TIP SUBIECT – PROGRAMARE

SUBIECT CLASA a V-a – PROBA PRACTICĂ

Problema SIR:

Enunţ: Micul Gates studiază următorul şir de numere: 1, 1, 1, 2, 2, 1, 1, 2, 3, 3, 2, 1, 1, 2, 3, 4, 4, 3, 2, 1.....

Cerinţe

1. Să se determine valoarea maximă care apare până la poziţia x inclusive, în şir.
2. Să se determine de câte ori apare valoarea y printre primele x elemente din şir.

Date de intrare

Fişierul de intrare: **sir.in** conţine de prima linie numărul c al cerinţei, care poate fi doar 1 sau 2. Pe a doua linie se găsesc două numere naturale nenule x şi y separate printr-un spaţiu, cu semnificaţia din enunţ.

Date de ieşire

Fişierul de ieşire: **sir.out** va conţine, pe prima linie, valoarea determinată conform cerinţei.

Restricţii şi precizări

- 1≤x,y≤1 000 000 000
- Pentru cerinţa 1, se acordă 32 de puncte;
- Pentru cerinţa 2, se acordă 68 de puncte.

Exemple:

sir.in	sir.out	Explicatii
1 **10 3**	**3**	Cerinta este 1, x=10, y=3. Numerele din șir sunt: 1, 1, 1, 2, 2, 1, 1, 2, 3, 3, 2, 1... Valoarea maximă care apare până la poziția 10 este 3.
2 **21 3**	**4**	Cerința este 2, x=21, y=3. Numerele din șir sunt: 1, 1, 1, 2, 2, 1, 1, 2, 3, 3, 2, 1, 1, 2, 3, 4, 4, 3, 2 1 1 2 3…. Numărul 3 apare de 4 ori printre primele 21 de valori din șir.

Editorial

Cerinţa 1 (32 puncte): Observăm că şirul este format din blocuri:

(1, 1),

(1, 2, 2, 1),

(1, 2, 3, 3, 2, 1)

...

(1, 2, 3, ... k, k, ... 3, 2, 1)

Observăm că blocul cu valoarea maximă k conţine numerele de la 1 la k, apoi încă o dată de la k la 1. Astfel, fiecare număr apare de două ori, iar lungimea blocului este 2 * k. Dacă adunăm lungimile blocurilor, obţinem:

Blocul 1: 2 => 2 * 1

Blocul 2: 2 + 4 = 6 => 3 * 2

Blocul 3: 2 + 4 + 6 = 12 => 4 * 3

Blocul 4: 2 + 4 + 6 + 8 = 20 => 5 * 4

...

Blocul k: 2 + 4 + 6 + 8 + ... + 2 * k = k * (k + 1)

Aceste valori reprezintă numărul total de elemente după fiecare bloc complet. Observăm că după blocul cu maximul k există exact: k * (k + 1) elemente. (Sumă Gauss cu raţia 2)

Se poate observa că în fiecare bloc, cele două valori maxime sunt în mijloc.

De exemplu: 1 2 **3 3** 2 1. Poziţia primului element maxim din fiecare bloc este chiar indicele blocului. (Poziţia elementului maxim din blocul k, este tot k.)

De exemplu:

Blocul 3: 1 2 **3** 3 2 1

Poziţii: 1 2 **3** 4 5 6

Astfel, considerând tot şirul, poziţia elementului maxim din fiecare bloc este: k * (k + 1) + k + 1, unde k * (k + 1) este poziţia de început a şirului, iar k + 1 poziţia maximului din şir. După ce dăm factor comun obţinem formula (k + 1) * (k + 1) = $(k + 1)^2$

Valoare:	Prima apariţie:
1	1
2	2 * 2 = 4
3	3 * 3 = 9

4 4 * 4 = 16

Prin urmare, fiecare valoare nouă apare pentru prima dată pe o poziţie care este pătrat perfect.

Deci valoarea maximă care apare în primele x poziţii este cel mai mare număr i pentru care: i * i <= x

Complexitate: O(sqrt(X)).

Cerinţa 2 (68 puncte): Observăm că într-un bloc complet fiecare valoare apare de exact două ori. De exemplu: 1 2 3 3 2 1 valoarea 2 apare de două ori, iar valoarea 3 apare tot de două ori. Prima idee importantă este să găsim primul bloc care conţine valoarea y. Din explicaţia de la cerinţa 1 ştim că prima apariţie a valorii y este pe poziţia: y * y. De aceea, dacă x < y * y înseamnă că încă nu am ajuns la primul y, deci răspunsul este 0.

În cazul în care x >= y * y parcurgem toate blocurile complete pornind de la primul bloc care conţine valoarea y. Poziţia de început a primului bloc care conţine valoarea y este: s = y * (y - 1), unde s reprezintă numărul de elemente aflate înaintea primului bloc care conţine valoarea y.

Pentru fiecare bloc complet de la s până la x, valoare y apare de 2 ori. La final poate exista un ultim bloc incomplet. Acest bloc poate conţine:

- ambele apariţii ale valorii y
- doar prima apariţie
- sau niciuna

De aceea trebuie verificat separat:

- dacă am ajuns până la prima apariţie a lui y. (Dacă ultimul bloc incomplet are minim y elemente, valoare y fiind pe poziţia y)
- şi dacă am ajuns şi până la a doua apariţie. (Prin diferenţă)

Contorizăm fiecare apariţie a lui y, şi afişăm rezultatul. Complexitate: O(x).

Cod C++

```
#include <iostream>
#include <fstream>
using namespace std;
ifstream fin("sir.in");
```

```
ofstream fout("sir.out");


int main()
{long long c, x, y;
long long a, s, ct = 0;
fin >> c >> x >> y;
a = y;
if(c == 1)
     {long long i = 1;
     while(i * i < x) i++;
     if(i * i == x) fout << i;
     else fout << i - 1;
     }
else
{if(x < y * y)
     {
     fout << 0;
     return 0;
     }
s = y * (y - 1);
while(s + 2 * y <= x)
     {
     ct += 2;
     s += 2 * y;
     y++;
     }
if(x - s >= a) ct++;
if(s + 2 * y - x <= a && x - s != s + 2 * y - x) ct++;
fout << ct;
}
return 0;
}
```

VARIANTA ALTERNATIVĂ

```
#include <fstream>
#include <iostream>
using namespace std;
ifstream fin("sir.in");
ofstream fout("sir.out");
int cerinta;
long long x,y,t,gr,vmax, rsp,k;
int main()
```

```
{fin>>cerinta>>x>>y;
 if(cerinta==1)
     {   t=0;
         while(t*(t+1)<x) t++;
         t--;
         vmax=t;
         if(x-(t*(t+1))>=vmax+1) vmax++;
         fout<<vmax<<'\n';
     }
 else
    {rsp=0;
     t=1;
     while(t*(t+1)<x) { if(t>=y) rsp=rsp+2;
                         t++;
                       }
     k=t*(t-1);
     if(x-k<=t && y<= x-k) rsp++;
     if(x-k>t) {rsp++;
                 if(y>=t*(t+1)-x+1) rsp++;
               }
      fout<<rsp<<'\n';
    }
}
```

Problema PRIETENI5

Se consideră un şir de **n** numere naturale nenule.

Pentru un număr natural **x**, numim ***divizor prietenos*** al lui **x** orice divizor pozitiv **d** al lui **x** care îndeplineşte simultan următoarele condiţii:

1. suma cifrelor lui **d** este un număr natural nenul;
2. această sumă divide pe **x**.

Indicele de prietenie al lui **x** reprezintă numărul total de divizori prietenoşi ai lui **x**.

Pentru fiecare dintre cele n numere naturale date, se calculează indicele de prietenie.

Cunoscând cerinţa **c**, **n** numărul de numere şi cele **n** numere din şir, se cere:

1. Să se determine câte numere au indicele de prietenie număr impar.
2. Să se afişeze cel mai mare număr de maxim 3 cifre care poate fi format cu cifre ale indicilor de prietenie ale tuturor numerelor din şir.

Date de intrare

Fişierul de intrare **prieteni.in** conţine pe prima linie numărul natural **c** al cerinţei, care poate fi doar 1 sau 2, pe a doua linie numărul natural nenul n şi apoi **n** numere naturale nenule, separate prin câte un spaţiu.

Date de ieşire

Fişierul de ieşire **prieteni.out** va conţine, pe prima linie, valoarea determinată conform cerinţei.

Restricţii şi precizări

- 2<=n<=10^6
- Fiecare număr din şir este un număr natural nenul <= 100000.
- Se garantează că există cel puţin un număr în şir.
- Pentru cerinta 1, se acordă 32 de puncte
- Pentru cerinta 2, se acordă 68 de puncte.

Exemple

prieteni.in	**prieteni.out**	**Explicatii**
1 5 56 36 42 30 48	**4**	Şirul conţine 5 numere: 56, 36, 42, 30, 48. Indicii de prietenie sunt: 56→5, 36→9, 42→7, 30→8, 48→9. Cerinţa este 1, deci există 4 numere cu indicele de prietenie impar.
2 5 56 36 42 30 48	998	Şirul conţine 5 numere: 56, 36, 42, 30, 48. Indicii de prietenie sunt: 56→5, 36→9, 42→7, 30→8, 48→9. Cerinţa este 2, cel mai mare număr de maxim 3 cifre care se poate forma cu cifrele 5, 9, 7, 8, 9 este 998

```
#include <fstream>
#include <algorithm>
using namespace std;
ifstream cin("prieteni.in");
ofstream cout("prieteni.out");

int n, numar, ct, x, c, a, b;
short int cer;
```

```
int main()
{   cin >> cer;
    cin >> n;
    if (cer == 1)
    {
        for (int i = 1; i <= n; i++)
        {
            cin >> x;

            int ct_prietenie = 0;
            int i_div;
            for (i_div = 1; i_div * i_div < x; i_div++)
            {
                if (x % i_div == 0)
                {
                    int temp1 = i_div;
                    int s1 = 0;
                    while (temp1)
                    {
                        s1 += temp1 % 10;
                        temp1 /= 10;
                    }
                    if (x % s1 == 0)
                        ct_prietenie++;

                    int temp2 = x / i_div;
                    int s2 = 0;
                    while (temp2)
                    {
                        s2 += temp2 % 10;
                        temp2 /= 10;
                    }
                    if (x % s2 == 0)
                        ct_prietenie++;
                }
            }

            if (i_div * i_div == x)
            {
                int temp3 = i_div;
                int s3 = 0;
                while (temp3)
```

```
            {   s3 += temp3 % 10;
                temp3 /= 10;
            }
            if (x % s3 == 0)
                ct_prietenie++;
        }

        if (ct_prietenie % 2 == 1)
            ct++;
    }
    cout << ct;
}
else if (cer == 2)
{
    a = 0;
    b = 0;
    c = 0;
    for (int i = 1; i <= n; i++)
    {
        cin >> x;

        int ct_prietenie = 0;
        int i_div;
        for (i_div = 1; i_div * i_div < x; i_div++)
        {
            if (x % i_div == 0)
            {
                int temp1 = i_div;
                int s1 = 0;
                while (temp1)
                {
                    s1 += temp1 % 10;
                    temp1 /= 10;
                }
                if (x % s1 == 0)
                    ct_prietenie++;

                int temp2 = x / i_div;
                int s2 = 0;
                while (temp2)
                {
                    s2 += temp2 % 10;
                    temp2 /= 10;
```

```
            }
            if (x % s2 == 0)
                ct_prietenie++;
        }
    }

    if (i_div * i_div == x)
    {
        int temp3 = i_div;
        int s3 = 0;
        while (temp3)
        {
            s3 += temp3 % 10;
            temp3 /= 10;
        }
        if (x % s3 == 0)
            ct_prietenie++;
    }

    int valoare = ct_prietenie;
    while (valoare)
    {
        int u = valoare % 10;
        ct++;
        valoare /= 10;
        if (u >= a)
        {
            c = b;
            b = a;
            a = u;
        }
        else if (u >= b)
        {
            c = b;
            b = u;
        }
        else if (u >= c)
        {
            c = u;
        }
    }
    if (a == 9 && b == 9 && c == 9)
    {
```

```
                cout << 999;
                return 0;
            }
        }

        if (ct >= 3)
            cout << a * 100 + b * 10 + c;
        else if (ct == 2)
            cout << a * 10 + b;
        else
            cout << a;
    }

    return 0;
}
```

Editorial:

Pentru a găsi toţi divizorii lui x în mod eficient se foloseşte parcurgerea până la radical. Pentru fiecare număr, când se gaseste un divisor, ii facem suma cifrelor, daca si suma este divizior al numarul x, crestem indicele de prietenie.

Pentru cerinta 1, verificam indicele de prietenie al fiecarui numar, daca este impar, crestem un contor pe care il afisam la finalul parcurgerii.

Pentru cerinta 2 problema îşi schimbă focusul pe procesarea cifrelor tuturor gradelor de prietenie obţinute. Scopul este să determinăm **cele mai mari 3 cifre distincte sau duplicate** apărute în compoziţia acestor grade de prietenie, pe parcursul întregului şir. În loc să salveze toate cifrele într-un vector uriaş şi să le sorteze la final, codul foloseşte trei variabile simple, a, b, şi c (a>=b>=c), reprezentând cele mai bune 3 cifre găsite până în prezent.

Pentru fiecare cifră **u** extrasă din gradul de prietenie al numărului curent, se aplică o structură de tip cascadă pentru actualizare:

- **Cazul 1**: Dacă u>=a, cifra curentă devine noul maxim absolut. Valorile vechi se deplasează în jos: c = b, b = a, a = u.
- **Cazul 2**: Dacă u nu e mai mare ca a, dar u>=b, atunci u devine al doilea cel mai mare: c = b, b = u.

- **Cazul 3**: Dacă u se află doar deasupra lui c (u>=c), se actualizează doar ultima poziţie: c = u.

La finalul programului, în funcţie de câte cifre s-au reuşit a fi extrase în total (reţinute în ct), se afişează numărul format din cele mai bune 3, 2 sau singura cifră găsită.

TIP SUBIECT – PROGRAMARE

SUBIECT CLASA a VI-a – PROBA PRACTICĂ

Problema PRIETENIE6

Se consideră un şir de **n** numere naturale nenule.

Pentru un număr natural **x**, numim ***divizor prietenos*** al lui **x** orice divizor pozitiv **d** al lui **x** care îndeplineşte simultan următoarele condiţii:

1. suma cifrelor lui **d** este un număr natural nenul;
2. această sumă divide pe **x**.

Indicele de prietenie al lui **x** reprezintă numărul total de divizori prietenoşi ai lui **x**.

Pentru fiecare dintre cele n numere naturale date, se calculează indicele de prietenie.

Cunoscând cerinţa **c**, **n** şi numerele din şir, se cere:

1. Să se determine valoarea din şir cu cel mai mare indice de prietenie. Dacă există mai multe valori cu acelaşi indice de prietenie maxim, se va afişa **ultima** dintre ele.
2. Să se afişeze, în ordine crescătoare, toate numerele din şir care au indicele de prietenie egal cu indicele de prietenie maxim găsit la punctul 1.

Date de intrare

Fişierul de intrare **`prietenie.in`** conţine pe prima linie numărul natural **c** al cerinţei, care poate fi doar 1 sau 2, pe a doua linie numărul natural nenul n şi apoi **n** numere naturale nenule, separate prin câte un spaţiu.

Date de ieşire

Fişierul de ieşire **`prietenie.out`** va conţine pe o linie valorile corespunzătoare cerinţei (la cerinţa 2 numerele sunt separate prin câte un spaţiu).

Restricții și precizări

- 2<=n<=10^5
- Fiecare număr din şir este un număr natural nenul <= 100000.
- Se garantează că există cel puţin un număr în şir.
- Pentru cerinta 1, se acordă 32 de puncte
- Pentru cerinta 2, se acordă 68 de puncte.

Exemple

prieteni.in	prieteni.out	Explicatii
1 5 56 36 42 30 48	**48**	Şirul conţine 5 numere: 56, 36, 42, 30, 48. Indicii de prietenie sunt: 56→5, 36→9, 42→7, 30→8, 48→9. Cerinţa este 1, indicele de prietenie maxim este 9, ultima valoare cu indicele de prietenie 9 este 48.
2 5 56 36 42 30 48	**36 48**	Şirul conţine 5 numere: 56, 36, 42, 30, 48. Indicii de prietenie sunt: 56→5, 36→9, 42→7, 30→8, 48→9. Cerinţa este 2, indicele maxim de prietenie este 9, iar valorile care au acest indice sunt (în ordine crescătoare): 36 şi 48.

```
#include <fstream>
#include <algorithm>
using namespace std;
int c,x,y,n,vf[10],ct[100005],numbers[1000005],ctn=0;
int main()
{
    ifstream f ("prietenie.in");
    ofstream g ("prietenie.out");
    f>>c>>n;
    for (int i=1;i<=100000;i++)
    {
        int x=i,cf=0;
        while (x)
        {
            cf+=x%10;
            x/=10;
        }
        for (int j=i;j<=100000;j+=i)
            if (j%cf==0)
                ct[j]++;
    }
    int rsp1=0,mx=0;
    for (int i=1; i<=n; i++)
```

```
    {
        f>>x;
        if (ct[x]>mx)
        {
            ctn=1;
            numbers[ctn]=x;
        }
        else if (ct[x]==mx)
            numbers[++ctn]=x;
        if (ct[x]>=mx)
        {
            mx=ct[x];
            rsp1=x;
        }
    }
    if (c==1)
        g<<rsp1;
    else
    {
        sort(numbers+1,numbers+ctn+1);
        for (int i=1;i<=ctn;i++)
            g<<numbers[i]<<' ';
    }
}
```

TIP SUBIECT – PROGRAMARE

SUBIECT CLASA a VI-a, a VII-a, a VIII-a – PROBA PRACTICĂ

Problema LUMANARI

Lumânări

Într-o seară de toamnă, n lumânări sunt aşezate în linie pe un pervaz, fiecare având o înălţime exprimată printr-un număr natural. Vine un vânt puternic care bate în mai multe runde, alternând direcţia. În prima rundă, vântul bate dinspre stânga spre dreapta. O lumânare se stinge dacă înălţimea ei este strict mai mică decât înălţimea primului vecin aprins din stânga ei. Toate stingerile dintr-o rundă au loc simultan. În a doua rundă, vântul bate dinspre dreapta spre stânga. Dintre lumânările rămase aprinse, se stinge orice lumânare a cărei înălţime este strict mai mică decât înălţimea primului vecin aprins din dreapta sa. Toate stingerile au loc simultan. Micul Gates doreşte să afle:

1. câte lumânări rămân aprinse după primele două runde şi care sunt poziţiile acestora.
2. numărul de runde în care cel puţin o lumânare s-a stins, dacă vântul continuă să bată alternativ (stânga, dreapta, stânga, dreapta, ..)până când, la o rundă completă nu stinge nicio lumânare.

Date de intrare

Fişierul de intrare **lumanari.in** conţine: pe prima linie, numărul natural c (1 sau 2) — cerinţa de rezolvat, pe a doua linie, numărul natural n — numărul de lumânări, pe a treia linie, n numere naturale separate prin spaţii reprezentând înălţimile lumânărilor, în ordinea poziţiei lor de la stânga la dreapta.

Date de ieşire

Fişierul de ieşire **lumanari.out** va conţine:

- Dacă c = 1: pe prima linie numărul de lumânări rămase aprinse, apoi poziţiile lor (câte una pe linie, în ordine crescătoare)
- Dacă c = 2: o singură linie conţinând numărul de runde în care cel puţin o lumânare s-a stins

Restricţii şi precizări

- $2 \leq n \leq 10.000$

- Înălţimile lumânărilor sunt numere naturale cu valori între 1 şi 100.000
- Prima lumânare aprinsă din stânga nu are vecin aprins la stânga şi nu se poate stinge într-o rundă de stânga
- Ultima lumânare aprinsă din dreapta nu are vecin aprins la dreapta şi nu se poate stinge într-o rundă de dreapta

Exemplul 1

lumanari.in	lumanari.out	Explicatii
1 7 4 2 5 3 6 6 1	2 5 6	După runda 1 (vânt din stânga), se sting lumânările de la poziţiile 2, 4 şi 7, deoarece: ● v[2] = 2 < v[1] = 4 ● v[4] = 3 < v[3] = 5 ● v[7] = 1 < v[6] = 6 Rămân aprinse poziţiile: 1, 3, 5, 6. După runda 2 (vânt din dreapta), dintre lumânările rămase aprinse: ● lumânarea de la poziţia 3 (h=5) are primul vecin aprins din dreapta la poziţia 5 (h=6): 5 < 6, se stinge ● lumânarea de la poziţia 1 (h=4) are primul vecin aprins din dreapta la poziţia 3 (h=5): 4 < 5, se stinge ● lumânările de la poziţiile 5 şi 6 rămân aprinse

Exemplul 2:

lumanari.in	lumanari.out	Explicatii
2 7 4 2 5 3 6 6 1	2	Runda 1 (stânga): se sting lumânările de pe poziţiile: 2, 4 şi 7. Rămân: 1, 3, 5, 6. Runda 2 (dreapta): se sting lumânările de la poziţiile:1, 3. Rămân: 5, 6. Runda 3 (stânga): nu se mai stige nicio lumînare. Total runde cu stingeri: 2. Runda 3 nu a stins nicio lumânare — procesul se opreşte şi nu se numără.

Editorial: Lumânări

Ideea de Bază

O parcurgere naivă la fiecare bătaie de vânt ar duce la o complexitate ineficientă de O(N^2), aceasta totuşi va primi un punctaj generos. Soluţia optimă reduce timpul la O(N) folosind două concepte cheie:

1. Folosim 2 array-uri pentru a şti în permanenţă cine sunt vecinii activi ai fiecărei lumânări. Când o lumânare se stinge, o "ştergem" în timp O(1) actualizând legăturile vecinilor săi.
2. O lumânare trebuie verificată din nou doar dacă unul dintre vecinii ei s-a stins (pentru că a devenit vizibilă o altă lumânare, potenţial mai mare). Vom menţine elementele "suspecte" în alte 2 array-uri.

Detalii de Implementare

- Cozi simulate prin Array: Pentru a evita utilizarea unei structure de tip std::deque, folosim array-uri statice prealocate. Pentru siguranţă la adăugări/extrageri, folosim 2 indecşi care pleacă de la mijlocul vectorului.
- Evitarea duplicatelor: Folosim 2 array-uri booleene pentru a asigura că o lumânare nu este adăugată de mai multe ori în acelaşi array.
- Stingerea simultană: În timpul unei runde, găsim toate lumânările care îndeplinesc condiţia de stingere şi le stocăm temporar. La finalul rundei:
    - Le marcăm active[X] = 0.
    - Le scoatem din lanţ: vecinul din stânga primeşte un nou vecin la dreapta şi viceversa.
    - Noii vecini sunt adăugaţi în array-uri, devenind posibili candidaţi pentru rundele viitoare.
- Oprirea: Procesul se opreşte fie când trec cele 2 etape pentru prima cerinţă (c == 1), fie când nu au fost stinse lumânări în decursul unei etape.

Complexitate

- Timp: O(N). O lumânare este procesată de un număr constant de ori. Evaluările se fac doar la schimbarea vecinilor, rezultând un număr maxim de operaţii egal cu numărul de lumânări N.
- Memorie: O(N).

```
#include <iostream>
#include <fstream>
#define nmx 10005
using namespace std;
int H[nmx],stanga[nmx],dreapta[nmx],n,c;
bool active[nmx],in_dq_stanga[nmx],in_dq_dreapta[nmx];
int main()
{
    ifstream f ("lumanari.in");
    ofstream g ("lumanari.out");
    f>>c>>n;
    int dq_stanga[100005],dq_dreapta[100005];
    int st_head=50000,st_tail=50000;
    int dr_head=50000,dr_tail=50000;
    for (int i=1; i<=n; i++)
    {
        f>>H[i];
        stanga[i]=i-1;
        dreapta[i]=i+1;
        active[i]=1;

        dq_stanga[st_tail++]=i;
        in_dq_stanga[i]=1;

        dq_dreapta[dr_tail++]=i;
        in_dq_dreapta[i]=1;
    }
    stanga[1]=0;
    dreapta[n]=0;

    int round_count=0,vant=0;

    while (1)
    {
        int se_sting[nmx];
        int se_sting_sz=0;

        if (vant==0)///stanga
        {
            if (st_head==st_tail)
                break;
```

```
    while (st_head<st_tail)
    {
        int i=dq_stanga[st_head];
        st_head++;
        in_dq_stanga[i]=0;

        if (!active[i])
            continue;

        int vecin=stanga[i];
        if (vecin!=0 && H[i]<H[vecin])
            se_sting[se_sting_sz++]=i;
    }
}
else///dreapta
{
    if (dr_head==dr_tail)
        break;

    while (dr_head<dr_tail)
    {
        int i=dq_dreapta[dr_head];
        dr_head++;
        in_dq_dreapta[i]=0;

        if (!active[i])
            continue;

        int vecin=dreapta[i];
        if (vecin!=0 && H[i]<H[vecin])
            se_sting[se_sting_sz++]=i;
    }
}
if (se_sting_sz==0)
    break;

round_count++;

for (int idx=0; idx<se_sting_sz; idx++)
{
    int X=se_sting[idx];
    active[X]=0;
    int l=stanga[X],r=dreapta[X];
```

```
            if (l!=0)
            {
                dreapta[l]=r;
                if (active[l] && !in_dq_dreapta[l])
                {
                    dq_dreapta[dr_tail++]=l;
                    in_dq_dreapta[l]=1;
                }
            }
            if (r!=0)
            {
                stanga[r]=l;
                if (active[r] && !in_dq_stanga[r])
                {
                    dq_stanga[st_tail++]=r;
                    in_dq_stanga[r]=1;
                }
            }
        }

        if (c==1 && round_count==2)
            break;
        vant=1-vant;
    }

    if (c==1)
    {
        int ct=0;
        for (int i=1; i<=n; i++)
            if (active[i])
                ct++;
        g<<ct<<'\n';
        for (int i=1; i<=n; i++)
            if (active[i])
                g<<i<<'\n';
    }
    else
        g<<round_count<<'\n';
}
```

Problema PERFECTE

Se consideră un şir de **n** numere naturale nenule. Micul Gates caută, pentru fiecare număr în parte, cel mai apropiat pătrat perfect de el, înlocuind numărul cu baza pătratului perfect găsit (cea negativă dacă pătratul perfect este mai strict mic decât numărul, cea pozitivă dacă pătratul perfect este mai mare sau egal cu numărul). De exemplu, pentru numărul 45 cel mai apropiat pătrat perfect este 49=(-7)^2=7^2, deci numărul este înlocuit cu 7, iar dacă numărul este 17, pătratul perfect cel mai apropiat este 16=(-4)^2=4^2 deci numărul este înlocuit cu -4.

Se formează în acest fel un nou şir pentru care Micul Gates doreşte să afle:

1. Care este numărul cel mai mare pe care îl găseşte de cele mai multe ori în noul şir.
2. Care este cea mai scurtă secvenţă în care numerele sunt distincte cu excepţia unuia singur, care apare de două ori, afisând lungimea secvenţei găsite, poziţia de unde începe şi poziţia unde se termină aceasta. Dacă există mai multe secvenţe cu aceeaşi proprietăţi, se alege cea mai din stânga. O secvenţă poate fi alcătuită din două elemente.

Fişierul de intrare: perfecte.in conţine de prima linie numărul **c** al cerinţei, care poate fi doar 1 sau 2. Pe a doua linie se găseşte un număr natural n nenul iar pe a treia linie n numere naturale nenule separate printr-un spaţiu, cu semnificaţia din enunţ.

Fişierul de ieşire: perfecte.out.va conţine, pe prima linie, o valoarea dacă cerinţa este 1, sau 3 valori separate prin câte un spaţiu dacă cerinţa este 2.

Restrictii si precizari:

1<=n <=1000000

Numerele din şir au valorile maxim 1000000.

Se garantează că pentru fiecare test există soluţie.

Pentru cerinta 1, se acordă 32 de puncte.

Pentru cerinta 2, se acordă 68 de puncte.

Exemplu:

perfecte.in	**perfecte.out**	**Explicatii**
1 **4** **45 20 25 23**	**5**	Cerinta este 1, n=4, pentru fiecare număr din şir se găseşte baza conform cerinţei şi şirul devine: 7, -4, 5, 5, deci numărul care apare de cele mai multe ori în noul şir este 5.

2 **6** **25 20 45 23 19** **31**	**4 1 4**	Cerinta este 2, n=6, pentru fiecare număr din şir se găseşte baza conform cerinţei şi şirul devine: 5, -4, 7, 5, -4, 6 deci secvenţa cerută este formată din 5, -4, 7, 5, are lungimea 4 şi are poziţiile de început şi sfârşit 1 şi 4.

```
#include <iostream>
#include <fstream>
#include <cmath>
#define nmx 1000005
#define inf 1e9
using namespace std;
int n,c,x,v[nmx],vf[2005],last[2005];
int main()
{
    ifstream f ("perfecte.in");
    ofstream g ("perfecte.out");
    f>>c>>n;
    for (int i=1; i<=n; i++)
    {
        f>>x;
        int a=int(sqrt(x));
        int b=a+1;
        if (a*a==x)
            v[i]=a;
        else if (x-a*a<b*b-x)
            v[i]=-a;
        else
            v[i]=b;
    }
    if (c==1)
    {
        int mx_frec=0,rsp=-inf,cnst=1000;
        for (int i=1; i<=n; i++)
        {
            int idx=cnst+v[i];
            vf[idx]++;
            if (vf[idx]>mx_frec)
            {
                mx_frec=vf[idx];
```

```
                rsp=v[i];
            }
            else if (vf[idx]==mx_frec && v[i]>rsp)
            {
                rsp=v[i];
            }
        }
        g<<rsp;
    }
    else
    {
        int cnst=1000,leng=inf,pzst=0,pzend=0;
        ///folosim o constanta pentru a evita apelarea unui array
la un index negativ
        for (int i=1; i<=n; i++)
        {
            int idx=cnst+v[i];
            if (last[idx]!=0 && i-last[idx]+1<leng)
            {
                leng=i-last[idx]+1;
                pzst=last[idx];
                pzend=i;
            }
            last[idx]=i;
        }
        g<<leng<<' '<<pzst<<' '<<pzend;
    }
}
```

Editorial:

Pentru un număr x, rădăcina sa pătrată calculată prin sqrt(x) și trunchiată la întreg (să o numim A) ne oferă cel mai mare pătrat perfect mai mic sau egal cu x, adică A^2 <=X.

Următorul pătrat perfect imediat mai mare este B^2, unde B = A + 1.

Numărul x se va afla mereu în intervalul A^2, B^2. Pentru a vedea de cine este mai aproape, comparăm distanțele:

- Distanța până la pătratul mai mic: x – A^2
- Distanța până la pătratul mai mare: B^2 – x.

- Deoarece valorile din vector pot fi negative (ajungând de exemplu până la -1000), nu le putem folosi direct ca indici într-un vector de frecvenţă clasic (vf[-3] determină o eroare de execuţie).
- **Soluţia:** Translatarea axei numerelor cu o constantă (cnst = 1000).
- Valoarea -1000$devine indexul 0.
- Valoarea -3 devine indexul 997.
- Valoarea 3 devine indexul 1003.
- Parcurgem elementele, mărim frecvenţa în vectorul translatat (vf[cnst + v[i]]++) şi actualizăm maximul în timp real, respectând şi condiţia de departajare (valoarea maximă la frecvenţe egale).

Cerinţa 2 se rezolvă astfel: Ni se cere să găsim o subsecvenţă de lungime minimă care începe şi se termină cu aceeaşi valoare procesată. Cu alte cuvinte, trebuie să găsim două apariţii ale aceluiaşi număr în vectorul V care sunt cât mai aproape una de cealaltă.

Tehnica: Vectorul de „ultima poziţie”

În loc să folosim o metodă brută de tip O(N^2) în care comparăm fiecare element cu toate celelalte, putem rezolva problema în O(N) folosind un vector numit last.

- last[idx] va reţine poziţia (indexul din şir) unde a fost văzută ultima dată valoarea asociată indexului idx (folosind aceeaşi translatare cu cnst = 1000).

Algoritmul pe paşi:

1. Iniţializăm distanţa minimă leng cu o valoare infinită (inf = 1e9).
2. Parcurgem şirul cu indicele i de la 1 la N.
3. Pentru elementul current V[i], verificăm dacă a mai apărut în trecut (last[idx] != 0).
4. Dacă a mai apărut, distanţa dintre apariţia curentă şi cea precedentă este:

   i - last[idx] + 1.
5. Dacă această distanţă este mai mică decât leng-ul actual, actualizăm distanţa minimă şi salvăm capetele (pzst = last[idx] şi pzend = i).
6. **Important:** Indiferent dacă a mai apărut sau nu, actualizăm last[idx] = i, deoarece pentru următoarele apariţii, poziţia curentă i va fi cea mai apropiată din stânga.