TIPURI STRUCTURATE DE DATE

Pentru început am scris programe care nu prelucrează decât date simple (de exemplu, numere întregi, numere reale, caractere). În realitate, un program trebuie să prelucreze volume mari de date şi pentru ca prelucrarea să se realizeze eficient este *necesară* organizarea acestor date în structuri. De exemplu, să presupunem că dorim să ordonăm elevii din şcoală în ordine alfabetică. Pentru aceasta este nevoie să reţinem „undeva" (într-o structură de date) numele şi prenumele elevilor şcolii şi abia apoi îi putem ordona.

O ***structură de date*** reprezintă un ansamblu (o colecţie) de date, organizate după anumite reguli, reguli care depind de tipul de structură.

Tablouri

Un ***tablou*** este o colecţie de date *de acelaşi tip*, memorate într-o zonă de *memorie contiguă*, reunite sub un nume comun (numele tabloului).

Declararea unei variabile de tip tablou:

```
tip nume[NrE];
```

Am declarat un tablou format din `NrE` elemente de tipul `tip`. `NrE` indică numărul de elemente din tablou şi trebuie să fie obligatoriu o expresie constantă.

Deoarece elementele unui tablou sunt memorate în ordine, unul după altul, într-o zonă contiguă, pentru a ne referi la un element al unui tablou putem specifica numele tabloului din care face parte elementul şi poziţia sa în tablou, prin numărul său de ordine (numerotarea începe de la `0`).

```
nume[indice]
```

Am specificat elementul `indice` al tabloului `nume` (`indice` reprezintă numărul de ordine al elementului în tablou, cuprins între `0` şi `NrE-1`). Parantezele pătrate (`[]`) constituie ***operatorul de indexare***. Operatorul de indexare are prioritate maximă (mai mare decât a operatorilor unari).

Exemple

1. Să declarăm un tablou cu `10` elemente de tip `int`:

```
int a[10];
```

Elementele tabloului sunt `a[0]`, `a[1]`, `a[2]`, ..., `a[9]`.

2. Să declarăm un tablou cu `100` de elemente de tip `float`:

```
float b[100];
```

Observaţii

1. La întâlnirea unei declaraţii de variabilă tablou, compilatorul verifică dacă dimensiunea zonei de memorie necesară pentru memorarea tabloului nu depăşeşte memoria disponibilă. Dimensiunea zonei de memorie necesară unui tablou se calculează înmulţind numărul de elemente cu numărul de octeţi necesari pentru memorarea unui element, adică `NrE*sizeof(tip)`.
2. Ca în cazul oricărei declaraţii de variabile, putem iniţializa elementele unei variabile tablou, chiar de la declarare.

```
tip nume[NrE]={val0, val1, ..., valk};
```

Ca urmare se vor atribui în ordine elementelor tabloului valorile din lista de iniţializare (`k≤NrE`). Dacă tabloul este integral iniţializat la declarare, nu este necesar să mai specificăm dimensiunea sa, fiind considerată egală cu numărul de valori din lista de iniţializare. De exemplu:

```
int a[]={12,20,30,5};
```

Am declarat un tablou cu `4` elemente de tip `int` iniţializate astfel:

12	20	30	5
a[0]	a[1]	a[2]	a[3]

3. Un astfel de tablou, pentru care la declarare este specificată o singură dimensiune, iar poziţia unui element este specificată utilizând un singur indice, se numeşte ***tablou unidimensional*** sau ***vector***.
4. Elementele unui tablou pot fi de orice tip al limbajului. Prin urmare.... elementele unui tablou, pot fi de tip tablou! Declarare:

```
tip nume[Nr1][Nr2];
```

Am declarat un tablou cu `Nr2` elemente, fiecare element fiind un tablou cu `Nr1` elemente de tipul specificat. Un astfel de tablou, pentru care la declarare trebuie să specificăm două dimensiuni, iar poziţia unui element este specificată utilizând doi indici, se numeşte ***tablou bidimensional*** sau ***matrice***.

Putem să ne imaginăm un tablou bidimensional ca pe o tablă de şah. Poziţia unui element pe tablă este identificată prin doi indici: linia şi coloana. Prin analogie cu tabla de şah, şi la informatică primul indice utilizat în referirea unui element este denumit „indice de linie", iar cel de al doilea indice este denumit „indice de coloană".

De exemplu să declarăm o matrice cu două linii şi trei coloane cu elemente întregi, pe care o vom iniţializa la declarare:

```
int a[2][3]={{1,2,3},{4,5,6}};
```

a	Coloana 0	Coloana 1	Coloana 2
Linia 0	1	2	3
Linia 1	4	5	6

Pentru a ne referi la un element al unei matrice, specificăm numele matricei, indicele de linie şi indicele de coloană astfel:

```
nume[indice_linie][indice_coloana]
```

De exemplu, pentru a ne referi la elementul de pe linia `1`, coloana `2` din matricea `a`, vom scrie `a[1][2]`.

5. Elementele unui tablou bidimensional pot fi de orice tip, inclusiv... tablou! Se obţin astfel tablouri multidimensionale. Să declarăm de exemplu un tablou tridimensional (vă imaginaţi un paralelipiped cu dimensiunile 10, 5, 40):

```
int a[10][5][40];
```

Prelucrări elementare pe vectori

Citirea unui vector

Citirea unui vector se realizează citind elementele vectorului, unul câte unul. Să considerăm ca exemplu un vector a cu n elemente (n≤100) de tip int.

```
int a[100], n, i;          //declarare vector
cout << "n= "; cin >>n;    //citesc numarul de componente
for (i=0; i<n; i++)        //citesc succesiv componentele
     {cout <<"a["<<i<<"]="; cin>>a[i];}
```

Afişarea unui vector

Afişarea unui vector se realizează afişând pe rând componentele vectorului.

```
for (i=0; i<n; i++)        //afisez succesiv componentele
     cout << a[i] <<' ';
```

Copierea unui vector

Să presupunem că dorim să copiem vectorul a într-un alt vector b. Copierea unui vector nu se poate face printr-o atribuire de forma b=a (veţi obţine un mesaj de eroare). Copierea unui vector se realizează element cu element:

```
for (i=0; i<n; i++) b[i]=a[i];
```

Determinarea elementului maxim/minim dintr-un vector

Pentru a determina cel mai mare element din vector, vom considera o variabilă (să o numim max) în care vom reţine la fiecare pas maximul dintre elementele analizate. Iniţializăm variabila max cu un element din vector (de exemplu, cu a[0]). Parcurgem apoi vectorul, comparând fiecare element din vector cu max, şi actualizând eventual maximul după comparare.

```
max=a[0];           //initializez maximul cu primul element
for (i=1; i<n; i++) //compar succesiv componentele cu max
     if (max<a[i]) max=a[i];            //actualizez max
```

Exerciţii

1. Modificaţi secvenţa de instrucţiuni precedentă, astfel încât să determine cel mai mic element din vector.

2. Modificaţi secvenţa de instrucţiuni precedentă, astfel încât să determine maximul, după iniţializarea lui `max` cu ultimul element din vector.

Media aritmetică a elementelor strict pozitive

Se consideră un tablou de `10` numere întregi. Scrieţi un program care citeşte de la tastatură cele `10` componente ale vectorului şi afişează pe ecran media aritmetică a valorilor strict pozitive din vector, cu două zecimale. (variantă bacalaureat, 2000)

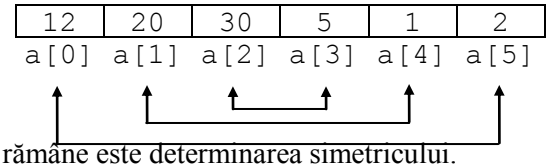
Soluţie

După citire, se parcurge vectorul, verificând pentru fiecare element dacă este sau nu strict pozitiv. Când găsim un element strict pozitiv, îl numărăm şi îl adunăm la suma elementelor strict pozitive:

```
#include <iomanip.h>
void main()
{ float a[10], s=0;
  int i, nr=0;
  for (i=0; i<10;i++) cin >> a[i];
  for (i=0; i<10; i++)
      if (a[i]>0) s+=a[i], nr++;
  if (nr) cout << setprecision(2) << s/nr;
      else cout << "Nu exista elemente strict pozitive";}
```

Inversarea ordinii elementelor din vector

Pentru a inversa ordinea elementelor dintr-un vector trebuie ca primul element să fie interschimbat cu ultimul, al doilea element cu penultimul, ş.a.m.d. Mai exact, trebuie să parcurgem vectorul până la mijloc şi să interschimbăm elementele simetrice faţă de mijloc. De exemplu:



Problema care rămâne este determinarea simetricului.

Poziţia elementului	Poziţia simetricului	Observaţie
0	n-1	0+n-1=n-1
1	n-2	1+n-2=n-1
2	n-3	2+n-3=n-1

...

Observăm că atunci când poziţia elementului creşte cu `1`, poziţia simetricului scade cu `1`, deci suma dintre poziţia elementului şi poziţia simetricului este constantă `(n-1)`. Deducem că simetricul elementului `a[i]` este `a[n-1-i]`.

```
for (i=0; i<n/2; i++)
    {aux=a[i]; a[i]=a[n-1-i]; a[n-i-1]=aux;}
```

Verificarea unei proprietăţi

Frecvent apar probleme în care se cere să se verifice dacă toate elementele unui vector au o anumită proprietate *P* sau dacă există în vector un element care are proprietatea *P*. Pentru a descrie o secvenţă de instrucţiuni cât mai generală, considerăm că `P(x)` are valoarea `1` dacă `x` are proprietatea respectivă şi `0`, altfel.

a. Pentru a verifica dacă toate elementele unui vector au proprietatea *P*, vom verifica succesiv fiecare element din vector. Pentru a reţine rezultatul verificării, vom considera o variabilă întreagă `ok`, care va avea valoarea `1` dacă toate elementele vectorului au proprietatea *P* şi `0`, altfel. Iniţial variabilei `ok` îi atribuim valoarea `1` (ipoteza optimistă, „prezumţia de nevinovăţie" – din moment ce nu am găsit încă nici un element care să conteste acest lucru, putem presupune că toate elementele vectorului au proprietatea *P*). Dacă, parcurgând vectorul, vom găsi un element care nu are proprietatea respectivă, variabilei `ok` îi vom atribui valoarea `0`:

```
for (ok=1, i=0; i<n && ok; i++)
//parcurg vectorul pana la sfarsit sau pana la intalnirea
//unui element care nu are proprietatea P
     if ( !P(a[i]) ) ok=0; //a[i] nu are proprietatea P
```

b. Pentru a verifica dacă există un element în vector cu proprietatea *P*, vom parcurge vectorul (până la sfârşit sau până la întâlnirea unui element cu proprietatea *P*). Rezultatul verificării îl vom reţine în variabila întreagă `gasit`. Iniţial (deoarece nu am găsit încă nici un element cu proprietatea *P*) îi atribuim variabilei `gasit` valoarea `0`. Dacă vom găsi în vector un element cu proprietatea *P*, îi vom atribui variabilei `gasit` valoarea `1`:

```
for (gasit=i=0; i<n && !gasit; i++)
     if ( P(a[i]) ) gasit=1; //a[i] are proprietatea P
```

Căutarea unui element într-un vector

Fie `a` un vector cu `n` (`n≤100`) componente întregi şi `x` o valoare întreagă. Verificaţi dacă `x` apare sau nu în vectorul `a`.

Practic, trebuie să verificăm dacă există în vectorul `a` un element cu proprietatea că este egal cu `x`. Particularizând secvenţa de verificare de la punctul b, obţinem:

```
for (gasit=i=0; i<n && !gasit; i++)
     if (a[i]==x) gasit=1;  //am gasit valoarea x in vector
```

Această metodă de căutare, în care testăm succesiv elementele vectorului, se numeşte ***căutare secvenţială***. În cazul cel mai defavorabil (când `x` nu se găseşte în vector sau este plasat pe ultima poziţie) căutarea secvenţială efectuează `n` comparaţii. O soluţie mai eficientă nu există, în cazul general.

În realitate ne confruntăm frecvent cu problema căutării unui element într-o mulţime şi frecvent mulţimea respectivă este ordonată (de exemplu, căutarea unui cuvânt în dicţionar, a unui număr în cartea de telefon, etc). În astfel de situaţii nu aplicăm o căutare secvenţială (de exemplu, pentru a găsi în cartea de telefon numărul lui Popescu Ion nu începem să căutăm de la litera `'A'`).

Cum procedăm? Deschidem cartea de telefon la întâmplare. Verificăm dacă am dat peste Popescu Ion. Dacă nu, verificăm dacă Popescu Ion este în prima parte a cărţii sau în cea de a doua şi continuăm căutarea numai în porţiunea respectivă. Procedeul se repetă până când găsim sau până când nu mai avem unde căuta.

Aceeaşi idee poate fi aplicată şi pentru căutarea unui element într-un vector ordonat. Cum pentru calculator este dificil să lucreze „la întâmplare”, vom lucra prin înjumătăţiri succesive: mai întâi comparăm elementul căutat cu elementul din mijloc. Dacă este egal, am găsit, am terminat. Dacă nu este egal, verificăm dacă elementul căutat este mai mare decât elementul din mijloc (în acest caz căutăm mai departe numai în cea de a doua jumătate). Dacă elementul căutat este mai mic decât elementul din mijloc, căutăm mai departe numai în prima jumătate.

```
for (st=0, dr=n-1, gasit=0; !gasit && st<=dr;)
//cautam pe x de la pozitia st la pozitia dr; cautarea
//continua pana gasim sau pana nu mai avem unde cauta
    { mijloc=(st+dr)/2;                 //calculam mijlocul
      if (a[mijloc]==x) gasit =1;       //am gasit pe x
         else
         if (a[mijloc]<x) st=mijloc+1; //caut in stanga
             else dr=mijloc-1;}        //caut in dreapta
if (gasit) cout<<x<<" se gaseste pe pozitia "<<mijloc;
    else cout <<x<<" nu se afla in vector ";
```

Acest algoritm de căutare se numeşte ***căutare binară***. Denumirea este sugestivă: la fiecare pas, alegem una din cele două alternative posibile: caut la stânga elementului din mijloc sau în dreapta lui.

Exerciţiu

Să considerăm un vector cu `n=7` elemente ordonate crescător. Câte comparaţii execută algoritmul de căutare binară în cazul cel mai defavorabil (când elementul căutat nu se află în vector sau este depistat abia la ultima comparaţie)? Dar pentru `n=10` elemente? Puteţi estima în general (în funcţie de `n`) numărul de comparaţii necesare în cazul cel mai defavorabil? Ce concluzie trageţi, comparând algoritmul de căutare binară cu algoritmul de căutare secvenţială?

Sortare

Fie `n` (`n`∈**N***) elemente a_0, a_1, …, a_{n-1} dintr-o mulţime total ordonată. Ordonaţi crescător elementele a_0, a_1, …, a_{n-1}.

Problema ordonării unor elemente (cunoscută şi sub denumirea de sortare) este frecvent întâlnită în practică şi din acest motiv a fost studiată intens. Ca urmare, au fost elaboraţi numeroşi algoritmi de sortare. Cum de obicei numărul de elemente care trebuie să fie ordonate este mare, s-a studiat şi eficienţa acestor algoritmi, în scopul elaborării unor algoritmi de sortare performanţi. Pentru început, vom studia algoritmi de sortare simpli, nu performanţi, urmând ulterior să învăţăm să evaluăm eficienţa acestor algoritmi şi să elaborăm algoritmi eficienţi.

Sortare prin selecţie

Sortarea prin selecţie are două variante: sortarea prin selecţia elementului maxim şi sortarea prin selecţia elementului minim. În ambele variante, ideea de bază este aceeaşi: se selectează cel mai mare element din vector (sau cel mai mic) şi se plasează pe ultima poziţie în vector (respectiv, pe prima poziţie). Apoi se calculează cel mai mare dintre elementele rămase şi se plasează pe penultima poziţie în vector (sau cel mai mic dintre elementele rămase şi se plasează pe a doua poziţie), ş.a.m.d. Acest procedeu se repetă de `n-1` ori.

```
for (dr=n-1; dr>0; dr--)   //calculez maximul de la 0 la dr
    {for (max=a[0],pozmax=0,i=1; i<=dr; i++)
          if (a[i] > max)  max=a[i], pozmax=i;
     a[pozmax]=a[dr];      //plasez maximul pe pozitia dr
     a[dr] = max; }
```

Observaţie

La fiecare iteraţie a ciclului `for` exterior este calculat max{a_0, a_1, …, a_{dr}} şi este plasat pe poziţia `dr`, elementele de la `dr+1` la `n-1` fiind deja plasate pe poziţiile lor definitive. Pentru a calcula max{a_0, a_1, …, a_{dr}} sunt necesare `dr` comparaţii. Deci, în total se execută `1+2+...+n-1=n·(n-1)/2` comparaţii, indiferent de ordinea iniţială a elementelor vectorului.

Exerciţii

1. Modificaţi algoritmul precedent, astfel încât să realizeze ordonarea descrescătoare a elementelor vectorului.
2. Modificaţi algoritmul precedent, astfel încât să realizeze ordonarea prin selecţia minimului.

Sortare prin compararea vecinilor (bubblesort)

O altă metodă de sortare este de a parcurge vectorul, comparând fiecare două elemente vecine şi (dacă este cazul) interschimbându-le. Cum într-o singură trecere nu se poate realiza sortarea vectorului, acest procedeu se repetă până când vectorul devine sortat (la ultima trecere nu am mai efectuat nici o interschimbare).

```
do
  {schimb=0;          //initial nu am facut nici o schimbare
   for (i=0; i<n-1; i++)                //parcurg vectorul
        if (a[i]>a[i+1])      //compar doua elemente vecine
           { //nu sunt in ordine, le interschimb
```

```
            aux=a[i]; a[i]=a[i+1]; a[i+1]=aux;
            schimb=1;}      //retin ca am facut schimbari
  }    //procedeul se repeta cat timp se executa schimbari
while (schimb);
```

Observaţii

1. Metoda este denumită şi *bubblesort* (metoda bulelor) deoarece la fiecare nouă trecere prin vector elementele cu valori mari migrează către sfârşitul vectorului („urcă” în vector, aşa cum bulele de aer se ridică la suprafaţă când fierbe apa).
2. În cazul cel mai defavorabil (când valorile sunt ordonate descrescător), această metodă execută aproximativ n^2 operaţii.

Exerciţii

1. Modificaţi algoritmul precedent, astfel încât să realizeze ordonarea descrescătoare a elementelor vectorului.
2. Observăm că dacă la parcurgerea curentă ultima interschimbare a fost efectuată la poziţia `x`, în oricare dintre parcurgerile ulterioare nu se vor efectua interschimbări după poziţia `x`. Îmbunătăţiţi algoritmul precedent utilizând această observaţie.

Sortare prin inserţie

Metoda de sortare prin inserţie, este de asemenea o metodă simplă, pe care o utilizăm adesea când ordonăm cărţile la jocuri de cărţi: de fiecare dată când tragem o carte o plasăm pe poziţia sa corectă, astfel încât în mână cărţile să fie ordonate.

Utilizând această idee, sortăm vectorul astfel: parcurgem vectorul, element cu element; la fiecare pas `i`, căutăm poziţia corectă a elementului curent `a[i]`, astfel încât secvenţa `a[0]`, `a[1]`, ..., `a[i]` să fie ordonată:

```
for (i=1; i<n; i++)
    {v=a[i];                  //caut pozitia lui v
     for (poz=i; poz && a[poz-1]>v; poz--)
         a[poz]=a[poz-1];   //mut la dreapta elementele > v
     a[poz] = v;  }     //poz este pozitia corecta pentru v
```

Sortare prin numărarea aparţiilor

În anumite probleme, elementele vectorului au ca valori numere naturale dintr-un interval `[0, Max)` de dimensiune redusă (sau au valori care pot fi asociate numerelor naturale dintr-un astfel de interval). Prin dimensiune redusă înţelegem că se poate declara un vector `V` cu `Max` componente întregi.

În acest caz particular cea mai bună metodă de sortare este sortarea prin numărarea aparţiilor: se numără pentru fiecare valoare din intervalul `[0, Max)` de câte ori apare în vector.

```
#include <iostream.h>
#define Max 1000
```

```
#define NrMax 10000

int n, V[Max], a[NrMax];
int main()
{int x, i, j, nr;
 cout<<"n="; cin>>n;
 for (i=0; i<n; i++)
     {cin>>x;  //citesc o noua valoare
     V[x]++;} //marim numarul de aparitii ale valorii x
//plasam in ordine elementele in vectorul a
for (nr=i=0; i<Max; i++)
    for (j=0; j<V[i]; j++)
        a[nr++]=i;
for (i=0; i<n; i++) cout<<a[i]<<' ';
cout<<endl; return 0;}
```

Interclasare

Fie `a` un vector cu `n` elemente şi `b` un vector cu `m` elemente, ordonaţi crescător. Să se construiască un al treilea vector, `c`, care să conţină atât elementele vectorului `a`, cât şi elementele vectorului `b`, în ordine crescătoare.

O soluţie (ineficientă, care nu ţine cont de faptul că `a` şi `b` sunt deja ordonaţi) este să copiem în vectorul `c` elementele din `a` şi din `b` şi apoi să sortăm vectorul `c`.

O altă idee este de a parcurge simultan cei doi vectori, comparând la fiecare pas elementul curent din `a` cu elementul curent din `b`. Cel mai mic dintre cele două elemente va fi copiat în vectorul `c`, avansând doar în vectorul din care am copiat. Când am epuizat elementele din unul din cei doi vectori copiem elementele rămase în celălalt, deoarece nu mai avem cu ce le compara.

```
for (i=j=k=0; i<n && j<m;)
     if (a[i]<b[j])    //copiez in c elementul cel mai mic
        c[k++]=a[i++];                    //copiez din a
        else c[k++]=b[j++];               //copiez din b
//copiez eventualele elemente ramase in a
for (; i<n; i++)
    c[k++]=a[i];
//copiez eventualele elemente ramase in b
for (; j<m; j++)
    c[k++]=b[j];
```

Aplicaţii

Ciurul lui Eratostene

Fie `n` un număr natural (`n≤10000`). Să se genereze toate numerele prime mai mici decât `n`.

Soluţie

O primă idee ar fi să parcurgem toate numerele naturale din intervalul [2, n], pentru fiecare număr să verificăm dacă este prim şi să afişăm numerele prime determinate.

O idee mai eficientă provine in antichitate şi poartă numele ciurul lui Eratostene[1]. Ideea este de a „pune în ciur" toate numerele mai mic decât , apoi de a „cerne" aceste numere până rămân în ciur numai numerele prime. Mai întâi „cernem" (eliminăm din ciur) toţi multiplii lui 2, apoi cernem multiplii lui 3, ş.a.m.d.

Vom reprezenta ciurul ca un vector cu 10000 de componente care pot fi 0 sau 1, cu semnificaţia ciur[i]=1 dacă numărul i este în ciur şi 0 altfel.

Vom parcurge vectorul ciur de la 2 (cel mai mic număr prim), până la $\sqrt{n}$).

Dacă ciur[i] este 1 deducem că i este prim, dar toţi multiplii lui nu vor fi (deci eliminăm din ciur toţi multiplii lui i).

În final în vectorul ciur vor avea valoarea 1 doar componentele de pe poziţii numere prime.

```
#include <iostream.h>
#define NMax 10000
int main()
{int ciur[NMax], n, i, j;
 cout<<"n= "; cin>>n;
 //initial toate numerele sunt in ciur
 for (i=2; i<n; i++) ciur[i]=1;
 for (i=2; i*i<=n; i++)
     if (ciur[i])//i este prim
        //elimin toti multiplii lui i
        for (j=2; j*i<n; j++)
            ciur[i*j]=0;
 for (i=2; i<n; i++)
     if (ciur[i]) cout<<i<<' ';
 return 0; }
```

Subsecvenţă de sumă maximă

Fie $x=(x_0, x_1, ..., x_{n-1})$ o secvenţă de numere întregi, dintre care cel puţin un element pozitiv. Să se determine o subsecvenţă de elemente consecutive $x_i, x_{i+1}, ..., x_j$, astfel încât suma elementelor din subsecvenţă să fie maximă. De exemplu,

1. Eratostene (276–196 î.e.n) a fost un important matematician şi filosof al antichităţii. Deşi puţine dintre lucrările lui s-au păstrat, a rămas celebru prin metoda rapidă de determinare a numerelor prime şi prin faptul că a fost primul care a estimat cu acurateţe diametrul Pământului.

pentru x=(1, 10, 3, -5, 4, 2, -100, 30, 15, 25, -10, 40, 1000) subsecvenţa de sumă maximă începe la poziţia 7, are lungimea 6, iar suma elementelor este 1100.

Soluţia 1

Considerăm toate subsecvenţele de elemente consecutive, calculăm suma elementelor din subsecvenţă şi reţinem poziţia de început şi lungimea subsecvenţei de sumă maximă.

```
#include <iostream.h>
int main ()
{int x[50], n, i, st, dr, poz, Lg, Sum, SMax;
 cout <<"n="; cin >> n;
 for (i=0; i<n; i++) {cout <<"x["<<i<<"]="; cin>>x[i];}
 for (SMax=x[0], st=0; st<n; st++)
     for (dr=st; dr<n; dr++)
         {for (Sum=0,i=st; i<=dr; i++) Sum += x[i];
              if (SMax<Sum) SMax=Sum, Lg=dr-st+1, poz=st; }
cout <<"Poz= "<<poz<<" ;Lg= "<<Lg<< " ; Smax= "<<SMax;
return 0;}
```

Datorită celor trei cicluri `for` imbricate, algoritmul realizează un număr de operaţii comparabil (ca ordin de mărime) cu n^3.

Soluţia 2

Pentru a calcula suma elementelor din subsecvenţa de la `st` la `dr`, vom folosi suma deja calculată a elementelor de la `st` la `dr-1`. Secvenţa de instrucţiuni care rezolvă cerinţele problemei devine:

```
for (SMax=x[0], st=0; st<n; st++)
     for (Sum=0,dr=st; dr<n; dr++)
          {Sum += x[dr];
           if (SMax < Sum) SMax=Sum, poz=st, Lg=dr-st+1;}
```

În acest caz, numărul de operaţii efectuate de algoritm este comparabil (ca ordin de mărime) cu n^2. Spunem că algoritmul este *pătratic*.

Soluţia 3

O dată cu parcurgerea vectorului `x` de la stânga la dreapta, se va memora suma maximă calculată până la momentul curent, în variabila `SMax`. în variabila `Sum` este reţinută suma elementelor subsecvenţei curente. Elementul `x[i]` va fi înglobat în subsecvenţa curentă dacă `Sum+x[i]`≥0. Dacă `Sum+x[i]`<0, înglobarea lui `x[i]` în orice subsecvenţă ar conduce la o subsecvenţă cu suma elementelor mai mică decât cea maximă până la momentul curent, prin urmare `x[i]` trebuie ignorat, iar noua subsecvenţă curentă va începe la poziţia următoare. Mai exact, la pasul `i`: `Sum=max{Sum, 0}+x[i]; SMax=max{SMax, Sum}`. Secvenţa de instrucţiuni care rezolvă cerinţele problemei devine:

```
for (SMax=Sum=x[0],st=poz=0, Lg=i=1; i<n; i++)
     if (Sum < 0)    //noua subsecventa incepe pe pozitia i
```

```
        Sum = x[i], st=i;
        else        //inglobez x[i] in subsecventa curenta
        {Sum += x[i];
         if (SMax<Sum) //subsecventa curenta este maximala
            SMax=Sum, poz=st, Lg=i-st+1;}
```

În acest caz, numărul de operaţii efectuate de algoritm este comparabil (ca ordin de mărime) cu n. Spunem că algoritmul a devenit *liniar*.

Probleme propuse

1. *Melci*

Se consideră n stâlpi (numerotaţi de la 1 la n, n<20) de înălţimi h_1, h_2, … , h_n metri. La baza fiecărui stâlp se află câte un melc codificat prin numărul stâlpului. Fiecare melc i urcă ziua p_i metri şi coboară noaptea q_i metri ($p_i \geq q_i$). Scrieţi un program care să afişeze melcii în ordinea în care ating vârfurile stâlpilor.

De exemplu, să considerăm 3 stâlpi cu înălţimile 2, 4, 5. Melcul aflat la baza stâlpului 1 urcă 1 m pe zi şi coboară 0 m pe noapte. Melcul aflat la baza stâlpului 2 urcă 4 m pe zi şi coboară 1 m pe noapte. Melcul aflat la baza stâlpului 3 urcă 1 m pe zi şi coboară 0 m pe noapte.

Melcul 1 ajunge în vârful stâlpului după două zile. Melcul 2 ajunge în vârful stâlpului după o zi. Melcul 3 ajunge în vârful stâlpului după 5 zile. Deci ordinea în care melcii ajung în vârf este 2 1 3. (Concurs PACO 1997)

2. *Mulţimi*

Se citesc de la tastatură două mulţimi A şi B cu n, respectiv m elemente numere naturale mai mici decât 1000. Să se determine intersecţia şi reuniunea celor două mulţimi. Implementaţi mai întâi o mulţime ca un vector în care memoraţi elementele mulţimii, apoi prin vector caracteristic. Realizaţi o comparaţie între eficienţa implementării celor două operaţii cu cele două reprezentări.

3. *Zig-zag*

Din fişierul ZigZag.in se citesc N numere întregi (N≤1000). Afişaţi pe prima linie a fişierului ZigZag.out cel mai lung MZigZag care se poate construi din numerele citite. Numim MZigZag o secvenţă de numere a_1, a_2, ..., a_m astfel încât $a_1 \leq a_2 \geq a_3 \leq a_4 \geq a_5 \leq \ldots \; a_{m-1} \geq a_m$. De exemplu:

ZigZag.in	ZigZag.out
7 7 5 0 1 4 9 3	0 9 3 7 1 5 4

Tema arhiva campion: jocprim, meteo, bursa, schi