
1 3 6 8 9   2 4 5 8   1 2 3 4 5 6 8 8 9
        i           j                 k
```

Сваки показивач пролази кроз један од два низа и укупан број корака је $m+n$, па је сложеност овог алгоритма $O(m+n)$.

```
#include <iostream>

using namespace std;

// objedinjava sortirani niz a od n elemenata i sortirani niz b od m
// elemenata smestajuci rezultat u sortirani niz c
void objedini(int a[], int n, int b[], int m, int c[]) {
  int i = 0, j = 0, k = 0;
  while (i < n && j < m)
    c[k++] = a[i] < b[j] ? a[i++] : b[j++];
  while (i < n)
    c[k++] = a[i++];
  while (j < m)
    c[k++] = b[j++];
}

// najveci broj elemenata niza predvidjen tekstom zadatka

const int MAX = 50000;
int main() {
  ios_base::sync_with_stdio(false);

  // ucitavamo prvi niz
  int n;
  cin >> n;
  int a[MAX];
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // ucitavamo drugi niz
  int m;
  cin >> m;
  int b[MAX];
  for (int i = 0; i < m; i++)
    cin >> b[i];

  // objedinjavamo dva niza u treci
  int c[MAX + MAX];
  objedini(a, n, b, m, c);

  // ispisujemo rezultat
  for (int i = 0; i < m + n; i++)
```

```
    cout << c[i] << " ";
  cout << endl;

  return 0;
}
```

Библиотечка функција

Рецимо и да у стандардној библиотеци језика C++ постоји функција `merge` која врши обједињавање. Функцији се прослеђују итератори који ограничавају први низ, итератори који ограничавају други низ и итератор на почетак трећег низа у који се смешта резултат. Додатно, могуће је проследити и функцију поређења која одређује редослед сортирања (слично као у задатку Сортирање такмичара).

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

using namespace std;

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

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

  // ucitavamo drugi niz
  int m;
  cin >> m;
  vector<int> b(m);
  for (int i = 0; i < m; i++)
    cin >> b[i];

  // objedinjavamo ih
  vector<int> c(m + n);
  merge(a.begin(), a.end(),
        b.begin(), b.end(),
        c.begin());

  // ispisujemo rezultat
  for (int i = 0; i < c.size(); i++)
    cout << c[i] << " ";
  cout << endl;

  return 0;
}
```

Задатак: Близанци

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

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

Улаз: Опис улаза: са стандардног улаза се учитава:

- у првом реду број мушких одела m ($1 \leq m \leq 50000$),

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

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

Пример

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

Објашњење

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

Решење

Груба сила

Један могући приступ је да одредимо разлику (прецизније, апсолутну вредност разлике) у цени између сваког мушког и сваког женског одела, па од тих разлика нађемо најмању.

Сложеност одговара броју парова и процењује се као $O(m \cdot n)$, где је m број мушких, а n број женских одела.

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

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

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

  cout << minRazlika << endl;

  return 0;
}
```

Упоредни пролаз кроз сортиране низове

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

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

На пример, нека су након сортирања вредности једнаке следећим.

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

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

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

Сложеношћу доминира сортирање, које се извршава у времену $O(m \log m + n \log n)$. Након сортирања, пролазак са два показивача се извршава у времену $O(m + n)$.

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

using namespace std;

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

  sort(begin(a1), end(a1));
  sort(begin(a2), end(a2));
  int i1 = 0, i2 = 0;
  int minRazlika = numeric_limits<int>::max();
  while (i1 < n1 && i2 < n2)
    if (a1[i1] <= a2[i2]) {
      minRazlika = min(minRazlika, a2[i2] - a1[i1]);
      i1++;
    } else {
```

```
      minRazlika = min(minRazlika, a1[i1] - a2[i2]);
      i2++;
    }

  cout << minRazlika << endl;

  return 0;
}
```

Задатак: Број парова датог збира

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

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

Решење

Груба сила

Задатак можемо решити анализирајући збир елемената сваког пара у низу a. Различити парови низа a су облика (a_i, a_j), при чему је $i < j$ (у супротном би се исти пар рачунао два пута). Зато i узима вредности од 0 до $n-2$, а j узима вредности од $i+1$ до $n-1$. Парове можемо набројати помоћу две угнежђене петље, тако да се у телу унутрашње петље проверавамо да ли је збир текућа два елемента једнак s и ако јесте, увећавамо бројач парова (пре петљи иницијализован на нулу).

Сложеност овог наивног приступа одговара броју парова наведеног облика, што је $O(n^2)$.

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

using namespace std;

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

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

  int brojParova = 0;
  for (int i = 0; i < n - 1; i++)
    for(int j = i + 1; j < n; j++)
      if (a[i] + a[j] == s)
        brojParova++;

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

Итеративни обилазак са два краја низа помоћу два показивача

Задатак можемо решити тако што сортирамо низ неопадајуће (пошто су сви елементи различити, он ће заправо бити сортиран строго растуће) и применимо технику два показивача, имплементирану итеративно.

Чланове датог збира можемо тражити полазећи са оба краја низа. Обилазимо низ са оба краја левог ($l = 0$) и десног ($d = n-1$). Упоредимо $a_l + a_d$ са s.

- Ако је $a_l + a_d > s$ потребно је смањити збир пара елемената, што постижемо узимањем заменом елемента мањим, па пошто је низ сортиран у растућем поретку, прелазимо на следећи елемент у десном делу низа (d умањујемо за 1).

- Ако је $a_l + a_d < s$ потребно је повећати збир пара елемената, што постижемо заменом елемента већим, па пошто је низ сортиран у растућем поретку, прелазимо на следећи елемент у левом делу низа (l увећавамо за 1).
- Ако је $a_l + a_d = s$ увећамо број тражених парова, и прелазимо на следећи елемент у левом делу низа (l увећавамо за 1) и на следећи елемент у десном делу низа (d умањујемо за 1).

Процес настављамо док не обиђемо цео низ, то јест док је $l < d$.

Прикажимо ово на примеру проналажења елемената чији је збир 14 у сортираном низу $1, 2, 5, 7, 9, 11, 13, 14$. Крећемо од пара $(1, 14)$. Пошто је збир већи од траженог, померамо десни крај улево и анализирамо пар $(1, 13)$, који има тражени збир. Зато прелазимо на $(2, 11)$. Пошто је збир сада мањи, померамо леви крај удесно и анализирамо пар $(5, 11)$. Збир је превелики и померамо десни крај улево и анализирамо пар $(5, 9)$. Он има тражени збир, па прелазимо на $(7, 7)$, но тај пар не анализирамо, јер су се показивачи сусрели.

Докажимо коректност претходног поступка. Посматрајмо скуп $S_{l,d}$ који садржи све парове (a_i, a_j) такве да је $0 \leq i < l$ и да је $d < j < n$. Инваријанта претходне петље биће то да:

- променљива која чува текући број парова (означимо је са b) чува број парова скупа $S_{l,d}$ такве да је $a_i + a_j = s$,
- за свако $0 \leq i < l$ важи да је $a_i + a_d < s$ и за свако $d < j < n$ важи да је $a_l + a_j > s$.

Пре уласка у петљу важи да је $l = 0$ и $d = n - 1$. Тада нема индекса i таквог да важи $0 \leq i < l$ нити индекса j таквог да важи $d < j < n$, па је $S_{l,d}$ празан. Пошто је $b = 0$, оба дела инваријанте важе.

Претпоставимо да инваријанта важи при уласку у тело петље и докажимо да је извршавање тела петље одржава.

Претпоставимо прво да је $a_l + a_d > s$. Тада d умањујемо за 1 не мењајући при том број парова b, тј. након извршавања тела петље важи да је $l' = l$, $d' = d - 1$ и $b' = b$. Скуп $S_{l',d'} = S_{l,d-1}$ се може разложити на:

1. скуп $S_{l,d}$ и
2. скуп скуп свих парова (a_i, a_j) таквих да је $0 \leq i < l$ и $j = d$.

Број парова траженог збира у скупу $S_{l,d}$ једнак је b (на основу првог дела инваријанте), док се у другом скупу не налази ни један такав пар. Заиста, на основу другог дела инваријанте знамо да свако $0 \leq i < l$ важи да је $a_i + a_d < s$, па међу паровима другог скупа не може бити ни један који има збир једнак s. Пошто је $b' = b$, први део инваријанте остаје очуван. Потребно је да покажемо и да други део инваријанте остаје очуван. Прво, треба да докажемо да за свако $0 \leq i < l'$ важи да је $a_i + a_{d'} < s$. Пошто је $l' = l$, на основу другог дела инваријанте знамо да за све такве индексе i важи да је $a_i + a_d < s$, а пошто је низ сортиран и пошто су му елементи различити, важи да је $a_{d'} = a_{d-1} < a_d$, па је $a_i + a_{d'} = a_i + a_{d-1} < a_i + a_d < s$. Треба да докажемо и да за свако $d' < j < n$, важи да је $a_{l'} + a_j > s$. На основу другог дела инваријанте то важи за све $d < j < n$. Пошто је $l' = l$ и $d' = d - 1$, остаје само још да се докаже да тај услов важи за d, тј. само да се докаже да важи да је $a_l + a_d > s$, но то важи на основу претпоставке (тј. гране коју тренутно анализирамо).

Веома слично се показује да се у случају $a_l + a_d < s$ повећањем броја l и не мењањем броја d инваријанта одржава.

На крају, остаје случај када је $a_l + a_d = s$. У том случају се врши увећање броја l, умањење броја d и увећање броја b, тј. важи да је $l' = l + 1$, $d' = d - 1$ и $b' = b + 1$. Скуп $S_{l',d'} = S_{l+1,d-1}$ се може разложити на:

1. скуп $S_{l,d}$,
2. скуп свих парова (a_i, a_j) таквих да је $0 \leq i < l$, $j = d$,
3. скуп свих парова таквих да је $i = l$, $d < j < n$ и
4. скуп који садржи само пар (a_l, a_d).

На основу првог дела инваријанте важи да у скупу $S_{l,d}$ има b парова чији је збир s. Пошто је $a_l + a_d = s$, (a_l, a_d) је још један тражени пар. Остаје још да покажемо да у преостала два скупа не постоји ни један пар чији је збир s. Заиста, скуп свих парова таквих да је $0 \leq i < l$, $j = d$ и $a_i + a_j = s$, је празан, јер на основу другог дела инваријанате знамо да је у свим тим паровима $a_i + a_j < s$. Аналогно, на основу другог дела инваријанте доказујемо да је празан и скуп свих парова таквих да је $i = l$, да је $d < j < n$ и $a_i + a_j = s$, јер за све те парове важи да је $a_i + a_j > s$. Дакле, први део инваријанте остаје очуван. Потребно је још доказати да је очуван и други део инваријанте. Потребно је доказати да је за за свако $0 \leq i < l'$ важи да је $a_i + a_{d'} < s$. На основу другог дела инваријанте знамо да за свако $0 \leq i < l$ важи да је $a_i + a_d < s$. Пошто је низ сортиран

и сви су му елементи различити и пошто је $d' = d - 1$, важи да је $a_{d'} < a_d$. Зато је $a_i + a_{d'} < a_i + a_d < s$. Пошто је $l' = l + 1$, остаје да се то докаже још да је $a_l + a_{d'} < s$. Но знамо да је $a_l + a_{d'} < a_l + a_d = s$. Аналогно се доказује још да за свако $d' < j < n$ важи $a_{l'} + a_j > s$.

Крај петље наступа када је $l \geq d$. Скуп свих парова (a_i, a_j) таквих да је $0 \leq i < j < n$ се може разложити на:

1. скуп свих парова (a_i, a_j) таквих да је $j \leq d$,
2. скуп $S_{l,d}$ тј. скуп свих парова (a_i, a_j) таквих да је $0 \leq i < l$ и $d < j < n$ и
3. скуп свих парова (a_i, a_j) таквих да је $l \leq i$.

У првом и трећем скупу нема ни један пар чији је збир једнак s. Заиста, ако је $j \leq d$, тада важи да је $0 \leq i < j \leq d \leq l$. На основу другог дела инваријанте знамо да тада важи да је $a_i + a_d < s$, а пошто је низ сортиран важи и да је $a_j \leq a_d$, па је $a_i + a_j \leq a_i + a_d < s$. Ако важи да је $l \leq i$, тада важи и да је $d \leq l \leq i < j < n$. На основу другог дела инваријанте знамо да је $a_l + a_j > s$, па пошто је низ сортиран важи да је $a_l \leq a_i$ и зато је $a_i + a_j \geq a_l + a_j > s$. На основу првог дела инваријанте знамо да је број b једнак броју парова из другог скупа, што је на основу претходног укупан број тражених парова.

Пошто се у сваком кораку разлика између d и l смањи бар за 1 (некада и за 2), укупан број корака не може бити већи од n, па је сложеност $O(n)$.

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

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

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

  sort(begin(a), end(a));

  int brojParova = 0;
  int levo = 0, desno = n - 1;
  while (levo < desno)
    if (a[levo] + a[desno] > s)
      desno--;
    else if (a[levo] + a[desno] < s)
      levo++;
    else {
      brojParova++;
      levo++;
      desno--;
    }

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

Задатак: Тројке датог збира (3sum)

Такмичари из програмирања имају свој рејтинг који је изражен као неки цео број. На државно екипно такмичење школе треба да пошаљу своје трочлане екипе, међутим, да би такмичење било што занимљивије, упутство организатора је да све екипе буду уједначене тј. да свака екипа на такмичењу има збирни рејтинг нула. Ако су познати рејтинзи свих такмичара из неке школе, напиши програм који одређује на колико начина школа може да одабере своју екипу.

Улаз: Са стандардног улаза се уноси број такмичара n ($3 \leq n \leq 1000$), а затим и n различитих целих бројева из интервала $[-10^6, 10^6]$, раздвојених са по једним размаком (то су рејтинзи такмичара).

Излаз: На стандардни излаз исписати број могућих трочланих екипа таквих да је укупан збир рејтинга та три члана једнак нули.

Пример

Улаз	*Излаз*
9 -8 -5 7 4 1 -2 9 -3 2	4

Решење

Груба сила

Наивно решење је да се провере све тројке елемената (a_i, a_j, a_k) за $i < j < k$, тј. да се употребе три угнежђене петље у чијем се телу проверава да ли је $a_i + a_j + a_k = 0$ и ако јесте, да се увећа бројач (користимо алгоритам бројања елемената филтриране серије, као, на пример, у задатку Просек одличних).

Сложеност овог алгоритма одговара броју тројки и једнака је $O(n^3)$, што је недопустиво велико.

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

using namespace std;
int main() {
  ios_base::sync_with_stdio(false);
  // ucitavamo niz
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // ukupan broj trojki ciji je zbir 0
  int brojTrojki = 0;
  // prolazimo kroz sve trojke (ai, aj, ak) takve da je i < j < k
  for (int i = 0; i < n - 2; i++)
    for (int j = i + 1; j < n - 1; j++)
      for (int k = j + 1; k < n; k++)
        // ako je zbir trojke 0
        if (a[i] + a[j] + a[k] == 0)
          // uvecavamo broj trojki
          brojTrojki++;

  // ispisujemo broj trojki
  cout << brojTrojki << endl;
  return 0;
}
```

Решење техником два показивача

Задатак може бити решен сортирањем и применом технике два показивача да би се иза сваког елемента a_i (почетног елемента тројке) израчунао број парова различитих елемената који имају збир $-a_i$. Претпоставимо да на почетку низ сортирамо растуће. Тада ће сваки суфикс иза позиције i бити сортиран, тако да на њега можемо применити алгоритам обиласка низа са два краја, на исти начин као у задатку Број парова датог збира.

Пошто је бројање парова у делу низа дужине m сложености $O(m)$ и понавља се за све дужине од $n-1$ до 2, а сложеност сортирања је $O(n \log n)$, укупна сложеност је $O(n^2)$.

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

using namespace std;

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

  // sortiramo ga
  sort(begin(a), end(a));

  // ukupan broj trojki ciji je zbir 0
  int brojTrojki = 0;

  // analiziramo svaki moguci pocetni element trojke
  for (int i = 0; i < n - 2; i++) {
    // izracunavamo broj parova u delu niza [i+1, n) ciji je zbir -a[i]
    // posto je ceo niz sortiran, sortiran je i taj deo

    // koristimo tehniku dva pokazivaca
    int l = i + 1;
    int d = n - 1;
    while (l < d) {
      if (a[i] + a[l] + a[d] > 0)
        d--;
      else if (a[i] + a[l] + a[d] < 0)
        l++;
      else {
        brojTrojki++;
        l++;
        d--;
      }
    }
  }

  // ispisujemo ukupan broj trojki
  cout << brojTrojki << endl;
  return 0;
}
```

Задатак: Разлика висина

У једном одељењу бирају се глумци за школску представу “Станлио и Олио”. Ови глумци су познати по томе што им је била велика разлика у висини. Напиши програм који одређује на колико начина можемо да одаберемо два глумца из одељења тако да им је разлика једнака датом броју r.

Улаз: Са стандардног улаза се уноси прво позитиван природан број r, у наредном реду број ученика у одељењу n ($1 \leq n \leq 50000$), а након тога у наредних n редова висина сваког ученика у милиметрима.

Излаз: На стандардни излаз испиши број парова које је могуће формирати.

Пример

Улаз	*Излаз*
2350 5 15745 18095 15745 16234 13395	4

Решење

Груба сила

Наиван начин да се задатак реши је да се испитају сви уређени парови ученика и да се преброје они чија је разлика једнака траженој.

Пошто уређених парова ученика има n^2, Сложеност оваквог алгоритма је $O(n^2)$.

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

using namespace std;

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

Сортирање

Као и у многим проблемима претраге, сортирање низа може довести до ефикаснијих решења. За почетак, ако је низ сортиран, довољно је само да проверавамо парове такве да је други елемент пара иза првог. Дакле, за сваки елемент низа одређујемо број елемената иза њега који са њим дају тражену разлику (он је умањилац, а тражимо потенцијалне умањенике). Пошто је низ сортиран, чим наиђемо на први елемент иза њега који има већу разлику од тражене, такви ће бити и сви елементи у наставку низа, па можемо извршити одсецање и прећи на обраду следећег елемента (умањиоца).

У најгорем случају не долази до одсецања, а проверавају се сви парови којих има $\binom{n}{2} = \frac{n(n-1)}{2}$, па је сложеност овог приступа $O(n^2)$.

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

using namespace std;

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

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

  sort(begin(a), end(a));

  int broj = 0;
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++)
      if (a[j] - a[i] == razlika)
        broj++;
      else if (a[j] - a[i] > razlika)
        break;

  cout << broj << endl;

  return 0;
}
```

Техника два показивача

Задатак можемо решити техником два показивача (слично као у задатку Број парова датог збира). Низ мора бити сортиран, а оба показивача се крећу слева надесно. Једноставности ради, претпоставимо прво да у низу нема дупликата.

Прикажимо како би алгоритам радио на примеру одређивања броја елемената чија је разлика 8 у наредном низу.

`1 3 7 8 11 14 16 17 19`

- Крећемо од пара `1 3`. Пошто је разлика мања од тражене елемент `3` не може бити умањеник, па повећавамо умањеник на `7`.
- Анализирамо пар `1 7`. Ситуација је опет иста, па опет повећавамо умањеник на `8`.
- Анализирамо пар `1 8`. И ту је ситуација иста, па опет повећавамо умањеник на `11`.
- Анализирамо пар `1 11`. Овај пут је разлика већа од тражене. Стога можемо закључити да `1` не може бити умањилац (даљим померањем умањеника надесно, разлика би се само повећала). Стога прелазимо на наредни умањилац, а то је `3`. Кључна напомена је да су разлике свих умањеника испред `11` и броја `3` мање од тражене разлике `8` (јер су такве биле разике и када је умањилац био мањи)
- Анализирамо пар `3 11`. То је први пар који има дату разлику. Ако су елементи различити, тада се за све умањенике после `11` добија већа разлика у односу на умањилац `3`, па можемо да померимо умањилац на `7`. Слично, умањеник `11` не може да направи ни један даљи пар чија би разлика била једнака траженој, па можемо да померимо умањеник на `14`.
- Анализирамо пар `7 14`. Разлика је мања од тражене, па повећавамо умањеник на `16`.
- Анализирамо пар `7 16` разлика је већа од тражене, па померамо умањилац на `8`.
- Анализирамо пар `8 16` чија је разлика једнака траженој. Након тога можемо повећати и умањилац на `11` и умањеник на `17`.
- Анализирамо пар `11 17` и пошто му је разлика мања од тражене, померамо умањеник на `19`.
- Анализирамо пар `11 19` коме је разлика већа од тражене и пошто се умањеник не може даље повећати, завршавамо поступак.

Опишимо формално претходни поступак и докажимо његову коректност. Одредимо колико парова дате разлике r постоји у интервалу $[i, n)$, ако знамо да важи инваријанта да је је $a_{j-1} - a_i < r$. Вршимо иницијали-

зацију $i = 0$ и $j = 1$, тако да одређујемо број парова и интервалу $[0, n)$, а инваријанта је задовољена јер је $a_{j-1} - a_i = a_0 - a_0 = 0 < r$.

- Ако је $j = n$, тада у интервалу $[i, n)$ не постоји ни један пар дате разлике. Заиста, на основу инваријанте важи да је $a_{n-1} - a_i < r$. Пошто је низ сортиран, и повећањем i и смањивањем j разлика се смањује. Зато парови бројева унутар интервала $[i, n)$ имају мању разлику од r.
- Ако је $a_j - a_i < r$, тада знамо да у интервалу $[i, j]$ не постоји ни један пар бројева чија је разлика једнака r (јер је разлика елемената унутар интервала увек мања неко разлика крајњих елемената). Инваријанта је задовољена за пар $(i, j + 1)$ па увећавамо j и настављамо поступак.
- Ако је $a_j - a_i > r$, тада ни један пар $a_{j'} - a_i$ за $i < j' < n$ нема разлику r. На основу инваријанте знамо да је $a_{j-1} - a_i$ мање од r, па пошто је низ сортиран то важи и за све елементе $i < j' < j$. Пошто је $a_j - a_i > r$ и пошто је низ сортиран повећањем j се повећава разлика, па су разлике за $j \leq j' < n$ веће од r. Зато се у интервалу $[i, n)$ сви евентуални парови чија је разлика r налазе у интервалу $[i + 1, n)$ и поступак настављамо тако што увећавамо i за 1. Још морамо доказати да тада инваријанта важи тј. да је $a_{j-1} - a_{i+1} < r$, међутим то важи јер је низ сортиран и важи $a_i < a_{i+1}$, а на основу инваријанте је важило да је $a_{j-1} - a_i < r$.
- На крају, ако је $a_j - a_i = r$, тада смо пронашли један пар. Пошто смо претпоставили да у низу нема дупликата и да је низ сортиран, a_i не може бити члан ни једног другог пара са разликом r у интервалу $[i, n)$ - пошто је низ сортиран померањем умањеника налево разлика се смањује, а померањем надесно, она се повећава. Дакле, сви евентуални парови чија је разлика r налазе се у интервалу $[i + 1, n)$. Поступак се може наставити увећавањем и индекса i и индекса j за 1. Заиста, инваријанта је задовољена јер је $a_{j+1-1} - a_{i+1} = a_j - a_{i+1} < a_j - a_i = r$.

Сложеност овог приступа је $O(n \log n)$ захваљујући почетном сортирању, док је сложеност друге фазе, након сортирања линеарна тј. $O(n)$. Заиста и умањеник и умањилац се крећу у истом смеру (вредност оба показивача се само увећава), па се може направити највише $2n$ корака.

Пређимо сада на случај у ком се елементи у низу могу понављати.

Обрада дупликата читањем серије истих елемената

Ако се елементи у низу понављају, онда у тренутку када нађемо први пар (i, j) такав да је $a_i - a_j = r$, одређујемо број појављивања n_i елемента a_i и број појављивања n_j елемента a_j, број парова увећавамо за $n_i \cdot n_j$ (јер свако појављивање вредности a_i можемо искомбиновати са сваким појављивањем вредности a_j) и након тога поступак настављамо од индекса $(i + n_i, j + n_j)$.

Сложеношћу и даље доминира сортирање чија је сложеност $O(n \log n)$, док је сложеност друге фазе и даље линеарна тј. $O(n)$.

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

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

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

  sort(begin(a), end(a));

  int broj = 0;
  int i = 0, j = 1;
  while (j < n) {
```

```
    if (a[j] - a[i] < razlika)
      j++;
    else if (a[j] - a[i] > razlika)
      i++;
    else {
      // pronalazimo sve elemente jednake a[i]
      int ii;
      for (ii = i+1; ii < n && a[ii] == a[i]; ii++)
        ;
      // odredjujemo koliko ih ima
      int broj_ai = ii - i;
      // preskacemo ih
      i = ii;

      // pronalazimo sve elemente jednake a[j]
      int jj;
      for (jj = j+1; jj < n && a[jj] == a[j]; jj++)
        ;
      // odredjujemo koliko ih ima
      int broj_aj = jj - j;
      // preskacemo ih
      j = jj;

      // uvecavamo brojac za broj parova (a[i], a[j])
      broj += broj_ai * broj_aj;
    }
  }

  cout << broj << endl;

  return 0;
}
```

Обрада дупликата пребројавањем појављивања

Још један начин да се реши проблем понављања елемената је да се у првој фази низ вредности трансформише у низ парова који садрже вредности и њихов број појављивања.

Сортирање је сложености $O(n \log n)$. Пребројавање елемената се након тога извршава у сложености $O(n)$, једним проласком кроз низ, након чега се са два показивача пролази кроз низ опет у сложености $O(n)$. Укупна сложеност је, дакле, $O(n \log n)$. Имплементација користи и помоћни низ, али меморијска сложеност остаје $O(n)$.

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

using namespace std;

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

```
  // sortiramo niz
  sort(begin(a), end(a));

  // odredjujemo broj pojavljivanja svakog elementa
  vector<pair<int, int>> b;
  b.reserve(n);
  b.push_back(make_pair(a[0], 1));
  for (int i = 1; i < n; i++) {
    if (a[i] == b.back().first)
      b.back().second++;
    else
      b.push_back(make_pair(a[i], 1));
  }

  // trazimo elemente cija je razlika jednaka datoj
  int broj = 0;
  int i = 0, j = 0;
  while (j < b.size()) {
    if (b[j].first - b[i].first < razlika)
      j++;
    else if (b[j].first - b[i].first > razlika)
      i++;
    else {
      broj += b[j].second * b[i].second;
      i++; j++;
    }
  }
  cout << broj << endl;
  return 0;
}
```

Обрада дупликата коришћењем мапе тј. речника

Још један интересантан начин имплементације је да уместо сортирања низа употребимо мапу тј. речник која пресликава елементе у њихов број појављивања (слично као у задатку Фреквенције речи) и да затим са два показивача пролазимо кроз кључеве мапе (у њиховом сортираном редоследу). Као имплементацију сортиране мапе језику C++ можемо употребити колекцију `map`.

Пошто је сложеност појединачног уметања и појединачне претраге у сортираној мапи тј. речнику $O(\log n)$, овим добијамо алгоритам сложености $O(n \log n)$. Итерација кроз мапу помоћу два показивача је сложености $O(n)$.

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

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  int razlika;
  cin >> razlika;

  // preslikavamo svaki element u njegov broj pojavljivanja
  map<int, int> m;
  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
```

```
    m[x]++;
  }

  // odredjujemo broj elemenata cija je razlika jednaka datoj
  int broj = 0;
  auto i = m.begin(), j = m.begin();
  while (j != m.end()) {
    if (j->first - i->first < razlika)
      j++;
    else if (j->first - i->first > razlika)
      i++;
    else {
      broj += j->second * i->second;
      j++;
    }
  }
  cout << broj << endl;
  return 0;
}
```

Задатак: Сегмент датог збира у низу природних бројева

У датом низу позитивних природних бројева наћи све сегменте (њихов почетак и крај) чији је збир једнак датом позитивном броју (бројање позиција почиње од нуле).

Улаз: У првој линији стандардног улаза налази се задати позитиван природни број z који представља дати збир $0 < z < 10^6$, у другој број елемената низа, N ($2 \leq N \leq 50000$), а затим, у свакој од наредних N линија стандардног улаза, по један елемент низа (позитиван природни број мањи од 200).

Излаз: У свакој линији стандарног излаза исписују се два броја (цели бројеви) одвојена празнином, који представљају индексе почетка и краја сегмента (бројано од нуле). Ако постоји више тражених сегмената њихове индексе исписати сортирано на основу левог краја.

Пример1

Улаз	*Излаз*
125	0 2
10	2 4
60	5 6
40	6 9
25	
50	
50	
100	
25	
35	
30	
35	

Решење

Груба сила

Наивно решење грубом силом претпоставља да се израчунају збирових свих сегмената и да се провери да ли су једнаки датом броју. Чак и када збирове сегмената рачунамо инкрементално (као, на пример, у задатку Префикс највећег збира), добијамо неефикасно решење.

Постоји $O(n^2)$ сегмената, а збир сваког сегмента на основу збира претходног сегмента добијамо у сложености $O(1)$, па је укупна сложеност $O(n^2)$.

```
#include <iostream>

using namespace std;
```

```
int main() {
  // ubrzavamo ucitavanje
  ios_base::sync_with_stdio(false);

  // ucitavamo trazeni zbir
  int trazeniZbir;
  cin >> trazeniZbir;

  // ucitavamo elemente niza
  int n;
  cin >> n;
  int a[50000];
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // analiziramo sve intervale [i, j] za i <= j
  for (int i = 0; i < n; i++) {
    // zbir intervala [i, j]
    int zbir = 0;
    for (int j = i; j < n; j++) {
      // izracunavamo zbir intervala [i, j] na osnovu zbira intervala [i, j-1]
      zbir += a[j];
      // ako je zbir jednak trazenom, prijavljujemo interval
      if (zbir == trazeniZbir)
        cout << i << " " << j << endl;
    }
  }

  return 0;
}
```

Техника два показивача

Претрагу сегмената грубом силом можемо коришћењем одсецања начинити много ефикаснијом.

Посматрајмо пример проналажења првог сегмента у низу `1 2 3 5 15 1 2 5` који има збир `21`.

Крећемо да испитујемо збирове сегмената који почињу на позицији 0. Све док је збир текућег сегмента строго мањи од 21, потребно је да проширујемо сегменте.

```
i   a[i]   zbir
0   1      1
1   2      3
2   3      6
3   5      11
4   15     26
```

У тренутку када је збир постао строго већи од тражене вредности 21, сигурни смо да ни један сегмент који почиње на позицији 0 не може имати збир 21. Наиме, пошто су сви даљи елементи стриктно позитивни, њиховим укључивањем би се добио само још већи збир. Због тога можемо да пређемо на сегменте који почињу на позицији 1. Важна (и не баш тривијална) опаска је то да сви сегменти који почињу на позицији 1 и завршавају се пре текуће позиције 4 имају збир строго мањи од 21 и стога их није потребно експлицитно испитивати. Наиме, сви сегменти који почињу на позицији 0, а завршавају се пре текуће позиције су имали збир мањи од 21, па се уклањањем елемента на позицији 0 добијају сегменти чији је збир још мањи. Дакле, први кандидат за збир 21, је сегмент који почиње на позицији 1 и завршава се на позицији 4. Његов збир лако добијамо одузимањем почетне вредности 1 са позиције 0 од збира текућег сегмента.

```
i   a[i]   zbir
1   2      2
2   3      5
```

```
3   5      10
4   15     25
```

Пошто и тај сегмент има збир већи од 21, такав збир ће имати и сви даљи сегменти који почињу на позицији 1, па можемо прећи на сегменте који почињу на позицији 2. Поново је први кандидат онај који се завршава на позицији 4 (јер сви који се раније завршавају сигурно имају збир мањи од 21).

```
i   a[i]   zbir
2   3      3
3   5      8
4   15     23
```

Поново је збир превелики, па прелазимо на сегменте који почињу на позицији 3. Први кандидат је сегмент који се завршава на позицији 4.

```
i   a[i]   zbir
3   5      5
4   15     20
```

Овај пут тај сегмент има збир мањи од траженог, па га је потребно проширити надесно. Додавањем наредног елемента добијамо сегмент чији је збир једнак траженом.

```
i   a[i]   zbir
3   5      5
4   15     20
5   1      21
```

Након овога смо сигурни да нема више сегмената траженог збира који почињу на позицији 3, па прелазимо на позицију 4 и поступак се по истом принципу наставља даље.

Дакле, одржавамо текући сегмент и његов збир. Док је тај збир мањи од траженог проширујемо сегмент надесно (док је то могуће), а када збир постане већи или једнак траженом скраћујемо сегмент са леве стране.

Докажимо и формално да је претходни поступак коректан. Обележимо са $z_{ij} = \sum_{k=i}^{j} a_k$ збир елемената низа a чији индекси припадају сегменту $[i, j]$, а са z тражени збир елемената. Пошто су сви елементи низа a позитивни, збирови елемената сегмента задовољавају својство монотоности тј. важи да из $i < i' \leq j$ следи $z_{ij} > z_{i'j}$ и да из $j < j' < n$ следи $z_{ij} < z_{ij'}$.

Претпоставимо да за неки интервал $[i, j]$ знамо да за свако j' такво да је $i \leq j' < j$ важи да је $z_{ij'} < z$. Постоје следећи случајеви за однос z_{ij} и z.

- Прво, ако је $z_{ij} < z$, тада ни за један интервал који почиње на позицији i, а завршава се најкасније на позицији j не може важити да му је збир елемената z, и проверу је потребно наставити од интервала $[i, j + 1]$, увећавајући j за 1. Ако такав интервал не постоји (ако је $j + 1 = n$), онда се претрага може завршити (јер је и за свако i' такво да је $i < i' \leq j = n - 1$ важи $z_{i'j} < z_{ij} < z$, а зато и за свако j' такво да је $i' \leq j' < j = n - 1$ важи да је $z_{i'j'} < z_{i'j} < z$, тако да за сваки интервал $[i', j']$ такав да је $i \leq i' \leq j' < n$ важи да је $z_{i'j'} < z$).
- Друго, претпоставимо да је $z_{ij} \geq z$. Ако је $z_{ij} = z$, тада је пронађен један задовољавајући интервал и потребно је обрадити његове границе i и j. То је једини сегмент који почиње на позицији i са збиром z (ако је $z_{ij} > z$, онда таквих сегмената нема). Наиме, пошто су сви елементи низа a позитивни, за свако j'' такво да је $j < j'' < n$ важи да је $z \leq z_{ij} < z_{ij''}$. Дакле, претрагу можемо наставити увећавајући вредност i. За све вредности j' такве да је $i + 1 \leq j' < j$ важи да је $z_{(i+1)j'} < z$. Наиме, пошто је $a_i > 0$ важи да је $z_{(i+1)j'} < z_{ij'} < z$. Дакле, на сегмент $[i + 1, j]$ може се применити анализа случајева истог облика као на интервал $[i, j]$.

Приликом имплементације одржаваћемо интервал $[i, j]$ и његов збир ћемо израчунавати инкрементално - приликом повећања броја j збир ћемо увећавати за a_j, а приликом повећања броја i збир ћемо умањивати за a_i (сличну технику смо користили, на пример, у задатку Префикс највећег збира).

Имплементација са једном петљом

Један начин да се на основу претходне анализе направи имплементација је да се у сваком кораку петље одржавају две променљиве i и j и променљива $zbir$. Обезбедићемо да при сваком уласку у тело петље важи да променљива $zbir$ чува текућу вредност z_{ij} и да је за свако $j' < j$ испуњено да је $z_{ij'} < z$. Ако се i и j иницијализују на нулу, тада се $zbir$ треба иницијализовати на a_0, чиме се задовољава претходни услов.

У телу петље проверавамо да ли је $zbir$ мањи од траженог и ако јесте, увећавамо вредност j. Ако j достигне вредност n, тада можемо прекинути петљу и завршити претрагу. Ако је увећано мање од n, онда збир увећавамо за a_j и прелазимо на наредни корак петље (услов који смо наметнули да важи при уласку у петљу ће бити овим бити задовољен).

Ако вредност променљиве $zbir$ није мања од тражене вредности z проверавамо да ли јој је једнака. Ако јесте, пријављујемо пронађени интервал $[i, j]$. Затим прелазимо на обраду наредног интервала тако што збир умањујемо за вредност a_i и i увећавамо за 1 (услов који смо наметнули да важи при уласку у петљу ће овим бити опет задовољен).

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

using namespace std;

int main() {
  // ubrzavamo ucitavanje
  ios_base::sync_with_stdio(false);

  // ucitavamo trazeni zbir
  int traženiZbir;
  cin >> traženiZbir;

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

  // granice segmenta
  int i = 0, j = 0;
  // zbir segmenta
  int zbir = a[0];
  while (true) {
    // na ovom mestu vazi da je zbir = sum(ai, ..., aj) i da
    // za svako i <= j' < j vazi da je sum(ai, ..., aj') < traženiZbir

    if (zbir < traženiZbir) {
      // prelazimo na interval [i, j+1]
      j++;
      // ako takav interval ne postoji, zavrsili smo pretragu
      if (j >= n)
        break;
      // izracunavamo zbir intervala [i, j+1] na osnovu zbira intervala [i, j]
      zbir += a[j];
    } else {
      // ako je zbir jednak trazenom, vazi da je sum(ai, ..., aj) = traženiZbir
      // pa prijavljujemo interval
      if (zbir == traženiZbir)
        cout << i << " " << j << endl;
      // prelazimo na interval [i+1, j]
      // izracunavamo zbir intervala [i+1, j] na osnovu zbira intervala [i, j]
      zbir -= a[i];
      i++;
    }
  }

  return 0;
```

```
}
```

Имплементација са угнежђеним петљама

Још једна могућа имплементација садржи две угнежђене петље. У првој десни крај сегмента померамо надесно све док не стигнемо до краја или док је збир текућег сегмента мањи од траженог. У другој леви крај сегмента померамо надесно све док је збир већи или једнак од траженог (при том проверавајући да ли је збир једнак траженом и ако јесте, пријављујући пронађени интервал).

Иако садржи угнежђене петље, ово решење је линеарне сложености у односу на дужину низа тј. сложености је $O(n)$. Наиме, промењиве i и j се у сваком кораку увећавају за један и никада се не умањују, тако да је укупан број корака ограничен вредношћу $2n$.

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

using namespace std;

int main() {
  // ubrzavamo ucitavanje
  ios_base::sync_with_stdio(false);

  // ucitavamo trazeni zbir
  int trazeniZbir;
  cin >> trazeniZbir;

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


  int i = 0, j = 0; // granice segmenta
  int zbir = 0;     // zbir segmenta [i, j-1]
  while (j < n) {
    // prosirujemo segment nadesno dok god je zbir manji od trazenog
    while (j < n && zbir < trazeniZbir) {
      // dodajemo novi element
      zbir += a[j];
      j++;
    }

    // skracujemo interval sve dok je zbir veci od trazenog
    while (zbir >= trazeniZbir) {
      // ako je zbir intervala jednak trazenom ispisujemo
      // pronadjeni interval
      if (zbir == trazeniZbir)
        cout << i <<  " " <<  j - 1 << endl;
      // uklanjamo pocetni element
      zbir -= a[i];
      i++;
    }
  }

  return 0;
}
```

Имплементација у којој се обрађује сваки десни крај сегмента

Још један начин имплементације је да у петљи обрађујемо све вредности десног краја j од 0 до $n-1$. Обезведићемо да при сваком уласку у тело петље променљива $zbir$ садржи вредност $z_{i(j-1)}$ и да је та вредност строго мања од тражене вредности збира z. Пошто i иницијализујемо на 0, збир је такође потребно иницијализовати на нулу, чиме се тражени услов обезбеђује (јер је тражена вредност строго позитивна).

На почетку циклуса увећавамо $zbir$ за вредност a_j и тако је постављамо на вредност z_{ij}.

Док је збир већи од траженог увећавамо леви крај интервала i и збир умањујемо за a_i све док збир не постане мањи или једнак траженом (ако је збир у почетку једнак траженом, ово померање левог краја се неће ни једном извршити). Таква вредност збира ће се сигурно достићи у неком тренутку (када i постане веће од j тада ће се збир спустити до нуле, што је сигурно мање од тражене вредности). Након тога проверавамо да ли је збир једнак траженом и ако јесте, пријављујемо да смо пронашли одговарајући интервал $[i, j]$, увећавамо i за један и умањујемо збир за вредност a_i. Након овога ће сигурно важити да је збир мањи од траженог и да ће при евентуалном наредном уласку у тело петље бити једнак вредности $z_{i(j-1)}$ (за увећану вредност j).

Рецимо и да, ако је $zbir$ мањи од траженог, није потребно ништа додатно урадити, јер ће у евентуалном наредном уласку у петљу тражени услов бити испуњен ($zbir$ садржати вредност $z_{i(j-1)}$, за увећану вредност j, која је мања тражене вредности).

Из истих разлога као и претходно и ово решење је линеарне сложености у односу на дужину низа тј. сложености $O(n)$.

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

using namespace std;

int main() {
  // ubrzavamo ucitavanje
  ios_base::sync_with_stdio(false);

  // ucitavamo trazeni zbir
  int trazeniZbir;
  cin >> trazeniZbir;

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

  int i = 0;      // pocetak intervala
  int zbir = 0;   // zbir segmenta
  for (int j = 0; j < n; j++) {
    // na ovom mestu vazi da je  zbir = sum(ai, ..., aj-1) < trazeniZbir

    // izracunavamo zbir intervala [i, j] na osnovu zbira intervala [i, j-1]
    zbir += a[j]; // sada vazi da je zbir = sum(ai, ..., aj)

    // dok je zbir intervala [i, j] veci od trazenog, umanjujemo ga
    // suzavanjem intervala s leve strane
    while (zbir > trazeniZbir) {
      zbir -= a[i];
      i++;
      // ostaje da vazi da je zbir = sum(ai, ..., aj)
    }
    // sada je zbir = sum(ai, ..., aj) <= trazeniZbir

    // ako je zbir intervala [i, j] jednak trazenom
    if (zbir == trazeniZbir) {
```

```
        // znamo da je zbir = sum(ai, ..., aj) = trazeniZbir, pa
        // prijavljujemo interval [i, j]
        cout << i <<  " " <<  j << endl;
        // prelazimo na interval [i+1, j], azurirajuci zbir
        zbir -= a[i];
        i++;
    }
    // na ovom mestu znamo da je zbir = sum(ai, ..., aj) < trazeniZbir
  }
  return 0;
}
```

Задатак: Најкраћа подниска која садржи све дате карактере

Графички дизајнер је преуредио неколико слова у једном фонту и жели да своје промене прикаже клијенту. У дугачком тексту је потребно да одабере најкраћи део (сегмент узастопних слова) који садржи сва слова која је променио.

Улаз: У првој линији стандардног улаза налази се текст (једноставности ради претпоставимо да је састављен само од малих слова енглеског алфабета) чија је дужина највише 50000 карактера. У другој линији се налази скуп слова (опет, претпоставимо малих слова енглеског алфабета) које је дизајнер променио (слова су написана једно до другог, без размака и без понављања).

Излаз: На стандардни излаз исписати један цео број који представља дужину најкраћег дела текста који садржи све карактере датог скупа. Ако такав део текста не постоји, исписати `nema`.

Пример 1		**Пример 2**	
Улаз	*Излаз*	*Улаз*	*Излаз*
`dobardansvimakakoste` `arnk`	`10`	`abababababab` `abc`	`nema`

Решење

Овај задатак представља једноставно уопштење задатка Конференција.

Техника два показивача

Бољи алгоритам можемо конструисати тако што и даље обрађујемо све релевантне позиције редом, са лева на десно, али за сваку од њих, уместо најкраћег подтекста који на њој почиње, проналазимо најкраћи подтекст који се на њој завршава (дуално, могли бисмо да тражимо најкраће подтекстове који почињу на свакој позицији, као што смо то радили у претходном алгоритму, али да бисмо добили ефикаснији алгоритам, позиције би требало обрађивати са десна на лево). Ако за сваку позицију знамо најкраћи подтекст који се на њој завршава, од свих таквих можемо наћи најкраћи и он ће уједно бити и глобално најкраћи. Опет имамо једноставан аргумент да ће најкраћи подтекст бити пронађен јер се он сигурно завршава на некој позицији и уједно је најкраћи од свих који се на тој позицији завршавају, тако да ће сигурно бити узет у обзир приликом одређивања најмање дужине. Кључне опаске за ефикасно одређивање најкраћег исправног подтекста који се завршава на некој релевантној позицији су то да ћемо увек знати најкраћи исправан подтекст који се завршава на претходној релеватној позицији, као и то да ако се први релевантан карактер у подтексту јавља бар још једном касније у подтексту, онда тај подтекст не може бити најкраћи који садржи све карактере скупа S (заиста, почетно парче тог подтекста све до следећег карактера из скупа S се може уклонити и опет ће се добити исправан подтекст тј. подтекст који садржи све потребне карактере).

Алгоритам ће прво пронаћи први исправан подтекст (ако такав постоји). Затим ће се обрађивати једна по једна релевантна позиција надесно, и за њу ће се одређивати најкраћи исправан подтекст који се на њој завршава, тако што ће се кретати од најкраћег исправног подтекста који се завршава на претходној позицији, затим ће се он проширивати надесно да укључи карактере до тренутне позиције и затим ће се из тог подтекста избацивати делови са почетка, све док се не наиђе на релевантан карактер који се не јавља касније у текућем подтексту. Тај подтекст мора бити најкраћи од свих исправних подтекста који се завршавају на тренутној позицији јер би се избацивањем овог карактера изгубило својство да подтекст садржи све карактере из скупа S тј. подтекст не би више био исправан.

У примеру `xCxxBxxBxxAxxBxxCxxxxBxAxAxCxxxBx` разматрали бисмо подтекст `C`, затим `CxxB`, затим `CxxBxxB` и затим `CxxBxxBxxA` који садржи све потребне карактере (он је најкраћи од свих подтекста који се завршавају

на тој позицији).

Преласком на следећу релевантну позицију добио би се подтекст CxxBxxBxxAxxB који је најкраћи од свих који се на њој завршава (зато што се почетно C јавља само један пут и не сме се уклонити).

Следећа позиција је позиција наредног слова C. Конструкцију најкраћег подтекста који се завршава на тој позицији започињемо тако што проширимо претходни подтекст надесно и добијамо CxxBxxBxxAxxBxxC. Затим уклањамо почетно C и карактере x иза њега (јер се C јавља у подтексту и касније), чиме добијамо BxxBxxAxxBxxC, затим уклањамо и Bxx (јер се B још два пута јавља у подтексту) и поново уклањамо Bxx (јер се B још једном јавља у подтексту). Када се дође до подтекста AxxBxxC он је најкраћи који се завршава на позицији тог слова C јер се A јавља само на његовом почетку и не сме се уклонити.

Истим поступком добија се да је најкраћи подтекст који се завршава на наредној позицији подтекст AxxBxxCxxxxB, на наредној позицији је то подтекст CxxxxBxA, на наредној позицији је то CxxxxBxAxA, затим BxAxAxC, и на крају AxCxxxB. Пошто смо за сваку позицију пронашли најкраће подтексте, можемо одредити и дужину глобално најкраћег и то је 7 (ту дужину имају подтекстови AxxBxxC, BxAxAxC и AxCxxxB).

Тренутни подтекст биће одређен помоћу два итератора i и j низа релевантних позиција, а пошто за сваки карактер треба да знамо колико пута се јавља у подтексту уместо скупа карактера тренутног подтекста који смо користили у прошлом алгоритму користићемо мапу која сваки карактер пресликава у његов број појављивања.

Оценимо сада и сложеност овог алгоритма. Спољна петља по j пролази кроз све релевантне позиције којих има највише $O(n)$. Кључна опаска за оцену сложености је да се унутрашња петља (у којој се врши скраћивање сегмента који се завршава на позицији j) може извршити укупно $O(n)$ пута (i је све време мање или једнако j и стално се увећава, а никада не умањује). У оквиру петљи врши се ажурирање вредности у мапи, али с обзиром на то да та операција има логаритамску сложеност (или чак константну, пошто је распон типа char ограничен), а број карактера у скупу S је мали, можемо рећи да је сложеност алгоритма $O(n)$, тј. да је алгоритам линеаран у односу на дужину текста.

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

using namespace std;

int main() {
  string T, S;
  getline(cin, T);
  getline(cin, S);

  vector<int> poz_karaktera_iz_S;
  for (int i = 0; i < T.size(); i++)
    // ako se T[i] nalazi u skupu S
    if (S.find(T[i]) != string::npos)
      // zapamti njegovu poziciju i
      poz_karaktera_iz_S.push_back(i);

  // najmanja duzina podniske
  int min_duzina = numeric_limits<int>::max();
  // broj pojavljivanja svakog relevantnog karaktera u trenutnoj podniski
  map<char, int> broj_pojavljivanja_u_podniski;
  // Trenutna podniska je odredjena pozicijama [i, j]
  vector<int>::const_iterator i, j;
  for (i = j = poz_karaktera_iz_S.begin(); j != poz_karaktera_iz_S.end(); j++){
    // trenutnu podnisku prosirujemo do sledece pozicije j i
    // uvecavamo broj pojavljivanja karaktera koji se nalazi na
    // poziciji odredjenoj sa j (jer se i on sada javlja u podniski)
    broj_pojavljivanja_u_podniski[T[*j]]++;
```

```
    // ako mapa ima isto elemenata kao i skup S, onda su svi elementi
    // skupa S prisutni u podtekstu
    if (broj_pojavljivanja_u_podniski.size() == S.size()) {
      // trazimo najkracu podnisku koji se zavrsava na poziciji
      // odredjenoj sa j tako sto podtekst skracujemo sa leve strane
      // dok god se prvi karakter podteksta javlja vise puta u njemu
      while (broj_pojavljivanja_u_podniski[T[*i]] > 1) {
        // podtekst skracujemo i uklanjamo sve karaktere sa pocetka sve do
        // sledeceg karaktera iz skupa S
        broj_pojavljivanja_u_podniski[T[*i]]--;
        i++;
      }
      // izracunavamo duzinu trenutnog podteksta
      int duzina = *j - *i + 1;
      // azuriramo minimum ako je to potrebno
      if (duzina < min_duzina)
        min_duzina = duzina;
    }
  }
  // prijavljujemo rezultat
  if (min_duzina != numeric_limits<int>::max())
    cout << min_duzina << endl;
  else
    cout << "nema" << endl;

  return 0;
}
```

Задатак: Двоструко сортирана претрага

Дата је матрица у којој су све врсте и све колоне сортиране растући. Напиши програм који ефикасно проналази елементе у таквој матрици.

Улаз: Са стандардног улаза уносе се димензије матрицеm и n ($1 \leq m, n \leq 1000$), а затим и елементи матрице (елементи сваке врсте у посебном реду, раздвојени размацима). Елементи су цели бројеви између -10^5 и 10^5. Након тога учитава се у сваком реду до краја улаза по један број који се тражи у матрици.

Излаз: За сваки број који се тражи у матрици на стандардни излаз исписати колико пута се појављује у матрици.

Пример

Улаз	*Излаз*
4 5 1 3 5 8 10 4 7 9 11 15 5 9 13 14 20 8 11 14 16 22 11 12 13	2 0 1

Решење

Линеарна претрага

Наиван начин је да сваки елемент тражимо линеарном претрагом (као, на пример, у задатку Негативан број), тако што у угнежђеним петљама пролазимо кроз сваки елемент матрице. Сложеност овог приступа била би $O(m \cdot n)$.

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

```
using namespace std;

vector<vector<int>> ucitajMatricu() {
  int m, n;
  cin >> m >> n;
  vector<vector<int>> A(m);
  for (int i = 0; i < m; i++) {
    A[i].resize(n);
    for (int j = 0; j < n; j++)
      cin >> A[i][j];
  }
  return A;
}

int brojPojavljivanja(const vector<vector<int>>& A, int x) {
  int broj = 0;
  int m = A.size(), n = A[0].size();
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++)
      if (A[i][j] == x)
        broj++;
  }
  return broj;
}

int main() {
  ios_base::sync_with_stdio(false);
  auto A = ucitajMatricu();
  int x;
  while (cin >> x)
    cout << brojPojavljivanja(A, x) << endl;
  return 0;
}
```

Бинарна претрага

Пошто су елементи сваке врсте сортирани, на претрагу сваке врсте могуће је применити поступак бинарне претраге (приказан у задатку Провера бар-кодова). Ако претрагу вршимо по врстама, онда можемо употребити библиотечку функцију. Сложеност тог приступа је $O(m \cdot \log n)$. Ако има много више врста него колона, ефикасније би било претрагу вршити по колонама (сложеност би тада била $O(n \cdot \log m)$), али тада не бисмо могли да применимо библиотечку функцију.

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

using namespace std;

// ovo je dovoljno efikasno zbog move semantike c++-11
vector<vector<int>> ucitajMatricu() {
  int m, n;
  cin >> m >> n;
  vector<vector<int>> A(m);
  for (int i = 0; i < m; i++) {
    A[i].resize(n);
    for (int j = 0; j < n; j++)
      cin >> A[i][j];
  }
```

```
  return A;
}

int brojPojavljivanja(const vector<vector<int>>& A, int x) {
  int broj = 0;
  int m = A.size(), n = A[0].size();
  for (int i = 0; i < m; i++)
    if (binary_search(A[i].begin(), A[i].end(), x))
      broj++;
  return broj;
}

int main() {
  ios_base::sync_with_stdio(false);
  auto A = ucitajMatricu();
  int x;
  while (cin >> x)
    cout << brojPojavljivanja(A, x) << endl;

  return 0;
}
```

Оптимизована бинарна претрага

У решењу са бинарном претрагом смо искористили чињеницу да су врсте сортиране, али не и да су колоне сортиране. На основу тога, можемо оптимизовати бинарну претрагу. Уместо претраге за тачним елементом, у сваком кораку можемо вршити претрагу за први елементом који је већи или једнак траженом (као, на пример, у задатку Први већи и последњи мањи). У наредној врсти претрагу можемо вршити само од почетка па до позиције на којој се у претходној врсти налазио елемент такав елемент (јер се у наредној врсти, због сортираности колона, на тој позицији налази елемент који је строго већи од траженог, а због сортираности колона, такви су и сви елементи те врсте иза те позиције). И даље је у најгорем случају сложеност $O(n \cdot \log m)$.

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

using namespace std;

// ovo je dovoljno efikasno zbog move semantike c++-11
vector<vector<int>> ucitajMatricu() {
  int m, n;
  cin >> m >> n;
  vector<vector<int>> A(m);
  for (int i = 0; i < m; i++) {
    A[i].resize(n);
    for (int j = 0; j < n; j++)
      cin >> A[i][j];
  }
  return A;
}

int brojPojavljivanja(const vector<vector<int>>& A, int x) {
  int broj = 0;
  int m = A.size(), n = A[0].size();
  for (int i = 0; i < m && n > 0; i++) {
    auto end = next(A[i].begin(), n);
    auto it = lower_bound(A[i].begin(), end, x);
    if (it != end) {
```

```
      n = distance(A[i].begin(), it);
      if (*it == x)
        broj++;
    }
  }
  return broj;
}

int main() {
  ios_base::sync_with_stdio(false);
  auto A = ucitajMatricu();
  int x;
  while (cin >> x)
    cout << brojPojavljivanja(A, x) << endl;

  return 0;
}
```

Претрага из доњег левог угла

Задатак веома елегантно и ефикасно у линеарној сложености можемо решити техником два показивача.

Посматрајмо део полазне матрице (подматрицу) у чијем се доњем левом углу налази елемент a_{vk} и који се простире до горњег десног угла полазне матрице. Пошто су елементи у врстама и колонама строго растући, сви елементи у колони k изнад елемента a_{vk} су мањи од њега, док су сви елементи у врсти v десно од елемента a_{vk} већи од њега.

На почетку посматрамо целу матрицу (тј. почињемо од елемента a_{vk} за $v = m - 1$ и $n = 0$).

Ако је елемент a_{vk} мањи од траженог, тада се тражени елемент не може налазити у последњој врсти посматране подматрице и претрагу можемо наставити у подматрици којој је доњи леви угао $a_{(v-1)k}$. Ако је нова подматрица празна (што се дешава када је $v = 0$), претрагу није потребно даље настављати. Претрагу нове подматрице настављамо на потпуно исти начин као и полазне (тако што умањимо индекс v за 1).

Ако елемент a_{vk} већи од траженог, тада се тражени елемент не може налазити у врсти у првој колони посматране подматрице и претрагу можемо наставити у подматрици којој је доњи леви угао $a_{v(k+1)}$. Ако је нова подматрица празна (што се дешава када је $k = n - 1$), претрагу није потребно даље настављати. Претрагу нове подматрице настављамо на потпуно исти начин као и полазне (тако што увећамо индекс k за 1).

Ако је елемент a_{vk} једнак траженом, тада можемо увећати бројач појављивања траженог елемента. Пошто су елементи у врстама и колонама строго растући, знамо да се елемент не може налазити ни изнад ни десно од елемента a_{vk}, тако да наставак претраге можемо наставити у подматрици којој је доњи десни угао $a_{(v-1)(k+1)}$. Ако је нова подматрица празна (што се дешава када је $k = n - 1$), претрагу није потребно даље настављати. Претрагу нове подматрице настављамо на потпуно исти начин као и полазне (тако што увећамо индекс k за 1).

Сложеност овог приступа у најгорем случају је $O(m + n)$.

Напоменимо да је ово решење лошије од бинарне претраге када имамо мали број прилично дугачких врста.

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

using namespace std;

vector<vector<int>> ucitajMatricu() {
  int m, n;
  cin >> m >> n;
  vector<vector<int>> A(m);
  for (int i = 0; i < m; i++) {
    A[i].resize(n);
    for (int j = 0; j < n; j++)
      cin >> A[i][j];
```

```
  }
  return A;
}

int brojPojavljivanja(const vector<vector<int>>& A, int x) {
  int broj = 0;
  int m = A.size(), n = A[0].size();
  int v = m - 1, k = 0;
  while (v >= 0 && k < n)
    if (A[v][k] < x)
      k++;
    else if (A[v][k] > x)
      v--;
    else {
      broj++;
      v--; k++;
    }
  return broj;
}

int main() {
  ios_base::sync_with_stdio(false);
  auto A = ucitajMatricu();
  int x;
  while (cin >> x)
    cout << brojPojavljivanja(A, x) << endl;
  return 0;
}
```

Претходни поступак сасвим природно може бити описан и рекурзивном функцијом.

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

using namespace std;

vector<vector<int>> ucitajMatricu() {
  int m, n;
  cin >> m >> n;
  vector<vector<int>> A(m);
  for (int i = 0; i < m; i++) {
    A[i].resize(n);
    for (int j = 0; j < n; j++)
      cin >> A[i][j];
  }
  return A;
}

int brojPojavljivanja(const vector<vector<int>>& A, int x, int v, int k) {
  if (v < 0 || k >= (int)A[0].size())
    return 0;
  if (A[v][k] < x)
    return brojPojavljivanja(A, x, v, k+1);
  else if (A[v][k] > x)
    return brojPojavljivanja(A, x, v-1, k);
  else
    return 1 + brojPojavljivanja(A, x, v-1, k+1);
}
```

```
int main() {
  ios_base::sync_with_stdio(false);
  auto A = ucitajMatricu();
  int m = A.size();
  int x;
  while (cin >> x)
    cout << brojPojavljivanja(A, x, m-1, 0) << endl;
  return 0;
}
```