Sortarea vectorilor

Ordonarea elementelor unui tablou unidimensional (sortarea) este poate cea mai necesară (și des întâlnită) operație cu șiruri. Mereu avem nevoie de acest lucru în programe. De asemenea, căutarea unei valori într-un șir se face mult mai rapid dacă acesta este sortat (căutare binară). Și sunt multe exemple.

Există foarte mulți algoritmi de sortare, de complexități diferite, în materialul de față ne vom ocupa de câteva metode cu ordin de complexitate n^2.

Sortarea prin comparare

Din punctul meu de vedere este algoritmul care se scrie cel mai ușor, necesită cele două `foruri` care dau complexitatea, un `if` și o interschimbare. Să scriem algoritmul și îl analizăm după (presupunem că avem un vector `v` de elemente `int`, cu `n` componentele pe poziții de la `1` la `n`).

```
for (i=1;i<n;i++)
    for (j=i+1;j<=n;j++)
        if (v[i] > v[j]) {
            aux = v[i];
            v[i] = v[j];
            v[j] = aux;
        }
```

Când `i` este `1` se compară `v[1]` cu toate componentele de după, iar când se găsește o valoare mai mică se realizează interschimare, aceasta ajungând pe poziția `1` și ea continuă să se compare cu restul elementelor. Așadar, când se întâlnește minimul acesta va fi dus pe poziția `1`. Când `i` devine `2` la primul `for` se aplică același principiu și minimul din restul elementelor este adus pe poziția `2`. Tot așa, la fiecare pas `i` se duce cea mai mică valoare rămasă pe poziția `i`.

Timpul de executare este de ordin n^2, mai exact se fac $n \cdot (n-1)/2$ operații de comparare. Cazul cel mai defavorabil din punct de vedere al interschimbărilor este tot $n \cdot (n-1)/2$, când șirul este ordonat descrescător.

Și în acest caz, ca și la ceilalți algoritmi de sortare, diferența între a sorta crescător și a sorta descrescător se face prin operatorul relațional folosit la comparare în `if`.

Un alt lucru de subliniat de la început: programatorii neexperimantați au tendința de a automatiza reținerea acestui algoritm și de foarte multe ori greșesc, făcând inițializarea la al doilea for de la `2` în loc de `i+1`. Acest lucru duce la un rezultate eronat, șirul final nesortându-se (ieea algoritmului este de a avea elementele din față deja sortate și de a compara pe cel de pe poziția unde trebuie să aducem următorul minim doar cu elemente aflate după; dacă am inițializa cu `2` am reveni la compararea unui element cu altele din urmă).

„Bubble sort”

La sortarea prin comparare operația de interschimbare se face între elemente aflate la orice distanță unul de altul. În acest caz vom compara mereu doar elemente vecine.

Astfel, la un moment dat facem o trecere prin vector făcând cele `n-1` comparări între elemente vecine iar când e cazul interschimbăm.

```
for (i=1;i<n;i++)
    if (v[i] > v[i+1]) {
```

```
        aux = v[i];
        v[i] = v[i+1];
        v[i+1] = aux;
    }
```

O astfel de trecere nu sortează șirul (nu există sortare cu o singură structură repetitivă sau fără recursivitate) dar mai așează dintre elemente în ordine și, mai cu seamă, duce maximul la final (când va fi întâlnit el se va tot interschimba până la final).

Deducem că dacă am repeta secvența efectul obținut este de a duce și al doilea maxim la locul său, etc. Așadar după `n-1` astfel de treceri șirul se sortează.

Odată cu secvența de cod de mai sus putem să verificăm și dacă la o trecere șirul este deja sortat. În acest fel, după trecerea la care verificarea da rezultatul *sortat* ne putem opri. De exemplu dacă șirul este deja sortat la început se face o singură parcurgere, cea după care se observă ordonarea.

```
do {
    sortat = 1;
    for (i=1;i<n;i++)
        if (v[i] > v[i+1]) {
            aux = v[i];
            v[i] = v[i+1];
            v[i+1] = aux;
            sortat = 0;
        }
} while (sortat == 0);
```

Am folosit repetiția cu test final întrucât oricum este necesară o parcurgere a șirului. Așadar timpul de executare este tot de ordin n^2 pe cazul cel mai defavorabil dar pe cazul mediu se obține un timp mai bun. Se ajunge chiar la timp de ordin `n` dacă șirul este de la început aproape sortat.

Sortarea prin selecție

Aici, la primul pas, parcurgem șirul și determinăm maximul și poziția sa, `poz`, interschimbând apoi pe `v[n]` cu `v[poz]`. La al doilea pas aflăm maximul din primele `n-1` elemente și, ca mai sus, ducem maximul acolo. După `n-1` astfel de pași șirul se sortează.

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

Punctul forte al acestei metode este numărul mic de interschimbări: maxim `n-1`, deci de ordin `n`. Numărul de comparări este ca în cazul celorlalte metode, de ordin n^2 .

Sortarea prin inserare

Presupunem că avem un șir sortat cu `n` elemente (memorate pe poziții de la `1` la `n`) și dorim să inserăm în el un nou element, `x`, așa încât șirul cu `n+1` elemente obținut să fie și el sortat. Obținem asta cu următoarea secvență:

```
k=n;
while(k>0  && v[k] > x) {
    v[k+1] = v[k];
    k--;
}
k++;
v[k] = x;
```

Așadar elementele din dreapta mai mari decât cel de inserat se deplasează o poziție la dreapta, păstrându-și deci ordinea relativă făcând loc elementului nou.

Noi vom aplica strategia de mai sus de mai multe ori pe șirul de sortat, gândind astfel, inductiv: "secvența" formată doar din primul element este un sir sortat deci putem considera pe `v[2]` pe post de `x` și îl inserăm în șirul sortat obținând un șir sortat cu două elemente. Secvența cu primele două elemente este acum un șir sortat și inserăm în acesta pe `v[3]` ...

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

Cumva acest algoritm este similar ca performanță cu bubble sort. Dacă elementul de inserat este mai mare decât cel de dinaintea lui instrucțiunea `while` nu face niciun pas. Adică dacă șirul ar fi aproape sortat avem timp de ordin `n`. Totuși, pe cazul cel mai defavorabil, al unui șir sortat descrescător, ajungem la timp de calcul de ordin n^2.

Concluzii

Metoda	Observații
Sortare prin comparare	Timp de ordin n^2 în orice caz.
Bubble sort	Timp de ordin n^2 pe cazul cel mai defavorabil dar timpul poate fi mai bun în funcție de structura inițială a șirului.
Sortarea prin selecție	Timp de ordin n^2 pe toate cazurile. Număr de interschimbări de ordin `n` pe cazul cel mai defaborabil.
Sortarea prin inserare	Timp de ordin n^2 pe cazul cel mai defavorabil dar timpul poate fi mai bun în funcție de structura inițială a șirului.
Noi am prezentat algoritmi de sortare pentru elementele unui șir aflate pe poziții de la `1` la `n`. Remarcăm că ei se pot aplica pe orice interval de indici `st`, `dr`.	

Probleme rezolvate

1. Se dă un vector cu `n` elemente, numere naturale distincte. Ordonați crescător elementele situate înaintea valorii maxime din vector și descrescător elementele situate după această valoare (pbinfo.ro, #512).

Rezolvare

Determinăm poziția maximului, `poz` și acum avem de realizat două sortări: una crescătoare pentru indici între `1` și `poz-1` și una descrescătoare pentru indici între `poz+1` și `n` (maximul nu are sens să îl deranjăm căci este deja la locul său).

```
#include <iostream>
using namespace std;
int v[1001];
int n, i, j, aux, maxim, poz;
int main () {
    cin>>n;
    for (i=1;i<=n;i++)
        cin>>v[i];
/// aflu pozitia maxumului
    maxim = v[1];
    poz = 1;
    for (i=2;i<=n;i++)
        if (v[i] > maxim) {
            maxim = v[i];
            poz = i;
        }
    for (i=1;i<poz;i++)
        for (j=i+1;j<=poz;j++)
            if (v[i] > v[j]) {
                aux = v[i];
                v[i] = v[j];
                v[j] = aux;
            }
    for (i=poz+1;i<n;i++)
        for (j=i+1;j<=n;j++)
            if (v[i] < v[j]) {
/// diferenta intre sortarea crescatoare si cea descrescatoare
/// este de un singur caracter, semnul compararii
                aux = v[i];
                v[i] = v[j];
                v[j] = aux;
            }
    for (i=1;i<=n;i++)
        cout<<v[i]<<" ";
    return 0;
}
```

2. Să se scrie un program care ordonează crescător elementele situate pe poziții pare într-un vector și descrescător elementele situate pe poziții impare. Considerăm elementele stocate în pe poziții de la `1` la `n`.

Rezolvare

Și aici putem folosi oricare dintre metodele de sortare, inclusiv bubble sort, elementele vecine, care trebuie comparate la fiecare pas, considerându-se acum la distanța 2. Facem deci, de asemenea, două sortări, una pentru indicii pari și una pentru cei impari. Ele sunt similare, diferențele fiind doar indicele de pornire și operatorul folosit la comparare.

```
#include <iostream>
using namespace std;
int v[102], n, i, sortat;
int main () {
    cin>>n;
    for (i=1;i<=n;i++)
        cin>>v[i];
    do {
        sortat = 1;
        for (i=1;i+2<=n; i+=2)
            if (v[i] < v[i+2]) {
                int aux = v[i];
                v[i] = v[i+2];
                v[i+2] = aux;
                sortat = 0;
            }
    } while (sortat == 0);
    do {
        sortat = 1;
        for (i=2;i+2<=n; i+=2)
            if (v[i] > v[i+2]) {
                int aux = v[i];
                v[i] = v[i+2];
                v[i+2] = aux;
                sortat = 0;
            }
    } while (sortat == 0);
    for (i=1;i<=n;i++)
        cout<<v[i]<<" ";
    return 0;
}
```

3. Dat fiind un vector cu n elemente aflate pe poziții de la 1 la n, se cere să sortăm elementele pare între pozițiile lor, cele impare rămânând peloc. La final trebuie afișate toate cele n elemente în noua ordine.

Rezolvate

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

În acest caz observăm că nu mai putem folosi bubble sort (iar sortarea prin inserție mai greu) pentru că elementele de comparat (și eventual interschimbat) sunt pe poziții la diverse distanțe.

4. Dat fiind un vector cu elemente distincte memorate pe poziții de la 1 la n, să se afișeze câte dintre elementele sale ar fi rămas pe aceeași poziție dacă șirul ar fi fost ordonat crescător.

Rezolvare

Soluția este reprezentată de o simulare a enunțului: Considerăm un alt vector în care facem o copie a primului (eventual la citire), îl sortăm pe unul dintre ei și apoi facem verificarea element cu element.

```
cin>>n;
for (i=1;i<=n;i++) {
    cin>>v[i];
    w[i] = v[i];
}
for (i=1;i<n;i++)
    for (j=i+1;j<=n;j++)
        if (v[i] > v[j]) {
            aux = v[i];
            v[i] = v[j];
            v[j] = aux;
        }
for (i=1;i<=n;i++)
    if (v[i] == w[i])
        sol++;
```

5. Se dau `n` numere naturale. Afișați aceste numere ordonate crescător după suma divizorilor. Dacă două numere au aceeași sumă a divizorilor, se va afișa mai întâi cel mai mic (pbinfo.ro, #515).

Rezolvare

Aceasta este o problemă în care avem de sortat după două criterii: unul principal (suma divizorilor) și unul secundar (doar în caz de egalitate după primul, că altfel nici nu ar avea sens să ne punem ptoblema). Procedăm astfel: construim un nou vector `x` și `x[i]` va fi suma divizorilor lui `v[i]`. Altfel spus, la fiecare element din `v` notăm suma în dreptul lui, în x. Când intershimbăm avem grijă să o facem între aceleași poziții în ambii vectori.

```
#include <iostream>
using namespace std;
int i,j,aux,n,d,s;
int v[1001],x[1001];
int main(){
    cin >> n;
    for(i=1; i<=n; i++){
        s=0;
        cin >> v[i];
        for(d=1; d<=v[i]/d; d++)
            if(v[i]%d==0){
                s=s+d;
                if (d != v[i]/d)
                    s += v[i]/d;
            }
        x[i]=s;
    }
    for(i=1; i<n; i++)
        for(j=i+1; j<=n; j++)
            if((x[i]>x[j])||(x[i]==x[j] && v[i]>v[j])){
/// criteril de ordonare: dupa valoarea din sirul cu sumele divizorilor
/// si in caz de egalitate dupa cea din sirul original
                aux=x[i];
```

```
                x[i]=x[j];
                x[j]=aux;
                aux=v[i];
                v[i]=v[j];
                v[j]=aux;
            }
    for(i=1; i<=n; i++)
        cout<<v[i]<<" ";
    return 0;
}
```