https://arena.petlja.org/competition/2021probnikontrolnibinarnapretraga

Бинарна претрага - пробни контролни задатак

1.

Vremensko ograničenje	Memorijsko ograničenje	ulaz	izlaz
0,1 s	64 MB	standardni ulaz	standardni izlaz

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

```
5
6
1
4
3
6
-1
5
```

Излаз

```
2
```

То су парови $(1,4)$ и $(6,-1)$.

Решавање методом грубе силе

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 бројачка променљива i узима вредности од 0 до n - 2, а бројачка промељива j узима вредности од i + 1 до n - 1.

Парове можемо пребројати унутар 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;
}
```

#	Status	Poeni	Vreme	Memorija
1	OK	1	0,00s	1,00MB
2	OK	1	0,00s	1,00MB
3	OK	1	0,03s	2,00MB
4	OK	1	0,06s	2,00MB
5	OK	1	0,05s	2,00MB
6	TLE	-	>0,11s	-
7	TLE	-	>0,11s	-
8	TLE	-	>0,11s	-
9	TLE	-	>0,11s	-
10	TLE	-	>0,11s	-
11	TLE	-	>0,11s	-

Решавање методом бинарне претраге

Можемо сортирати улазни низ у растућем поретку.

Елемент a[j] = s - a[i] можемо тражити у низу бинарном претрагом почев од позиције i + 1, јер на тај начин нећемо исти пар бројати два пута.

Приметимо да је важан услов, дат у формулацији задатка, 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)} + ... + log{1}), што је O(log{(n-1)!})
и за то је могуће показати да је O(n log{n}). (Стирлингова формула)

```
#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;
}
```

#	Status	Poeni	Vreme	Memorija
1	OK	1	0,00s	2,00MB
2	OK	1	0,00s	0,00MB
3	OK	1	0,02s	2,00MB
4	OK	1	0,00s	0,00MB
5	OK	1	0,02s	2,00MB
6	OK	1	0,02s	2,00MB

#	Status	Poeni	Vreme	Memorija
7	OK	1	0,02s	2,00MB
8	OK	1	0,00s	1,00MB
9	OK	1	0,00s	1,00MB
10	OK	1	0,02s	2,00MB
11	OK	1	0,02s	2,00MB

Решавање коришћењем структура података (скуп, бинарно стабло претраге)

Потребно је пронаћи пар (a[i], a[j]) такав да је a[i] + a[j] = s. Један начин на који можемо да брзо извршимо проверу да ли се за сваки елемент a[i] у низу налази њему одговарајући елемент s - a[i] је да елементе уместо у низу чувамо у некој структури података која омогућава ефикасну проверу припадности датог елемента. Најзгоднији тип података за то је скуп.
У језику C++ можемо користити `set` или `unordered_set`.

Ако све елементе убацимо у скуп, сваки пар елемената ће се пронаћи два пута и коначно решење ће бити једнако половини пронађених парова. Обратимо пажњу на то да ако се половина броја s налази у скупу, тада ће бити пронађен и вештачки пар (s/2, s/2), па коначан број парова можемо добити заокруживањем количника пронађеног броја
парова и броја 2 наниже.

Сложеност уметања и претраге оваквих структура података је обично највише логаритамска у односу на број елемената, и пошто се уметање и претрага врше n пута, укупна сложеност оваквог приступа је O(n log{n}).
Ако се користи скуп заснован на хеширању, сложеност уметања и претраге елемента у том скупу се може сматрати константном (мада је сложеност најгорег случаја заправо линеарна), па се онда сложеност целог алгоритма може сматрати линеарном тј. O(n), али уз значајан константни фактор.

```
#include <iostream>
#include <set>

using namespace std;

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

  int s, n;
  cin >> s >> n;
  set<int> a;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    a.insert(x);
  }

  int brojParova = 0;
```

```
  for (int x : a)
    if (a.find(s - x) != a.end())
      brojParova++;

  cout << brojParova / 2 << endl;
  return 0;
}
```

Решавање коришћењем структура података (скуп, бинарно стабло претраге)

Још бољи начин да употребимо скуп је да за сваки елемент a[i] који учитамо проверимо да ли се њему одговарајући елемент налази у скупу претходно учитаних елемената тј. да током рада чувамо скуп претходно учитаних елемената, у петљи учитавамо један по један елемент, проверавамо да ли чини пар са неким претходно учитаним елементом и увећавамо бројач ако чини, и након тога га додајемо у скуп претходно учитаних елемената.

Ако претпоставимо да се уметање елемента у скуп и провера припадности елемента скупу од k елемената може урадити у времену O(log{k}),тада је цео алгоритам сложености O(n log{n})

```
#include <iostream>
#include <set>

using namespace std;

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

  int s, n;
  cin >> s >> n;

  int brojParova = 0;
  set<int> prethodni;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    if (prethodni.find(s - x) != prethodni.end())
      brojParova++;
    prethodni.insert(x);
  }

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

2.

Vremensko ograničenje	Memorijsko ograničenje	ulaz	izlaz
0,1 s	64 MB	standardni ulaz	standardni izlaz

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
6
-3
-1
1
3
5
7
Излаз
3

Решавање методом линеарне претраге

Директан начин да се задатак реши је да се употреби линеарна претрага 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;
}
```

Решавање методом бинарне 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 <= i < j.
Пошто је низ a строго растући, важи да је a[i+1] > a[i], па је a[i+1] >= a[i] +1
Слично је a[i+2] > a[i+1] па је a[i+2] >= a[i+1] + 1 >= a[i] + 2.

Настављањем овог резона важи да је a[j] >= a[i] + j
Зато је a[j]- j >=a[i] >= 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`

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

Слично, даља претрага за првим елементом који задовољава услов је у најгорем случају сложености O(n), што поништава предности бинарне претраге.

```
#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;
}
```

Решавање методом бинарне претраге без трансформисања низа

Бинарна претрага се може прилагодити и применити тако да се не модификује низ.

Један могући покушај имплементације прати основну варијанту бинарне претраге у којој се тражи елемент унутар низа. Након налажења средишњег елемента можемо проверити

- да ли је a[s] < s (тада би у трансформисаном низу важило да је a[s]- s < 0, па је претрагу потребно наставити десно од позиције s),
- да ли је a[s] > s (тада би у трансформисаном низу важило да је a[s] - s > 0, па је претрагу потребно наставити лево од позиције s)
- или је a[s] = s и тада је елемент пронађен.

Када се пронађе неки елемент који задовољава дати услов, потребно је проверити да ли је најмањи. Пошто други елементи који задовољавају дати услов могу бити само непосредно уз пронађени елемент, најмањи елемент који задовољава услов проналазимо крећући се улево тј. мењајући текући елемент њему претходним све док тај претходни елемент задовољава тражени услов (или док евентуално не дођемо до самог почетка низа). Нагласимо да је оваква имплементација бинарне претраге заправо лоша, јер у случају када у низу постоји велики број елемената који су на свом месту, померање улево може трајати дуго.

Сложеност бинарне претраге је O(log{n}), међутим, даља претрага за првим елементом који задовољава услов је у најгорем случају сложености O(n), што поништава предности бинарне претраге. Временом свакако доминира учитавање низа које захтева O(n) корака.

```
#include <iostream>
#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];

  // sprovodimo binarnu pretragu - ako trazeni element a[k] = k
  // postoji, on se nalazi u intervalu [l, d]
  int l = 0, d = n-1;
  // dok interval [l, d] nije prazan
  while (l <= d) {
    // pronalazimo sredinu intervala
    int s = l + (d - l) / 2;
    if (a[s] < s)
      // trazeni element se moze nalaziti samo desno od s
      l = s + 1;
    else if (a[s] > s)
      // trazeni element se moze nalaziti samo levo od s
      d = s - 1;
    else {
      // pronasli smo element na poziciji s
      // mozda postoji i neki pre njega?
      // Opasnost - ovo može biti linearna pretraga
      while (s - 1 >= 0 && a[s - 1] == s - 1)
        s = s - 1;
      cout << s << endl;
      break;
    }
  }

  // ako se interval [l, d] ispraznio, element ne postoji
  if (l > d)
    cout << "nema" << endl;

  return 0;
}
```

Решавање методом бинарне претраге преломне тачке

Боља имплементација бинарне претраге је она у којој се приналази најмања позиција s за коју важи да је a[s] >= s, а онда да се провери да ли се на њој налази елемент који је једнак s. Дакле, у питању је претрага преломне тачке (прве позиције у којој почиње да важи неки услов).
Сложеност бинарне претраге је O(log{n}). Међутим, временом доминира учитавање низа које захтева O(n) корака.

```
#include <iostream>
#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];

  // sprovodimo binarnu pretragu - ako trazeni element a[k] = k
  // postoji, on se nalazi u intervalu [l, d]
  int l = 0, d = n-1;
  // dok interval [l, d] nije prazan
  while (l <= d) {
    // pronalazimo sredinu intervala
    int s = l + (d - l) / 2;
    if (a[s] >= s)
      // trazeni element se moze nalaziti samo levo od s
      d = s - 1;
    else
      // trazeni element se moze nalaziti samo desno od s
      l = s + 1;
  }
  // l je prva pozicija za koju je a[i] >= i

  if (l < n && a[l] == l)
    cout << l << endl;
  else
    cout << "nema" << endl;

  return 0;
}
```

Коришћење библиотечких функција

Имплементација може бити урађена и коришћењем библиотечке функције `upper_bound`. Пошто унутар функције поређења морамо одредити индекс тренутног елемента, тај елемент мора бити прослеђен по референци. Индекс елемента у низу затим можемо одредити израчунавањем разлике меморијске адресе тог елемента и првог елемента у низу (у питању је такозвана показивачка аритметика, својствена за програмски језик C).

Сложеност бинарне претраге је O(log{n}). Међутим, временом доминира учитавање низа које захтева O(n) корака.

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

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];

  auto it = upper_bound(begin(a), end(a), 0,
                        [&a](int _, const int& x) {
                          int i = &x - &a[0];
                          return a[i] >= i;
                        });

  int i = distance(begin(a), it);
  if (i < n && a[i] == i)
    cout << i << endl;
  else
    cout << "nema" << endl;

  return 0;
}
```

3.

Гласници

Vremensko ograničenje	Memorijsko ograničenje	ulaz	izlaz
1 s	64 MB	standardni ulaz	standardni izlaz

Дуж једног пута налазе се гласници. Поруку први сазнаје гласник на почетку пута и циљ је да сви гласници што пре сазнају поруку. Сваки гласник може викањем пренети поруку свима који се од њега налазе на растојању мањем од задатог домета гласа (истог за све гласнике). При том се сви гласници могу кретати брзином од највише једног метра у секунди у било ком смеру (током времена могу мењати своју брзину и смер кретања, а могу и стајати у месту).

Улаз

Са стандардног улаза се учитава домет гласа d **(реалан број,** $1 \leq d \leq 10^6$**), затим број гласника** n **(цео број,** $1 \leq n \leq 10^5$**), а након тога положај сваког гласника** g_i **(реалан број,** $0 \leq g_i \leq 10^9$**, који представља растојање од почетка пута).**

Излаз

На стандардни излаз исписати најмање време протекло од тренутка када први гласник сазна поруку до тренутка када поруку сазнају сви гласници, као реалан број заокружен на три децимале (одступање од тачне вредности сме да буде највише 10^{-3}**).**

Пример 1

Улаз

```
3.000
2
0.000
6.000
```

Излаз

```
1.500
```

Пример 2

Улаз

```
2.000
4
0.000
4.000
4.000
8.000
```

Излаз

```
1.000
```

Решавање методом линеарне претраге

Када на располагању имамо функцију за проверу да ли је поруку могуће пренети у датом времену, наиван начин је да време мало по мало повећавамо (0.001 - узевши у обзир тражену прецизност решења), све док се први пут не догоди да је поруку могуће пренети.

Време потребно да се провери да ли сви гласници могу да приме поруку током времена t захтева један пролаз кроз низ гласника и износи O(n). Број позива функције зависи од броја децимала k (и то фактором 10^k) и од оптималне вредности времена, а оно је ограничено

растојањем између првог по последњег гласника. Ако је то растојање X, тада се сложеност најгорег случаја може проценити на $O(10^k X n).

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

using namespace std;

bool moguceJeRasiritiVest(const vector<double>& pozicije,
                          double domet, double vreme) {
  double p = pozicije[0] + vreme;
  for (int i = 1; i < pozicije.size(); i++) {
    if (p + domet < pozicije[i] - vreme)
      return false;
    p = min(pozicije[i] + vreme, p + domet);
  }
  return true;
}

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

  double vreme = 0;
  while (!moguceJeRasiritiVest(pozicije, domet, vreme))
    vreme += 0.001;

  cout << fixed << showpoint << setprecision(3)
       << vreme << endl;

  return 0;
}
```

Решавање методом бинарне претраге
Много ефикасније, оптимално решење можемо пронаћи бинарном претрагом. Најмање могуће време је 0, највеће је једнако разлици између првог и последњег гласника
X = g[n-1] - g[0].
Важно је уочити монотоност низа временских јединица, која је неопходна за бинарну претрагу (ако се порука не може доставити за неко време t, не може се доставити ни за време мање од t. Иначе, ако се може доставити за време t, може се доставити и за време веће од t).

Бинарном претрагом одређујемо узак интервал [l, d] такав да се порука не може пренети за l временских јединица, а може се пренети за d временских јединица и за оптимум проглашавамо средину тог интервала.

Ако се тражи прецизност од k децимала, број позива провере логаритамски зависи од производа 10^k и максималне вредности времена која је одређена растојањем између првог и последњег гласника X.
Пошто се свака провера врши у времену O(n), сложеност се може проценити на O(n log{10^kX)) = O(n klog{X}), што је прилично ефикасно у поређењу са решењем које користи линерану претрагу.

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

using namespace std;

bool moguceJeRasiritiVest(const vector<double>& pozicije,
                          double domet, double vreme) {
  double p = pozicije[0] + vreme;
  for (int i = 1; i < pozicije.size(); i++) {
    if (p + domet < pozicije[i] - vreme)
      return false;
    p = min(pozicije[i] + vreme, p + domet);
  }
  return true;
}

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

  double levo = 0, desno = pozicije[n-1] - pozicije[0];
  while (desno - levo > 0.001) {
    double s = (levo + desno) / 2.0;
    if (moguceJeRasiritiVest(pozicije, domet, s))
      desno = s;
    else
      levo = s;
  }

  cout << fixed << showpoint << setprecision(3)  << (levo + desno) / 2.0 << endl;

  return 0;
}
```