1. Experiment fizică

La laboratorul de fizică, efectuaţi următorul experiment cu colegii voştri: se încălzeşte un lichid şi cu ajutorul lui termometru se măsoară temperatura lichidului din *t* în *t* secunde (un singur elev citește temperatura la momentul *t* şi o notează în caiet). Ştiind că lichidul se încălzeşte urmând o funcţie strict crescătoare, determinaţi momentul de timp *t* în care lichidul a atins pentru prima dată o temperatură pozitivă.

Fiind dată o funcţie strict crescătoare ***f*****,** adică ***f(t + 1) > f(t),*** ∀ **x** ∈ℕ, determinați valoarea lui **t** pentru care **f** devine pozitivă pentru prima dată.

Rezolvare

Metoda 1

Cea mai simplă metodă de rezolvare ar fi să calculăm, în mod secvenţial, valoare funcţiei ***f*** pentru toate valorile lui ***t*** şi să ne oprim la prima valoare a lui *t* pentru care f(t) este mai mare sau egală cu 0.

```
int findFirstPositiveM1() {
    for (unsigned int t = 0; t < std::numeric_limits<unsigned int>::max(); t++)
        if (f1(t) >= 0)
            return t;
    return -1;
}
```

Complexitatea acestei metode este O(ℕ).

Metoda 2

O metodă mai eficientă ar fi de a aplica căutarea binară (precondiţia este satisfăcută pentru că funcţia este strict crescătoare); dar pentru căutarea binară, avem nevoie de un interval marginit (poziţia de start şi de final a sub intervalului în care facem căutarea).

Începem rezolvarea prin a determina o valoare pentru *end* pentru care funcţia este pozitivă

- Începem cu *end* de la 0, și cât timp *f(end)* este mai mic decât zero, dublăm valoarea lui *end.*
- Odată ce *f(end)* este mai mare decât zero, atunci luăm *end* ca valoare de final a intervalului pentru căutarea binară şi *end/2* ca valoare de început.
- Apoi aplicăm algoritmul de căutare binară pentru a găsi prima valoare pozitivă a lui f(*end*) între *end/2* şi *end.*

```
int binarySearch(int start, int end)
{
    if (end >= start)
    {
        int mid = start + (end - start) / 2;
        if (f(mid) >= 0 && (mid == start || f(mid - 1) < 0))
            return mid;
        if (f(mid) <= 0)
            return binarySearch((mid + 1), end);
        else
            return binarySearch(start, (mid - 1));
    }
    return -1;
}


int findFirstPositiveM2()
{
    if (f(0) > 0)
        return 0;

    unsigned int end = 1;
    while (f(end) <= 0)
        end = end * 2;

    return binarySearch(end / 2, end);
}
```

Complexitatea acestei metode este O(log N).
Numărul de paşi pentru a găsi limita superioară *end* a intervalului de căutare este O(log N).

Valoarea lui *end* este cel mult 2*N. Numărul de elemente dintre end/2 şi end este ~N. Complexitatea căutării binare este O(log N), iar complexitatea totală a timpului este 2*O(log N), ceea ce înseamnă O(log N).

2. Numere concurs

Alice organizează un concurs de dans (în perechi) şi a tipărit etichetele cu numărul de concurs (valoare între 1 şi 10000) pentru fiecare dintre cei *n* participanţi. Membrii unei echipe au acelaşi număr de concurs. Totuşi, înainte de concurs, Alice observă că îi lipseşte o etichetă (are doar *n-1* etichete).
Fiind dat un tablou unidimensional care conţine numerele de pe etichetele listate pentru concursul de dans (toate elementele apar de două ori, cu excepţia unui element care apare o singură dată), scrieţi un program C++ care determină elementul care nu are pereche.

Metoda 1

Cea mai simplă şi ineficientă metodă ar fi să parcurgem tabloul şi pentru fiecare număr să calculăm de câte ori apare în tablou.

```
int findElementNotInPairM1(int arr[], int n) {
    int count = 0;
    for (int i = 0; i < n; i++) {
        count = 0;
        for (int j = 0; j < n; j++)
            if (arr[i] == arr[j]) {
                count++;
            }
        if (count != 2)
            return arr[i];
    }
    return -1;
}
```

Complexitatea acestei metode este $O(N^2)$.

Metoda 2

O metodă puţin mai eficientă ar fi să sortăm vectorul, apoi să-l parcurgem şi să determinăm elementul care nu apare de două ori.

```
int findElementNotInPairM2(int arr[], int n) {
   // sortam tabloul
    sort(arr, arr + n);
    // parcurgem tabloul pentru a determina care este elementul ca nu apare de 2 ori
```

```
    for (int i = 0; i < n - 1; i+=2) {
        if (arr[i] != arr[i + 1])
            return arr[i];
    }
    return arr[n - 1];
}
```

Complexitatea acestei metode este O(N logN). Cel mai costisitor pas este dat de sortare cu complexitate O(NlogN).

Metoda 3

O altă metodă ar fi să calculăm vectorul de frecvență pentru tablou, și pe baza acestuia să determinăm elementul care apare o singură dată.

```
int findElementNotInPairM3(int arr[], int n) {
    int* freq = new int[10000](); // () - initializeaza vectorul de frecventa cu 0
    for (int i = 0; i < n; i++) {
        if(arr[i] >=0 && arr[i]< 10000) // ne asiguram indexul pe care vrem sa-l accesam
este valid
            freq[arr[i]]++;
    }
    // parcurgem vectorul de frecventa si determinam numarul care apare o singura data
    for (int i = 0; i < 10000; i++)
        if (freq[i] == 1)
            return i;
    // eliberam memoria alocata dinamic
    delete[] freq;
    return -1;
}
```

Complexitatea acestei metode este O(N), dar folosim și un tablou auxiliar, deci și complexitatea spațiu este O(M).

Metoda 4

Operaţia XOR (^), sau "sau exclusiv", returnează 1 în fiecare poziţie de biţi unde biţii corespunzători ai celor doi operanzi sunt diferiţi.

x	y	x^y
0	0	0
0	1	1
1	0	1

1	1	0

Dacă aplicăm această operaţie asupra a două numere care sunt egale, rezultatul va fi întotdeauna 0.
În plus observăm că x^0 = x, ∀ x ∈ {0, 1}, pentru că:

- x =0: 0^0 = 0;
- x = 1: 1^0 = 1.

De asemenea, operaţia XOR este asociativă şi comutativă, ceea ce înseamnă că ordinea în care se fac operaţiile nu contează.

Să luăm ca exemplu operaţia 13 ^ 13. 13 reprezentat în binar este: 1101 ($2^3+2^2+2^0 = 8+4+1 = 13$).

1	1	0	1	^
1	1	0	1	
0	0	0	0	

Rezultatul este deci 0.

În schimb, dacă efectuăm operaţia 13 ^ 5. 13 reprezentat în binar este: 1101 ($2^3+2^2+2^0 = 8+4+1 = 13$), iar 5 în binar este 0101 ($2^2+2^0 = 4+1 = 5$).

1	1	0	1	^
0	1	0	1	
1	0	0	0	

Rezultatul este diferit de 0 (8).

În cazul nostru, toate numerele din array-ul dat apar de două ori, cu excepţia unui singur număr care apare o singură dată. Dacă am aplica operaţia XOR pe toate numerele, numerele care apar de două ori ar fi eliminate, rămânând doar numărul care apare o singură dată. Acest lucru se întâmplă deoarece, atunci când două numere identice sunt "XOR-ate", toţi biţii corespunzători sunt identici, iar rezultatul operaţiei este 0.

De exemplu, dacă avem tabloul
{**4**, **5**, **2**, **2**, **16**, **5**, **4**}
4^4 = 0
5^5 = 0
2^2 = 0

iar 16^0^0^0 = 16.

```
int findElementNotInPairM4(int arr[], int n) {
    int el = 0;
    for (int i = 0; i < n; i++)
        el = el ^ arr[i];
    return el;
}
```

Complexitatea acestei metode este O(N).

3. Tranzacționare de acțiuni

Se dă un tablou unidimensional *p*, în care fiecare element *p[i]* reprezintă preţul acţiunii în ziua *i*. Determinaţi profitul maxim pe care îl puteţi obţine alegând o zi în care să cumpăraţi o acţiune şi o altă zi (ulterioară) în care să vindeţi această acţiune.

Rezolvare

În primul rând convertim tabloul de preţuri ale acţiunii într-un tablou de diferenţe de preţ între zile succesive, deoarece profitul maxim care poate fi obţinut prin cumpărarea şi vânzarea unei acţiuni la momente diferite este echivalent cu găsirea sumei maxime a oricărui subsecvenţe contigue din tabloul de diferenţe de preţ.

Dacă luăm în considerare oricare doi indici *i* și *j* astfel încât *i* < *j*, atunci profitul care poate fi obţinut prin cumpărarea acţiunilor la indicele *i* și vânzarea lor la indicele j este dat de p[j] - p[i]. Dorim să găsim valoarea maximă a acestei expresii pentru toate valorile posibile ale lui *i* și *j*.

Să presupunem acum că creăm un tablou d de diferenţe de preţ între zile succesive sub forma *d[i] = p[i] - p[i-1]* pentru *i* în intervalul *[1, n-1]*. Reţineţi că *d* are cu un element mai puțin decât *p*. Valoarea lui *d[i]* reprezintă diferenţa dintre preţul acţiunilor din ziua *i* şi preţul acţiunilor din ziua precedentă *i-1*. Dacă d[i] este pozitiv, înseamnă că preţul acţiunilor a crescut faţă de ziua precedentă, iar dacă *d[i]* este negativ, înseamnă că preţul acţiunilor a scăzut faţă de ziua precedentă.

Acum, dacă luăm în considerare oricare doi indici *i* și *j*, astfel încât *i* < *j*, atunci profitul care poate fi obţinut prin cumpărarea acţiunilor la indicele i şi vânzarea lor la indicele j este dat de suma elementelor din intervalul *[i, j-1]* al tabloului *d*. Acest lucru poate fi văzut prin extinderea expresiei: *p[j] - p[i]* ca *(p[j] - p[j-1]) + (p[j-1] - p[j-2]) + ... + (p[i+1] - p[i]) + (p[i] - p[i-1])* şi observând că fiecare termen din expansiune este egal cu un element al tabloului *d*.

Prin urmare, găsirea profitului maxim care poate fi obţinut prin cumpărarea şi vânzarea acţiunilor la momente diferite este echivalentă cu **găsirea sumei maxime a oricărei subsecvenţe contigue din tabloul diferenţelor** de preț între zile succesive.

Metoda 1. Metoda cea mai simplă de rezolvare ar fi să considerăm toate perechile posibile pentru începutul şi finalul subsecvenţei cu suma maximă, şi să calcu

Cod c++:

```
int maximumSubarraySumV1(int arr[], int n) {
    int maxSum = std::numeric_limits<int>::min();
    int currSum = 0;
    for (int i = 0; i < n - 1; i++) {
        currSum = arr[i];
        for (int j = i + 1; j < n; j++) {
            currSum += arr[j];
            if (currSum > maxSum) {
                maxSum = currSum;
            }
        }
    }
    return maxSum;
}
```

Complexitatea acestei metode este $O(N^2)$

Metoda 2.

Algoritmul lui Kadane este un algoritm eficient pentru a găsi suma celui mai mare subşir comun (maximum subarray problem) într-un tablou unidimensional de numere întregi. La fiecare pas, algoritmul adaugă un element la subşirul curent şi decide dacă subşirul cu elementul adăugat este mai bun decât subşirul de sumă maximă curent sau nu. Dacă subşirul cu elementul adăugat are o sumă mai mare decât subşirul maxim curent, atunci se actualizează subşirul de sumă maximă curent.

Algoritmul menţine două variabile importante: *maxSoFar* şi *maxEndingHere*. Variabila *maxSoFar* reprezintă valoarea sumei subşirului cu suma maximă întâlnită până în prezent în tabloul dat, inclusiv subşirul curent. Iniţial, această valoare este 0. Variabila *maxEndingHere* reprezintă valoarea maximă a subşirului care se termină cu elementul curent. Adică, valoarea maximă a sumei tuturor subşirurilor care includ elementul curent şi se termină cu acesta. Iniţial, această valoare este 0.

În timpul parcurgerii array-ului, valoarea lui *maxEndingHere* este actualizată la fiecare pas prin adăugarea elementului curent. Dacă *maxEndingHere* devine mai mare decât *maxSoFar* , *maxSoFar* este actualizat la valoarea *maxEndingHere*. Acest lucru se întâmplă pentru că *maxEndingHere* reprezintă valoarea maximă a subşirului care se termină cu elementul curent, iar *maxSoFar* reprezintă valoarea maximă a subşirului care se termină cu unul dintre elementele precedente. Deci, dacă *maxEndingHere* este mai mare decât *maxSoFar*, înseamnă că subşirul care se termină cu elementul curent este mai bun decât oricare alt subşir care se termină cu un element precedent.

```
int maximumSubarraySumV2(int arr[], int n) {
    int maxSoFar = 0; //  valoarea sumei subsirului cu suma maxima intalnita pana in prezent
in tabloul dat
    int maxEndingHere = 0; // suma maxima a subsirului care se termina cu elementul curent

    for (int i = 0; i < n; i++) {
        maxEndingHere += arr[i]; // adaugam elementul curent in subsirul curent

        if (maxEndingHere > maxSoFar) {
            maxSoFar = maxEndingHere; // actualizam profitul maxim de pana acum
        }

        if (maxEndingHere < 0) {
            maxEndingHere = 0; // resetam profitul maxim daca acesta devine negativ
        }
    }
    return maxSoFar;
}
```

De exemplu, dacă avem tabloul { 2, -4, 6, -1, 5, 3, -12, -6}:

i = 0; maxEndingHere = 2; maxSoFar = 2
i = 1; maxEndingHere = -2; maxSoFar = 2
maxEndingHere = 0
i = 2; maxEndingHere = 6; maxSoFar = 6
i = 3; maxEndingHere = 5; maxSoFar = 6
i = 4; maxEndingHere = 10; maxSoFar = 10
i = 5; maxEndingHere = 13; maxSoFar = 13
i = 6; maxEndingHere = 1; maxSoFar = 13
i = 7; maxEndingHere = -5; maxSoFar = 13
maxEndingHere = 0

Rezultatul va fi 13.

Astfel, algoritmul lui Kadane găseşte suma celui mai mare subşir comun într-un array unidimensional de numere întregi într-un mod eficient şi elegant. Algoritmul are o complexitate temporală de O(N), unde N este lungimea array-ului, ceea ce îl face foarte eficient în comparaţie cu algoritmul prezentat anterior.

Revenind la problema propusă:

```
int computeMaxProfit(int p[], int n) {

    int diff[MAX_SZ];
    // calculam diferentele dintre preturile actiunilor
    for (int i = 1; i < n; i++) {
        diff[i-1] = p[i] - p[i - 1];
    }

    // atentie, tabloul cu diferente are n-1 elemente
    int  maxProfit = maximumSubarraySumV1(diff, n - 1);
    return maxProfit > 0 ? maxProfit : 0;
```

```
}
```

De exemplu, dacă avem următorul tablou unidimensional cu preţuri:

10	12	8	14	13	18	21	9	3

Prima dată calculăm diferenţele dintre preţurile acţiunii în zile consecutive:

2	-4	6	-1	5	3	-12	-6

iar profitul maxim pe care îl putem obține este 13.