Tablouri unidimensionale (vectori)

Pentru a crea și o motivație a învățării structurii pe care o vom prezenta, vom porni cu un enunț de problemă:

Se cere de la tastatură un număr `n`, apoi `n` numere naturale. Se consideră `1 ≤ n ≤ 1000` iar valorile de maxim `9` cifre. Avem două cerințe:

a) Determinați cea mai mare valoare dintre cele `n`;
b) Determinați câte dintre valorile citite sunt egale cu ultima introdusă;

Pentru cerința a) rezolvarea este cunoscută (și recomandată) folosind câteva variabile simple, chiar în momentul citirii datelor:

```
#include <iostream>
using namespace std;
int n, maxim, x, i;
int main () {
    cin>>n;
    maxim = -1;
    for (i=1;i<=n;i++) {
        cin>>x;
        if (x > maxim)
            maxim = x;
    }
    cout<<maxim;
}
```

Aici observăm că toate numerele au fost citite (și memorate) în aceeași variabilă și imediat după citire am folosit valoarea comparând-o cu maximul.

Pentru rezolvarea celei de-a doua cerințe observăm că în momentul citirii ultimei valori este necesar să cunoaștem valoarea celorlalte, citite înainte. Deci avem nevoie, în acest caz, ca la final să avem memorate toate numerele introduse. Cu modul de folosire a variabilelor utilizat până acum nu am putea rezolva. Ar apărea întrebări de genul: cum declar până la `1000` de variabie? Cum le denumesc? etc.

Limbajele de programare oferă soluții pentru astfel de probleme prin folosirea tablourilor de memorie.

Un tablou de memorie este format din mai multe variabile, toate de același tip, care pot fi accesate folosind numele tabloului și un număr (sau, vom vedea ulterior, mai multe numere) ce reprezintă poziția în tablou.

Exemplu:

`int x;`	O singură variabilă de tip int.
`int v[100]`	100 de variabile de tip int.

Din exemplu deducem și modul general de a declara tablouri:

`tip nume[dimensiune];`

- *tip* reprezintă tipul comun al tuturor elementelor tabloului; poate fi folosit orice tip al limbajului;
- *nume* reprezintă identificatorul cu care ne referim la variabila compusă `v`;
- *dimensiune* trebuie să fie o expresie constantă (adică poate conține constante și eventual operații cu ele); această valoare reprezintă numărul de elemente (de variabile de tipul *tip*) ale tabloului;

Componentele tabloului se accesează în program prin construcții de forma `nume[poziție]`, unde poziție reprezintă o expresie de tip întreg, nu neapărat constantă, și a cărei valoare să fie, în momentul accesării, cuprinsă între `0` și `dimenesiune-1`.

Așadar, pozițiile (indicii) elementelor unui tablou sunt numere cu valori cuprinse între `0` și `dimensiune-1`.

Vom vedea că putem declara și variabile compuse cu mai multe dimensiuni (la care, evident, vom folosi la accesare mai mulți indici). Cele prezentate acum, mai sus, se numesc **tablouri unidimensionale** sau li se mai spune **vectori**.

Componentele unui tablou sunt variabile obișnuite de tipul comun și pot fi folosite în orice loc este permisă o variabilă de tipul respectiv.

Fie declarațiile:
`int v[100];`
`int i, x;`	

`cin>>v[3];`	Corect, ce cere de la tastatură o valoare și se memorează în cea de-a patra componentă a tabloului (numerotarea începe de la `0`).
`v[2] = 2*v[3];`	Este o atribuire corectă în care intervin două componente ale tabloului. Cea din stânga este variabila a cărei valoare se va modifica.

`i=2;`
`cin>>x;`
`v[i] += 10*x;`
	Aici intervin trei variabile de tip `int`. Observăm că putem avea o componentă a tabloului în stânga unei atribuiri (în ea se poate memora ceva ca în orice altă variabilă de tip `int`).

Pentru o mai bună înțelegere a ceea ce reprezintă un vector, ne putem gândi că o declarație `int v[2]` este totuna ca și cum am fi declarat `int a,b;`

Elementele unui vector sunt stocate în memorie într-o zonă continuă iar numele vectorului este de fapt o variabilă care reține adresa de început a acestei zone. Când accesăm un element printr-o scriere de forma `v[expresie]`, se calculează `expresie * sizeof(tip)` octeți de la adresa de început și se obține exact adresa unde se află octeții componentei care ne interesează. Așadar, accesul la o conponentă a unui vector este unul direct cu cost de timp neglijabil. Mai multe despre reprezentarea internă a tablourilor vom studia la capitolul dedicat pointerilor.

Acum este momentul să vedem cum folosim efectiv un vector într-un program. Mai concret, vom rezolva punctul b) al problemei enunțate.

Vom urma pașii:

- declarăm un vector care să poată memora cel puțin `1000` de elemente de tip `int`;
- citim de la tastatură pe `n` și cele `n` numere și, foarte important, le memorăm în vector;
- parcurgem iarăși vectorul pentru a compara elementele sale cu ultimul citit, aflând astfel rezultatul.

```
#include <iostream>
using namespace std;
int n, i, cnt;
int v[1001];
int main () {
    cin>>n;
    for (i=1;i<=n;i++) {
        cin>>v[i];
    }
    cnt = 0;
    for (i=1;i<=n;i++)
```

```
        if (v[n] == v[i])
            cnt++;
    cout<<cnt;
}
```

Observații

- structura potrivită pentru a traversa element cu element vectorul este `for`;
- am decis să folosim componentele aflate pe pozițiile `1, 2, ..., n`; astfel componenta de pe poziția `0` rămâne nefolosită; tot așa rămân nefolosite și componentele de pe pozițiile `n+1, n+2 ... 1000` (am declarat tabloul cu dimensiunea `1001`, deci componentele sale disponibile sunt pe pozițiile `0, 1, ..., 1000`).
- Faptul că folosim componente de pe pozițiile `1..n` (și nu `0..n-1`) este pentru a obișnui mai rapid pe programatorii începători cu folosirea tablourilor.
- Avem pentru un tablou o dimensiune fizică (`1001`, cea declarată) și una logică (`n` – numărul de componente folosite efectiv);

Să analizăm programul de mai sus. Prima structură repetitivă face ca indicele i să ia pe rând valori ale fiecărei poziții din `v` la care se află elemente unde dorim să memorăm valorile introduse. Secvența este echivalentă cu:

`cin>>v[1]; cin>>v[2]; cin>>v[3]; … cin>>v[n];` (iar noi fără să folosim o repetiție nu putem obține acest efect de *puncte puncte*).

După terminarea acestei repetiții numerele introduse de noi se află toate în memorie, fiecare în câte o componentă a tabloului, deci putem să le folosim în alte scopuri.

Traversăm acum iarăși elementele tabloului, testând prin comparație care sunt egale ca valoare cu cel stocat pe ultima poziție.

Putem concluziona: în general, elementele unui vector, aflate înre pozițiile `p1` și `p2` (cu `p1 ≤ p2`) se pot vizita cu o secvență de forma:

```
for (indice = p1; indice <= p2; indice++)
      accesează elementul v[indice];
```

Probleme rezolvate

1. Se cere de la tastatură un număr `n` (de maxim `4` cifre) și apoi `n` numere întregi cu maxim `9` cifre. Să se determine:

a) Memorarea numerelor citite într-un vector pe poziții de la `1` la `n`;
b) Suma valorilor pare (dacă nu sunt valori pare se va afișa `0`);
c) Suma valorilor de pe poziții pare;
d) Numărul valorilor nule;
e) Maximul elementelor negative (decă nu sunt valori negative se va afișa `0`);
f) Media aritmetică a valorilor de două cifre (dacă nu sunt astfel de valori se va afișa `0`);

Rezolvare

Problema este un exercițiu bun pentru aplicarea unor operații de bază asupra unui șir de numere care are elementele stocate în memorie. Fiecare cerință poate fi rezolvată fără a fi necesară folosirea unu vector, dar noi acum o vom face după memorarea elementelor, fiind vorba, pe de o parte de o aplicație didactică, pe de altă parte cerințele pot face parte dintr-o aplicație mai mare care cere memorarea elementelor.

Toate cerințele necesită parcurgerea clasică a unui tablou unidimensional, de exemplu cu un `for`. Am ales varianta de a rezolva fiecare cerință individual, cu o structură repetitivă separată, dar știm că se putea folosi o singură repetiție pentru toate (chiar în momentul citirii, mai ales că am observat că nu era nevoie de stocatea elementelor).

```
#include <iostream>
using namespace std;
int n, i, sumaPare, sumaPozitiiPare, z;
int maxim, s, nr;
int v[10001];
/**
am declarat tabloul pentru a incapea
numarul maxim posibil de componente
conform enuntului
**/
int main () {
    cin>>n;
    /// citirea datelor intr-un vector
    for (i=1;i<=n;i++) {
        cin>>v[i];
    }

    /// suma valorilor pare
    sumaPare = 0;
    for (i=1;i<=n;i++)
        if (v[i] % 2 == 0)
            sumaPare+=v[i];
    cout<<sumaPare<<"\n";

    /// suma valorilor de pe pozitii pare
    sumaPozitiiPare = 0;
    for (i=2;i<=n;i+=2)
        sumaPozitiiPare += v[i];
    cout<<sumaPozitiiPare<<"\n";
    /** sau
    for (i=1;i<=n;i++)
        if (v[i] % 2 == 0)
            sumaPozitiiPare += v[i];

    **/

    /// numarul valorilor nule
    z = 0;
    for (i=1;i<=n;i++)
        if (v[i] == 0)
            z++;
    cout<<z<<"\n";
    ///maximul elementelor negative
    maxim = -1000000000;
    for (i=1;i<=n;i++)
        if (v[i] < 0 && v[i] > maxim)
            maxim = v[i];
```

```
    if (maxim == -1000000000)
        cout<<0<<"\n";
    else
        cout<<maxim<<"\n";

    ///media aritmetica a valorilor de doua cifre
    s = 0;
    nr = 0;
    for(i=1;i<=n;i++)
        if (v[i] > 9 && v[i] < 100) {
            s += v[i];
            nr ++;
        }
    if (nr == 0)
        cout<<0;
    else
        cout<<s*1.0/nr;
}
```

2. Se citesc elementele unui vector despre care se știe că se dau distincte. Afișați-le pe cele aflate între valoarea maximă și valoarea minimă (pbinfo.ro, #490).

Rezolvare

Determinăm pozițiile pentru maxim și minim și parcurgem elementele dintre cele două poziții, afișându-le. Trebuie analizat cazul când avem maximul înaintea minimului și invers.

```
#include<iostream>
using namespace std;
int v[1001];
int n, i, maxim, minim, pmaxim, pminim, aux;
int main () {
    cin>>n;
    for (i=1;i<=n;i++)
        cin>>v[i];
    minim = v[1];
    pminim = 1;
    maxim = v[1];
    pmaxim = 1;
    for (i=2;i<=n;i++) {
        if (v[i] < minim) {
            minim = v[i];
            pminim = i;
        }
        if (v[i] > maxim) {
            maxim = v[i];
            pmaxim = i;
        }
    }
    if (pmaxim < pminim) {
        aux = pmaxim;
        pmaxim = pminim;
        pminim = aux;
```

```
    }
    for (i=pminim; i<=pmaxim; i++)
        cout<<v[i]<<" ";
    return 0;
}
```

3. Se memorează într-un vector n valori naturale citite de la tastatură. Să se determine câte perechi egal depărtate de extremități sunt prime între ele (pbinfo.ro, #492).

Repetiția folosită are `n/2` pași întrucât la fiecare dintre ei se accesează două elemente ale vectorului. Observăm relația dintre elementul de pe poziția `i` din vector și cel de pe aceeași poziție dacă am număra de la cealaltă extremitate (aceasta este `n+1-i`).

```
#include <iostream>
using namespace std;
int v[201];
int a, b, r, sol, i, n;
int main () {
   cin>>n;
   for (i=1;i<=n;i++)
       cin>>v[i];
   for (i=1;i<=n/2;i++) {
       /// cmmdc dintre v[st] si v[dr]
       int a = v[i];
       int b = v[n-i+1];
       while (b!=0){
           r = a%b;
           a = b;
           b = r;
       }
       if (a == 1)
           sol++;
   }
   cout<<sol;
   return 0;
}
```

4. Se dă un vector `x` cu `n` elemente, numere naturale. Să se construiască un alt vector `y`, cu `n` elemente, cu proprietatea că `y[i]` este egal cu suma elementelor din `x`, cu excepția lui `x[i]`. (pbinfo.ro, #494).

Rezolvare

Observăm că putem calcula suma tuturor elementelor o singură dată (aici am făcut asta la citire) și atunci o vom folosi la fiecare pas obținând: `y[i] = suma - x[i]`.

```
#include <iostream>
using namespace std;
int x[201], y[201];
int n, i, suma;
int main () {
    cin>>n;
    for (i=1;i<=n;i++) {
        cin>>x[i];
        suma += x[i];
    }
    for (i=1;i<=n;i++)
        y[i] = suma - x[i];
```

```
    for (i=1;i<=n;i++)
        cout<<y[i]<<" ";
}
```

5. Se dă un vector `x` cu `n` elemente, numere naturale. Să se construiască un alt vector `y`, care să conțină elementele prime din `x`, în ordine inversă (pbinfo.ro, #495).

Rezolvare

În acest caz elementul `y[i]` nu se mai calculează pe baza celui de pe aceeași poziție din `x` și în plus numărul de elemente din `y` poate să difere de `n`. Se procedează astfel: folosim o variabilă `k` care va indica faptul că de la poziția `1` la poziția `k` s-au memorat deja elemente în `y`. Inițial `k` este `0` și de câte ori trebuie să adăugăm un element `e` la `y` avem: `k++; y[k] = e;`

```
#include <iostream>
using namespace std;
int x[201], y[201], n, k, i, j;
int main () {
    cin>>n;
    for (i=1;i<=n;i++)
        cin>>x[i];
    k = 0; /// cate elemente am pus in w
    for (i=n;i>=1;i--) {
        /// verific daca este prim v[i]

        int divizori = 0;
        for (j=2;j<=x[i]/j;j++)
            if (x[i] % j == 0) {
                divizori++;
                break;
            }
        if (x[i] > 1 && divizori == 0) {
            k++;
            y[k] = x[i];
        }
    }
    for (i=1;i<=k;i++)
        cout<<y[i]<<" ";
    return 0;
}
```

6. Se dă un șir cu `n` elemente (`n` este maxim `1000` și valorile de maxim `9` cifre). Să se determine câte elemente apar o singură dată.

Rezolvare

La această problemă este nevoie de două structuri repetitive, una în alta. Cu primul `for` fixăm elementul pe care îl testăm dacă este unic iar cu al doilea for parcurgem restul elementelor pentru a le compara cu cel fixat.

```
#include <iostream>
using namespace std;
int v[201];
int n, i, j, nr, sol;
int main () {
    cin>>n;
    for (i=1;i<=n;i++) {
        cin>>v[i];
    }
    for (i=1;i<=n;i++) {
```

```
        nr = 0;
        for (j=1;j<=n;j++)
            if (v[i] == v[j])
                nr++;
        if (nr == 1)
            sol++;
    }
    cout<<sol;
    return 0;
}
```

7. Se dă un șir cu `n` elemente (`n` este maxim `1000` și valorile de maxim `9` cifre). Să se determine câte elemente apar o singură dată. În plus, se știe că elementele sunt ordonate crescător.

Rezolvare

Enunțul este foarte asemănător cu cel de la problema anterioară, doar că elementele sunt ordonate. Această informație schimbă semnificativ modul de abordare, nefiind nevoie de două foruri (care înseamnă timp de executare de ordin n^2), un for fiind suficient, comparând doar elemente de pe poziții alăturate.

```
#include <iostream>
using namespace std;
int v[201];
int n, i, sol;
int main () {
    cin>>n;
    for (i=1;i<=n;i++) {
        cin>>v[i];
    }
    for (i=2;i<n;i++) {
        if (v[i] != v[i-1] && v[i] != v[i+1])
            sol++;
    }
    if (v[1] != v[2])
        sol++;
    if (v[n] != v[n-1])
        sol++;
    cout<<sol;
    return 0;
}
```

8. Se citește de la tastatură un număr `n` și apoi `n` numere naturale distinte de la `1` la `n` (o permutare a mulțimii numerelor naturale de la `1` la `n`). Determinați numărul de inversiuni ale acestei permutări. O inversiune este formată din două valori `P[i]` și `P[j]` cu `i < j` și totodată `P[i] > P[j]`.

Rezolvare

Soluția prezentată testează pe rând fiecare pereche posibilă de indici.

```
#include <iostream>
using namespace std;
int p[101];
int n, i, j, sol;
int main () {
    cin>>n;
    for (i=1;i<=n;i++) {
```

```
        cin>>p[i];
    }
    for (i=1;i<n;i++)
        for (j=i+1;j<=n;j++)
            if (p[i] > p[j])
                sol++;
    cout<<sol;
    return 0;
}
```

9. Se dau n numere naturale distincte. Determinaţi câte triunghiuri distincte pot avea lungimile laturilor printre aceste numere (n este maxim 100).

Rezolvare

Valoarea mică a lui n ne permite să fixăm în toate modurile toate tripletele de numere și la fiecare să testăm dacă valorile pot reprezenta lungimile laturilor unui triunghi (suma oricăror două să fie mai mare ca al treilea).

Observăm în cod, că pentru a nu verifica același set de 3 indici de mai multe ori, îi vom genera pe aceștia în ordine crescătoare (pornim cu j de la i+1 iar k pornește de la j+1).

```
#include <iostream>
using namespace std;
int v[101];
int n, i, j, k, sol;
int main () {
    cin>>n;
    for (i=1;i<=n;i++) {
        cin>>v[i];
    }
    for (i=1;i<n-1;i++)
        for (j=i+1;j<n;j++)
            for (k=j+1;k<=n;k++)
                if ((v[i] < v[j] + v[k])
                    && (v[j] < v[i] + v[k])
                    && (v[k] < v[i] + v[j]))
                    sol++;
    cout<<sol;
    return 0;
}
```

Ștergeri / inserări de elemente.

Folosind vectorii putem păstra orice listă de elemente de orice tip. Gândiți-vă de exemplu la lista de melodii de la player. Mereu dorim să adăugăm sau să ștergem elemente în aceasta. Ne vom ocupa acum de modul de realizare pentru astfel de operații.

1. Ștergerea elementului aflat pe o poziție dată p, în vectroul v cu n elemente.

Nu ne mulțumește o soluție care traversează șirul și care afișează toate elementele mai puțin cel de pe poziția p, ca mai jos.

```
for (i=1;i<=n;i++)
```

```
    if (i!=p)
        cout<<v[i]<<" ";
```

Această improvizație nu modifică șirul și ar fi probleme pentru operații ulterioare de realizat cu el.

Modificarea efectivă în memorie presupune mutarea cu un indice în stânga a tuturor elementelor aflate după poziția `p` iar la final actualizarea lui `n`.

```
for (i=p;i<n;i++)
    v[i] = v[i+1];
n--;
```

```
for (i=p+1;i<=n;i++)
    v[i-1] = v[i];
n--;
```

Cele două variante de mai sus sunt echivalente, ele diferă prin felul în care am gândit semnificația indicelui de la `for`: în secvența din partea stângă el reprezintă elementul către care atribui iar în cea din dreapta elementul pe care îl atribui.

2. Ștergerea unei secvențe de elemente care începe la o anume poziție `p` și are lungimea `L`.

Cel mai simplu este să folosim doi indici, unul memorând locul unde voi aduce o valoare și celălalt locul de unde aduc valoarea. După fiecare pas avansez cu ambii.

```
for (i=p,j=p+L; j<=n; i++,j++)
    v[i] = v[j];
n = i-1;
```

3. Ștergerea tuturor elementelor care îndeplinesc o anumită proprietate. Aici exemplificăm prin ștergerea tuturor elementelor pare.

Prima abordare este aceea de a traversa șirul și a aplica algoritmul de ștergere prezentat mai sus, pentru fiecare element par.

```
for (i=1;i<=n;i++)
    if (v[i]%2 == 0) {
        for (j=i;j<n;j++)
            v[j] = v[j+1];
        n--;
        i--;
    }
```

Evident că după fiecare element șters trebuie scăzut `n` cu `1`. De obicei atrage atenția `i--`. Acest lucru se explică astfel: dacă am avea două elemente pare, pe pozițiile `i` și `i+1`, după ștergerea celui de pe poziția `i`, celălalt, care trebuie șters și el ajunge pe aceeași poziție `i`. Dar forul va incrementa pe `i` la trecerea la pasul următor, iar noi trebuie să îl ținem pe loc. De aici decrementarea respectivă.

Secvența de cod de mai sus ajunge la timp de rulare de ordin n^2 daca trebuie șterse majoritatea alementelor. Există însă o abordare simplă, cu timp de calcul de ordin `n`.

Ne imaginăm că în loc să ștergem elementele pare, vom păstra elementele impare, construind cu ele un nou vector.

```
k = 0;
for (i=1;i<=n;i++)
    if (v[i] % 2 != 0) {
        k++;
        w[k] = v[i];
    }
```

Acum vectorul w conține exact elementele de ar rămâne după ștergere, pe poziții de la 1 la k. Observăm că a fost necesar doar un for pentru această operație. Singura problemă este că se dublează spațiul de memorie necesar. Dacă ne uităm însă atent, observăm că putem să înocuim în codul de mai sus pe w cu v și nu greșim. Efectiv elementele impare se duc în partea din față a lui v. Aceasta este soluția simplă, eficientă (ca timp de calcul dar și ca memorie) și elegantă.

```
k = 0;
for (i=1;i<=n;i++)
    if (v[i] % 2 != 0) {
        k++;
        v[k] = v[i];
    }
```

Elementele lui v nu se suprascriu în mod greșit întrucât cât timp avem la început doar elemente impare acestea se suprascriu pe ele însele, iar odată ce a apărut primul element par, k devine și apoi rămâne mai mic decât i.

4. Inserarea la poziția p a unei valori x într-un vector v cu n elemente.

Elementele aflate pe pozițiile p, p+1, ..., n trebuie mutate în componenta cu indicele cu 1 mai mare. Trebuie pentru asta un for asemănător cu acela de la ștergere, dar în celălalt sens. Însă, foarte important, ordinea de efectuare a atribuirilor trebuie să fie de la dreapta la stânga.

Dacă s-ar executa:

```
for (i=p;i<=n;i++)
    v[i+1] = v[i];
```

după primul pas, valoarea de pe poziția p ajunge și pe poziția p+1 (mai mult, vechea valoare de la poziția p+1 se pierde), la al doilea pas aceeași valoare o suprascrie și pe cea de la poziția p+2 ...

Soluția este:

```
for (i=n;i>=p;i--)
    v[i+1] = v[i];
v[p] = x;
n++;
```

5. Inserarea unui vector w cu m elemente, într-un vector v cu n elemente începând cu poziția p.

Soluția este asemănătoare cu cea de la probema anterioară. Mai întâi "facem loc" elementelor din w, deplasând pe cele din v de la poziția p la poziția n cu m poziții către "dreapta". Apoi ducem cele m elemente ale lui w pe pozițiile lor cu alt for. Pentru siguranța codării, aici se recomandă folosirea a doi indici care cresc cu 1 la fiecare pas. Unul în w și unul în v.

```
for (i=n;i>=p;i--)
    v[i+m] = v[i];
for (i=p,j=1;j<=m;i++,j++)
    v[i] = w[j];
n+=m;
```

6. Fiind dat un vector v cu n elemente, să se insereze după fiecare valoare pară dublul ei.

În primul rând trebuie să estimăm că numărul de elemente din vector se poate dubla, deci trebuie declarat dublu față de numărul de valori ce le poate avea initial.

Soluția prezentată în continuare parcurge elementele lui `v` de la stânga la dreapta și la întâlnirea unei valori pare aplică secvența de inserare element prezentată mai sus. Foarte important, aici trebuie să mai mărim noi `i` cu `1`, ca efect invers al celui de la ștergere, când scădeam `i` cu `1`, pentru că altfel ajungem să testăm iar un element par - cel tocmai insetat - ajungându-se la ciclu infinit.

```
for (i=1;i<=n;i++)
    if (v[i]%2 == 0){
        for (j=n;j>=i;j--)
            v[j+1] = v[j];
        n++;
        i++;
    }
```

Timpul de executare este mai sus de ordin n^2. Se poate obține timp de executare de ordin `n` prin următoarea abordare: se numără întâi valorile pare din vectorul inițial, determinându-se astfel dimensiunea finală. Apoi se merge cu doi indici, unul de la `n` înapoi și unul de la dimensiunea finală, tot înapoi, memorând elementele în soluție dinspre dreapta.

Verificarea unor proprietăți

Sunt probleme la care data de intrare este un set de elemente iar data de ieșire este un mesaj: `DA` sau `NU`.

Strategiile pe care le vom discuta aici sunt aplicabile în general la probleme cu șiruri, nu neapărat cu elementele memorate într-un vector.

Pentru început să încercăm formularea problemei la general: **dându-se un set de elemente, să se verifice dacă există cel puțin unul care îndeplinește o anume proprietate, să o notăm cu *P*. (a)**

Problema se poate pune și altfel (`b`): să se verifice dacă toate elementele îndeplinesc o anume proprietate `P`. Cazul (`b`) se reduce la cazul (`a`), problema echivalentă cu (`b`) fiind: să se verifice dacă există cel puțin un element care nu verifică proprietatea enunțată la (`b`).

O primă abordare a problemei (`a`) este următoarea: numărăm câte elemente din set îndeplinesc proprietatea `P`. Dacă rezultatul este nenul, afișăm `DA` (verificare reușită), în caz contrar afișăm `NU`.

Un alt fel de abordare este următorul: Presupunem că niciun element nu îndeplinește proprietatea (inițializăm o variabilă cu semnificație logică, `ok = fals`). Traversăm șirul și dacă un element îndeplinește proprietatea, facem `ok = true` (și așa rămâne `ok`, nu îl mai putem face la loc `false`).

Ambele strategii pot fi folosite și la o cerință de tipul (`b`).

Pentru a o folosi pe a doua (la b) procedăm astfel: O variabilă logică o setăm la început cu semnificația că toate elementele îndeplinesc proprietatea (`ok = true`). Traversând setul, dacă găsim cel puțin un element care nu îndeplinește proprietatea, facem `ok = false`.

Iată câteva exemple:

1. Se dă un șir cu `n` elemente. Să se verifice dacă există vreunul egal cu `0`.

Soluția 1: Numărăm valorile de `0`.

```
z = 0;
for (i=1;i<=n;i++)
    if (v[i] == 0)
        z++;
if (z>0)
    cout<<"da";
else
    cout<<"nu";
```

Soluția 2: Presupunem că niciun element nu e nul iar dacă găsim unul egal cu `0`, schimbăm presupunerea:

```
nule = 0;
for (i=1;i<=n;i++)
    if (v[i] == 0)
        nule = 1;
if (nule)
    cout<<"da";
else
    cout<<"nu";
```

Ambele cazuri acceptă și optimizarea: dacă se intră în `if`-ul din `for`, pe lângă setarea variabilei respective se poate executa și `break`.

O greșeală pe care o fac cei care nu au înțeles încă abordarea este de a reveni în `for` la valoarea inițială, după ce ea a fost deja setată invers.

```
nule = 0;
for (i=1;i<=n;i++)
    if (v[i] == 0)
        nule = 1;
    else
        nule = 0;
if (nule)
    cout<<"da";
else
    cout<<"nu";
```

E clar că rezultatul final depinde la codul de mai sus doar de ultimul element testat, fără a conta celelalte, lucru incorect.

2. Se dă un vector cu `n` elemente. Să se verifice dacă este ordonat strict crescător.

Reformulând, să verificăm dacă la fiecare pereche de elemente vecine `v[i]` este mai mic ca `v[i+1]`. Preupunem `ok = 1` cu semnificația de ordonat și dacă găsim că vreo pereche nu îndeplinește condiția vom face `ok = 0`.

```
ok = 1;
for (i=2;i<=n;i++)
    if (v[i-1] >= v[i])
        ok = 0;
if (ok)
    cout<<"ordonat";
else
```

```
    cout<<"nu este ordonat";
```

3. Orice şir se încadrează în următoarele categorii: *șir constant*, *șir strict crescător*, *șir crescător*, *șir strict descrescător*, *șir descrescător* sau *șir neordonat*. Se citește un șir cu `n` elemente naturale. Să se verifice în ce categorie se încadrează (pbinfo, #1320).

Vom face câte o verificare independentă pentru fiecare dintre primele `5` categorii. Dacă niciuna dintre verificări nu dă `adevărat` înseamnă că suntem în ultima categorie. La fiecare verificare setăm de la început variabila corespunzătoare ei cu adevărat și parcurgem perechile de elemente vecine din vector testând dacă găsim vreuna care să o facă `fals`. Vom folosi totuși un singur `for`, realizând în același timp cele `5` verificări.

Trebuie atenție la final întrucât e posibil ca mai multe verificări să dea `adevărat`. De exemplu, dacă șirul este strict crescător, atât variabila corespunzătoare acestui test cât și cea pentru testul crescător vor fi cu valoarea `adevărat`, dar noi va trebui să afișăm doar "strict crescător". Dacă șirul este constant, atât variabila pentru constant cât și cea pentru crescător și cea pentru descrescător vor fi setate ca `adevărate`. O ordine corectă de a testa este: constant, strict crescator, strict descrescător, crescator, descrescator.

```
#include <iostream>
using namespace std;
int n, i, v[1002];
int crescator, strictCrescator, descrescator, strictDescrescator, constant;
int main () {
    cin>>n;
    for (i=1;i<=n;i++)
        cin>>v[i];
    strictCrescator = 1;
    strictDescrescator = 1;
    crescator = 1;
    descrescator = 1;
    constant = 1;
    for (i=2;i<=n;i++) {
        if (v[i] <= v[i-1])
            strictCrescator = 0;
        if (v[i] >= v[i-1])
            strictDescrescator = 0;
        if (v[i] < v[i-1])
            crescator = 0;
        if (v[i] > v[i-1])
            descrescator = 0;
        if (v[i] != v[i-1])
            constant = 0;
    }
    if (constant == 1)
        cout<<"sir constant";
    else
        if (strictCrescator == 1)
            cout<<"sir strict crescator";
        else
            if (strictDescrescator == 1)
                cout<<"sir strict descrescator";
            else
                if (crescator == 1)
                    cout<<"sir crescator";
                else
                    if (descrescator == 1)
                        cout<<"sir descrescator";
                    else
```

```
                        cout<<"sir neordonat";
    return 0;
}
```