

# Algoritmi de căutare și sortare vectori

## Tablouri de memorie

În cazul în care o aplicație trebuie să prelucereze un șir de 20 de numere, să le ordoneze crescător de exemplu, numerele sunt date de intrare pentru algoritm și în primul pas vor fi introduse în memoria calculatorului. La pasul următor numerele vor fi prelucrate și aranjate conform criteriului de ordonare precizat, după care, la al treilea pas, vor fi furnizate utilizatorului sub formă de date de ieșire. În memoria internă ele vor fi reprezentate într-o structură de date de tip tablou de memorie.

Tabloul este o structură de date internă formată dintr-o mulțime ordonată de elemente, ordonarea făcându-se cu un ansamblu de indici. Tabloul de memorie se va identifica după un nume, iar fiecare element al său, după numele tabloului și numărul său de ordine. Fiecare element al structurii se identifică după numele tabloului și poziția din tablou. De la început trebuie să se precizeze câte elemente are tabloul pentru ca sistemul să-i aloce zona de memorie corespunzătoare. În timpul prelucrării tabloului nu i se pot adăuga mai multe elemente decât au fost alocate, pentru că se iese din spațiul de memorie alocat. Tabloul de memorie este o structură de date statică.

Tabloul cu o singură dimensiune este numit **vector**. Declararea unui tablou de memorie de tip vector se face prin instrucțiunea:

**Tip\_data nume [ nr\_elemente ];**

Aici, **tip\_data** precizează tipul elementelor vectorului, **nume** este identificatorul vectorului, iar **nr\_elemente** este o constantă întregă care specifică numărul de elemente ale vectorului. De exemplu, prin:

```
int a[20];
```

se declară un vector cu 20 de elemente de tip întreg.

La declararea unui vector se pot atribui valori inițiale elementelor sale astfel:

**Tip\_data nume[ nr\_elemente ] = { lista\_valori };**

Exemplu:

```
int a[5]={10, 20, 2, 4, 9 };
```

În cazul declarării unui vector inițializat se poate omite numărul elementelor sale, dacă se inițializează toate elementele.

Referirea la un element al vectorului se face prin construcția:

**nume[indice];**

unde **nume** este numele tabloului, iar **indice** este numărul de ordine al elementului în vector. În C++ numerotarea indicilor se face pornind de la 0!

## Căutare secvențială

Căutarea secvențială este unul dintre cei mai simpli algoritmi studiați. El urmărește să verifice apartenența unui element la un șir de elemente de aceeași natură, în speță a unui număr la un șir de numere. Pentru aceasta se parcurge șirul de la un capăt la celălalt și se compară numărul de căutat cu fiecare număr din șir. În cazul în care s-a găsit corespondență

(egalitate), un indicator flag este poziționat. La sfârșitul parcurgerii șirului, indicatorul ne va arăta dacă numărul căutat aparține sau nu șirului.

```
#include<iostream> using
namespace std; void main()
{
    int n,x,i,gasit,a[50];
    cout<<"Dati numarul de elemente ale sirului : "; cin>>n; cout<<"Dati numarul de
    cautat : "; cin>>x;
    cout<<"Dati elementele sirului"<<endl; gasit=0;
    for(i=0; i<n; i++){
        cout<<"a["<<i<<"]= "; cin>>a[i]; if (a[i]==x)
            gasit=1;
    }
    if (gasit==0)
        cout<<"Numarul nu apartine sirului !"<<endl; else
        cout<<"Numarul apartine sirului !"<<endl;
}
```

**Temă.** Să se transforme programul anterior astfel încât să se afișeze pozițiile în care apare elementul căutat.

### **Căutare binară**

Algoritmul de căutare binară oferă performanțe mai bune decât algoritmul de căutare secvențială. El funcționează astfel: se compară numărul de căutat cu elementul aflat la mijlocul șirului (element care se mai numește și pivot). În cazul în care cele două elemente coincid căutarea s-a încheiat cu succes. Dacă numărul de căutat este mai mare decât pivotul, se continuă căutarea în aceeași manieră în subșirul delimitat de pivot și capătul șirului inițial. Dacă numărul de căutat este mai mic decât pivotul se continuă căutarea în aceeași manieră în subșirul delimitat de pivot și începutul șirului inițial.

Algoritmul prezentat se încadrează în clasa algoritmilor elaborați conform tehnicii de programare *Divide et Impera*.

Unul din dezavantajele acestui algoritm este că șirul în care se face căutarea trebuie să fie inițial sortat.

```
#include<iostream> using
namespace std; void main()
{
    int i, n, x, st, dr, mijl, gasit, a[50]; cout<<"Introduceți numarul elementelor
    vectorului : "; cin>>n;
    cout<<"Introduceți elementul de cautat : "; cin>>x;
    for(i=0; i<n; i++) {
        cout<<"a["<<i<<"]= "; cin>>a[i];
    }
    st=0; dr=n-1; gasit=0;
    while (st<=dr && gasit!=1) {
        mijl=(st+dr)/2;
        if (a[mijl]==x)
            gasit=1;
        else
```

```

        if (x<a[mijl])
            dr=mijl-1;
        else
            st=mijl+1;
    }
    if (st>dr)
        cout<<"Nu s-a gasit elementul cautat !"<<endl;
    else
        cout<<"Elementul cautat apartine sirului !"<<endl;
    }

```

**Problemă.** Să se elaboreze un algoritm pentru ștergerea dintr-un vector a elementului cu indicele k și deplasarea elementelor vectorului, începând cu indicele k+1, cu o poziție spre stânga, lungimea logică a vectorului micșorându-se cu 1.

```

#include<iostream>
using namespace std; void
main()
{
    int i,n,k,a[10];
    cout<<"Introduceti dimensiunea vectorului : "; cin>>n; cout<<"Introduceti indicele elementului
    ce se elimina : "; cin>>k; for(i=0;i<n;i++) {
        cout<<"a["<<i<<"]=""; cin>>a[i];
    }
    for(i=k;i<n-1;i++)
        a[i]=a[i+1]; cout<<"Vectorul rezultat
    este : "; for(i=0;i<n-1;i++)
        cout<<a[i]<<" ";
    cout<<endl;
}

```

**Problemă.** Algoritm pentru inserarea unui element într-un vector în poziția indicelui k înseamnă deplasarea elementelor vectorului, începând cu indicele k+1, cu o poziție spre dreapta și apoi atribuirea noii valori elementului cu indicele k. Lungimea logică a vectorului se va mări cu o unitate și va fi n+1, cu condiția ca n+1 să fie mai mic sau cel puțin egal cu lungimea fizică a vectorului, altfel ultimul element al vectorului se pierde.

```

#include<iostream> using
namespace std; void main()
{
    const int dim=10; int
    i,n,k,x,a[dim];
    cout<<"Introduceti dimensiunea vectorului (<=10): "; cin>>n; cout<<"Introduceti pozitia de
    inserare : "; cin>>k; cout<<"Introduceti elementul ce se insereaza : "; cin>>x;
    for(i=0;i<n;i++) { //sunt citite componentele vectorului cout<<"a["<<i<<"]="";
        cin>>a[i];
    }
    cout<<"Vectorul obtinut prin inserare este : "; if(n!=dim) {
        //daca dimensiunea era strict mai mica decat dim, putem obtine
        //un vector de dimensiune mai mare, fara pierderea de elemente for(i=n;i>k-1;i--)
        a[i]=a[i-1];
    }
}

```

```

        a[k-1]=x;
        for(i=0;i<n+1;i++)
            cout<<a[i]<<" ";
    }
    else {
        //prin inserarea elementului x se pierde ultima componenta a
        //vectorului citit de la tastatura
        //prima ramura din if-ul urmatore este pentru situatia in care
        //pozitia de inserare nu este ultima pozitie a vectorului initial if (k<n-1) {
            for(i=n-1;i>k-1;i--)
                a[i]=a[i-1]; a[k-
                1]=x;
            for(i=0;i<n;i++)
                cout<<a[i]<<" ";
        }
        else {
            //cazul in care trebuie inserat un element pe ultima pozitie
            //a vectorului initial a[n-
            1]=x;
            for(i=0;i<n;i++)
                cout<<a[i]<<" ";
        }
    }
    cout<<endl;
}

```

## Metode de sortare pentru vectori

### 1. Sortarea prin selecție directă

Considerăm un vector de elemente comparabile între ele și dorim să le ordonăm crescător. Pentru aceasta comparăm primul element cu toate elementele care urmează după el. Dacă găsim un element mai mic decât primul atunci le interschimbăm pe cele două. Apoi continuăm cu al doilea element al șirului, pe care, de asemenea îl comparăm cu toate elementele care urmează după el și în caz de inversiune interschimbăm cele două elemente. Apoi procedăm la fel cu al treilea element al șirului iar procesul continuă astfel până la penultimul element al șirului care va fi comparat cu ultimul element din șir.

```

#include<iostream> using
namespace std; void main()
{
    int i,j,n,aux,a[50];
    cout<<"Introduceti numarul de elemente din sir : "; cin>>n;
    cout<<"Introduceti numerele"<<endl; for(i=0;i<n;i++)
    {
        cout<<"a["<<i<<"]=""; cin>>a[i];
    }
    //urmeaza algoritmul de sortare
    for(i=0;i<n-1;i++)
        for(j=i+1;j<n;j++)
            if (a[j]<a[i]) {
                aux=a[i]; a[i]=a[j];

```

```

        a[j]=aux;
    }
    //urmeaza afisarea sirului sortat
    cout<<"Sirul sortat este:"<<endl; for(i=0;i<n;i++)
    cout<<a[i]<<" ";
    cout<<endl;
}

```

## 2. Sortarea prin metoda bulelor

Acest algoritm se mai numește și "sortarea prin selecție și interschimbare", "sortarea prin propagare" sau "metoda lentă de sortare" datorită numărului mare de operații care trebuie efectuate. Succesul algoritmului este asigurat de trecerea succesivă prin tablou, până când acesta este sortat, cu specificația că, la fiecare trecere, elementele succesive  $i$  și  $i+1$  pentru care  $a[i] > a[i+1]$ , vor fi interschimbate.

Metoda poate fi îmbunătățită dacă, după fiecare trecere, se va reține ultima poziție din tablou în care a avut loc o interschimbare, iar trecerea următoare se va efectua doar până la acea poziție. În cazul în care la o trecere nu a avut loc nici o interschimbare algoritmul se va încheia.

Pentru o și mai bună optimizare se poate înlocui trecerea prin tablou într-un sens cu trecerea în dublu sens. În acest caz, dacă la două treceri succesive, între două elemente  $i$  și  $i+1$  nu a avut loc o interschimbare, atunci nici la trecerile următoare nu se vor înregistra interschimbări.

Cu toate optimizările aduse acestei sortari, ea se dovedește a fi cu mult mai lentă decât sortarea prin inserție, fiind însă preferată de unii datorită simplității, ce nu ridică probleme de implementare. Pentru valori suficient de mari ale dimensiunii vectorului ce urmează a fi sortat se recomandă utilizarea unor algoritmi ce au o complexitate mai redusă și, prin urmare, oferă un timp de calcul mai mic.

```

#include<iostream> using
namespace std; void main()
{
    int n,i,aux,terminat,a[50];
    cout<<"Introduceti dimensiunea vectorului : "; cin>>n;
    for(i=0;i<=n-1;i++) { cout<<"a["<<i<<"]="";
        cin>>a[i];
    }
    terminat=0; while(!terminat) {
        terminat=1;
        for(i=0;i<=n-1;i++)
            if(a[i]>a[i+1]) {
                aux=a[i];
                a[i]=a[i+1];
                a[i+1]=aux;
                terminat=0;
            }
    }
    cout<<"Vectorul ordonat este : ";
    for(i=0;i<=n-1;i++)
        cout<<a[i]<<" ";
    cout<<endl;
}

```

### 3. Sortarea prin inserție

Sortarea prin inserție se bazează pe aceleași principii ca și cele aplicate de majoritatea jucătorilor de cărți, adică după ridicarea unei cărți de pe masă, aceasta se așează în pachetul din mână la locul potrivit. Cu alte cuvinte, considerăm că avem vectorul sortat a, iar la ivirea unui nou element care se va adăuga vectorului, el va fi pus pe locul potrivit printr-o inserție în interiorul vectorului.

- **Inserția directă.** Este cea mai simplă implementare a algoritmului și se face în felul următor: Se consideră că primele i elemente al vectorului sunt deja sortate. Pentru elementul al (i+1)-lea, din tabloul inițial, se va găsi poziția în care trebuie inserat printre primele i elemente. Toate elementele tabloului de la această poziție și până la i vor fi deplasate cu o poziție mai la dreapta iar poziția eliberată va fi ocupată de elementul i+1.

- **Inserarea rapidă.** Deoarece vectorul destinație este un vector ordonat crescător, căutarea poziției în care va fi inserat a[i] se poate face nu secvențial (ca în cazul inserării directe) ci prin algoritmul de căutare binară. Subvectorul destinație este împărțit în doi subvectori, se examinează relația de ordine dintre elementul de la mijlocul subvectorului și elementul a[j] și se stabilește dacă elementul a[i] va fi inserat în prima jumătate sau în a doua jumătate. Operația de divizare a subvectorului continuă până se găsește poziția în care urmează să fie inserat a[i].

```
#include<iostream> using
namespace std; void main()
{
    int i,j,n,aux,a[50];
    cout<<"Introduceți dimensiunea sirului : "<<endl; cin>>n;
    cout<<"Dati elementele sirului : "<<endl; for(i=0;i<n;i++) {
        cout<<"a["<<i<<"]=" "; cin>>a[i];
    }
    for(j=1;j<n;j++) {
        aux=a[j]
        ; i=j-1;
        while (aux < a[i] && i>=0){ a[i+1]=a[i];
            i=i-1;
        }
        a[i+1]=aux;
    }
    cout<<"Sirul ordonat este: "; for(i=0;i<n;i++)
        cout<<a[i]<<" ";
    cout<<endl;
}
```

### 4. Sortarea prin numărare

Această metodă necesită spațiu suplimentar de memorie. Ea folosește următorii vectori:

- vectorul sursă (vectorul nesortat) a;
- vectorul destinație (vectorul sortat) b;
- vectorul numărător (vectorul de contoare) k.

Elementele vectorului sursă a[i] se copie în vectorul destinație prin inserarea în poziția corespunzătoare, astfel încât, în vectorul destinație să fie respectată relația de ordine. Pentru a se cunoaște poziția în care se va insera fiecare element, se parcurge vectorul sursă și se numără în

vectorul k, pentru fiecare element a[i], câte elemente au valoarea mai mică decât a lui. Fiecare element al vectorului k este un contor pentru elementul a[i]. Valoarea contorului k[i] pentru elementul a[i] reprezintă câte elemente sunt mai mici decât el și arată de fapt poziția în care trebuie copiat în vectorul b.

```
#include<iostream> using
namespace std; void main()
{
    int i,j,n,a[50],b[50],k[50]={0};
    cout<<"Introduceti dimensiunea sirului : "; cin>>n;
    for(i=0;i<n;i++) {
        cout<<"a["<<i<<"]="; cin>>a[i];
    }
    for(i=0;i<n-1;i++)
        for(j=i+1;j<n;j++)
            if(a[i]>a[j])
                k[i]++;
            else k[j]++; for(i=0;i<n;i++)
                b[k[i]]=a[i]; cout<<"Vectorul
ordonat este : "; for(i=0;i<n;i++)
    cout<<b[i]<<" ";
    cout<<endl;
}
```

#### **Algoritm pentru interclasarea a doi vectori (\*)**

Interclasarea a doi vectori înseamnă reuniunea celor doi vectori simultan cu ordonarea elementelor vectorului rezultat.

Presupunem că cei doi vectori sunt sortați (crescător sau descrescător). Prin acest algoritm se parcurg simultan cei doi vectori sursă pentru a se compara un element dintr-un vector cu un element din celălalt vector.

Elementul cu valoarea mai mică, respectiv mai mare, este copiat în vectorul destinație și șters din vectorul sursă. Procesul continuă până când este epuizat unul dintre vectori. Elementele rămase în celălalt vector se adaugă la sfârșitul vectorului destinație.

Se folosesc următoarele variabile de memorie pentru:

- vectori: a și b - vectorii sursă (se presupune că sunt deja sortați crescător sau descrescător) și c - vectorul rezultat (vectorul destinație) care reunește elementele din vectorii a și b, ordonate după același criteriu ca și cele din vectorii sursă;
- lungimile logice ale vectorilor: n - pentru vectorul a (se introduce de la tastatură) și m - pentru vectorul b (se introduce de la tastatură); lungimea vectorului c va fi n+m;
- contorii pentru parcurgerea vectorilor: i - pentru vectorul a, j - pentru vectorul b și k - pentru vectorul c.

Pasii algoritmului (pentu vectori sortati crescator) sunt:

**Pas 1.** Se inițializează variabilele contor, pentru vectorii surse și pentru vectorul rezultat, cu 0: i=0, j=0, k=0.

**Pas 2.** Se compară elementele a[i] și b[j]. Dacă a[i]<b[j] se copie elementul a[i] în vectorul c prin c[k]=a[i] și se șterge elementul a[i] din vectorul a; altfel, se copie elementul b[j] în vectorul c prin c[k]=b[j] și se șterge elementul b[j] din vectorul b. Operația de ștergere nu se execută prin eliminarea efectivă a elementului din vector, ci prin incrementarea contorului: dace se șterge elementul din vectorul a se incrementează contorul i, iar dacă se șterge elementul din vectorul b se incrementează contorul j.

**Pas 3.** Se incrementează contorul k pentru că urmează să se copie un element în acest vector.

**Pas 4.** Se compară contorii vectorilor sursă cu lungimea lor. Dacă fiecare contor este mai mic

sau egal cu lungimea vectorului ( $i < n$  și  $j < m$ ) se revine la pasul 2, în caz contrar se trece la pasul urmator.

**Pas 5.** Se adaugă la sfârșitul vectorului c elementele rămase în celălalt, printr-o simplă operație de copiere: dacă  $i < n$  înseamnă că s-a epuizat vectorul b și se vor adăuga la vectorul c elementele rămase în vectorul a (se execută  $c[k]=a[i]$ ,  $k=k+1$ ,  $i=i+1$  până când i devine egal cu lungimea n a vectorului a); altfel, s-a epuizat vectorul a și se vor adăuga la vectorul c elementele rămase în vectorul b (se execută  $c[k]=b[j]$ ,  $k=k+1$ ,  $j=j+1$  până când j devine egal cu lungimea m a vectorului b).

```
#include<iostream> using
namespace std; void main()
{
    int i,j,k,n,m,a[50],b[50],c[100];
    cout<<"Introduceti dimensiunea primului vector : "; cin>>n; cout<<"Introduceti dimensiunea celui de al
doilea vector : "; cin>>m;
    // se citesc cei doi vectori ordonati deja crescator !
    cout<<endl;
    cout<<"ATENȚIE! Vectorii trebuie introdusi ordonati crescator"<<endl<<endl;
    cout<<"Componentele primului vector"<<endl; for(i=0;i<n;i++) {
        cout<<"a["<<i<<"]=""; cin>>a[i];
    }
    cout<<"Componentele celui de al doilea vector"<<endl; for(j=0;j<m;j++) {
        cout<<"b["<<j<<"]=""; cin>>b[j];
    }
    // incepe interclasarea celor doi vectori i=0; j=0; k=0;
    while(i<n && j<m) {
        if(a[i]<b[j]) {
            c[k]=a[i]; i=i+1;
        }
        else {
            c[k]=b[j]; j=j+1;
        }
        k=k+1;
    }
    if(i<n)
        while(i<n) {
            c[k]=a[i]; k=k+1; i=i+1;
        }
    else
        while(j<m) {
            c[k]=b[j]; k=k+1;
            j=j+1;
        }
    // este afisat vectorul obtinut
    for(k=0;k<n+m;k++)
        cout<<c[k]<<" "; cout<<endl;
}
```