Tehnici de sortare

1. Introducere, definiții

Prin **sortare** se înțelege ordonarea unei mulțimi de obiecte de acelasi tip, pe baza unei componente de tip numeric, ce caracterizează fiecare obiect.

De regulă, obiectele sunt organizate sub formă de fisier sau tablou, în ambele cazuri ele formează un sir. Lungimea sirului este dată de numărul de obiecte (n).

Teoretic, obiectele sunt considerate înregistrări (în C sunt structuri):

$$R_0, R_1, \ldots, R_{n-1};$$

Fiecare înregistrare R_k, are asociată o *"cheie"* C_k, în funcție de care se face sortarea. Pe mulțimea cheilor se defineste o relație de ordine, cu proprietățile:

- oricare ar fi două chei C_i , C_j are loc numai una din relațiile:

$$C_i < C_j,\ C_i = C_j,\ C_i > C_j;$$

- relația de ordine este tranzitivă.

Un sir este **sortat** dacă oricare ar fi $i, j \in \{0, 1, \ldots, n-1\},\ i<j$,avem $C_i \le C_j$. Această definiție implică o sortare în sens crescător pe mulțimea cheilor. În cele ce urmează, algoritmii de sortare vor fi implementați pentru sortare crescătoare, fapt ce nu micsorează gradul de generalitate; dacă dorim ca sirul final să fie sortat descrescător, el va fi citit si înregistrat de la componenta n-1 către 0.

Un algoritm de sortare se numeste **stabil**, dacă înregistrările cu chei egale rămân, în sirul sortat în aceeasi ordine relativă în care se găseau în sirul inițial (nu se schimbă ordinea înregistrărilor egale).

Metodele de sortare se pot clasifica în două mari categorii:

- **metode de sortare internă**, în care înregistrările se află în memoria internă si sunt rapid accesibile;
- **metode de sortare externă**, în care înregistrările se află pe suport extern (disc) iar timpul de acces (mare) face inacceptabil accesul repetat la înregistrări.

Dacă înregistrările ocupă spațiu mare în memorie (multe locații de memorie / înregistr.), se preferă construirea unui tablou de adrese ale înregistrărilor. În loc de muta înregistrări, se mută adrese în tabel, cu o importantă economie de timp. Dacă privim adresele înregistrărilor ca indici, tabloul de adrese conține în final permutarea indicilor, care permite accesul la înregistrări în ordinea crescătoare a cheilor.

Această metodă se numeste **sortare prin tablou de adrese**.

În cazul sortării interne, dacă înregistrările nu ocupă volum mare de memorie, ele vor fi mutate efectiv iar metoda se numeste **sortare de tablouri** .

2. Funcții polimorfice

Aceste funcții sunt construite pentru a se putea aplica oricăror tipuri de date. Evident că prelucrările concrete vor fi diferite de la un tip de date la altul dar aceste aspecte concrete se pot transfera funcțiilor specifice de prelucrare. Transmitând funcției

polimorfice un pointer la funcția de prelucrare si utilizând peste tot tipul de pointer universal void *, se poate realiza usor o funcție polimorfică.

De exemplu, funcția standard, de bibliotecă qsort() implementează algoritmul de sortare internă Quicksort, având prototipul:

```
void qsort(void *tab, size_t n, size_t dim,
           int (*cmp)(const void *, const void *));
```

Se sortează tabloul tab, cu dimensiunea n, fiecare element din tablou având dimensiunea dim iar cmp este un pointer la o funcție care primeste doi pointeri la două elemente ale tabloului, returnând valoare<0, 0 sau valoare>0 după cum este mai mare al doilea, sunt egale sau este mai mare primul element.

Exemple:

1. Pentru sortarea unui tablou de întregi, funcția de comparație trebuie să fie:

```
int cmp_int(const void *a, const void *b)
{
return *((int*)a – *((int*)b;
}
```

Se convertesc pointerii (void *) la (int*), se ia apoi conținutul fiecăruia (deci doi întregi), se face diferența lor, care se returnează ca rezultat. Apelul va fi în acest caz:

```
int tab_int[100];
qsort(tab_int, 100, sizeof(int), cmp_int);
```

2. Pentru sortarea unui tablou de 100 de siruri de caractere, fiecare cu lungimea maximă de 50 de octeți, declarat prin:

```
char a[100][50];
```

în condițiile în care se doreste să se modifica efectiv poziția din memorie a celor 100 de siruri, funcția de comparație va fi:

```
int cmp_sir(const void *a, const void *b)
{
return strcmp(a, b);
}
```

Se utilizaează funcția standard de comparație a sirurilor **strcmp**(); apelul funcției de sortare va fi:

```
qsort(a, 100, 50, cmp_sir);
```

ceea ce pune în evidență că un element al tabloului este de 50 de octeți.

3. Definim o structură si un tablou de structuri:

```
typedef struct {int key; char *sir} STRUCT_1;
STRUCT_1 tab[100];
```

Funcția de comparație pentru căutarea unui element în tablou, după cheia numerică **key,** se defineste acum astfel:

```
int cmp_str_1(const void *key, const void *elem)
{
return *((int *) key) – ((STRUCT_1 *) elem) ->
key;
}
```

Dacă valoarea returnată este zero, elementul căutat a fost găsit. Aici key este adresa unui întreg (cheia numerică) iar elem este adresa unui element al tabloului, deci adresa unei structuri de tipul STRUCT_1, deci este vorba de tipuri diferite. Se converteste pointerul key la (int *) si apoi se ia conținutul, apoi se converteste pointerul elem la (STRUCT_1 *) si se ia câmpul key al structurii indicate de elem. Se întoarece diferența celor doi întregi.

Dacă dorim acum să sortăm tabloul de structuri de mai sus, cu funcția polimorfică qsort(), după cheia key, trebuie modificată funcția de comparație, deoarece se primesc de data asta adresele a două elemente de acelasi tip (elementele tabloului).

```
int cmp_str_2(const void *a, const void *b)
{
return ((STRUCT_1 *)a)->key – ((STRUCT_1 *)b) – >key;
}
```

Pentru generalitate, toți algoritmii de sortare vor fi implementați prin **funcții polimorf ice,** încât să se poată sorta orice tip de tablou, fără modificări în program. Pentru aceasta, vom considera o funcție externă care compară două înregistrări, pe baza adreselor lor din memorie (de tip void * pentru generalitate) si care întoarce o valoare întreagă negativă, zero sau o valoare pozitivă, în funcție de relația dintre cele două înregistrări (<, =, >).

Definim, pentru aceasta, tipul de date:

```
typedef int (*PFCMP) (const void *, const void *) ;
```

adică un pointer la funcția de comparație. Toți algoritmii vor fi implementați după prototipul:

```
void sort(void *v, size_t n, size_t size, PFCMP cmp);
```

în care **v** este adresa de început a tabloului de înregistrări, **n** este numărul de înregistrări, **size** este dimensiunea în octeți a unei înregistrări iar **cmp** este pointerul la funcția care compară două înregistrări.

După apelul funcției de sortare, tabloul **v** va conține înregistrările în ordinea crescătoare a cheilor.

Procedând în acest mod, partea specifică a problemei (structura înregistrării, poziția si natura cheii, tipul relației de ordine) este preluată de funcția de comparație si detasată de algoritmul propriu-zis, ceea ce conferă algoritmului un mare grad de generalitate.

Pentru transferul înregistrărilor dintr-o zonă de memorie în alta, considerăm două funcții externe, **`swap()`** - care schimbă între ele pozițiile din memorie a două înregistrări si **`copy()`** - care copiază o înregistrare dată la o adresă de salvare.

```
void swap(void *v, size_t size, int i, int j);
void copy(void *a, void *b, size_t size);
```

Prima funcție schimbă între ele înregistrările **v[i]** si **v[j]** iar a doua copiază înregistrarea de la adresa **b** (sursă) la adresa **a** (destinație).

Dimensiunea fiecărei înregistrări este de **size** octeți. Calculul adresei se face prin relația **(BYTE *)v + i*size**, unde tipul **BYTE** este definit de:

```
typedef unsigned char BYTE;
```

Metodele de sortare internă se clasifică în trei categorii:
- sortare prin interschimbare;
- sortare prin inserție;
- sortare prin selecție.

Fiecare din aceste categorii cuprinde atât metode elementare, în general neeficiente pentru blocuri mari de date, cât si metode evoluate, de mare eficiență. Metodele elementare au avantajul că sunt mai usor de înțeles si au implementare simplă. Ele se pot folosi cu succes pentru tablouri de dimensiuni relativ reduse (sute, mii de înregistrări).

3. Eficiența algoritmilor de sortare

Este naturală problema comparării algoritmilor. Ce este un algoritm eficient si de ce unul este superior altuia? Analiza eficienței unui algoritm se face, de regulă în trei situații: sir inițial sortat, sir aleator si pentru sir inițial sortat în ordine inversă.

Pentru algoritmii care nu necesită spațiu suplimentar de memorie, comparația se face după numărul de operații efectuate, care în esență sunt: **comparații de chei** si **transferuri de înregistrări.** Transferurile se pot detalia în interschimbări si copieri (o interschimbare necesită 3 copieri).

Evident, numărul de operații depinde de dimensiunea **n** a tabloului care este supus sortării. Natura acestei dependențe este cea care diferențiază algoritmii de sortare. Se defineste o mărime care depinde de **n**, numită ordinul algoritmului **O(n)** .

Dacă numărul de operații se exprimă printr-o funcție polinomială de grad k, se spune că ordinul este De exemplu, dacă numărul de $O(n_k)$. operații este n(n-1)/2, ordinul este . $O(n_2)$

Algoritmii care apar în domeniul prelucrării datelor sunt de ordin polinomial, logaritmic sau combinații ale acestora.

În algoritmii de sortare, apar ordinele si n*log(n). n_2

Un algoritm bun de sortare este O(n*log(n)); algoritmii elementari sunt de ordin $O(n_2)$.

4. Sortare prin interschimbare: Bubblesort

Cea mai simplă este interschimbarea directă sau metoda bulelor. Pentru v[i] fixat, se compară v[j] cu v[j-1] pentru j=n-1, n-2, . . ., i. si se face un schimb reciproc de locuri

în tablou, atunci când nu se respectă condiția de sortare crescătoare v[j]>=v[j-1]. La sfârsitul primei etape, pe locul v[0] ajunge, evident cel mai mic element.

După a două trecere, pe locul v[1] va ajunge , cel mai mic element din cele rămase, s.a.m.d.

Se procedează analog până la ultimul element, care nu se mai compară cu nimic, fiind cel mai mare.

Numărul de interschimbări este n(n-1)/2, în cazul cel mai defavorabil, când sirul inițial este sortat invers (dar noi nu stim !).

O crestere a eficienței se poate realiza astfel: dacă la o trecere prin bucla interioară (de comparări), nu s-a făcut nici o inversare de elemente, înseamnă că sirul este sortat si algoritmul trebuie oprit. Acest lucru se realizează prin introducerea unei variabile logice **sortat** care este inițializată cu 1 si pusă în 0 la orice interschimbare. Se reduce numărul de operații în cazul unui tablou inițial sortat (la n-1) si cresc performanțele în cazul unui sir aleator.

Denumirea metodei provine din faptul că elementele v[i], de valoare mare, se deplasează către poziția lor finală (urcă în sir), pas cu pas, asemănător bulelor de gaz într-un lichid.

```
void bubble_sort(void *v, size_t size, PFCMP cmp)
{
int i, j, sortat;
BYTE *a=(BYTE *) v;
sortat=0;
for(i=1; i<n && !sortat; i++) {
sortat=1;
for(j=n-1; j>=i; j--)
if(cmp(a+(j-1)*size, a+j*size)>0)
{ swap(a, size, j-1, j);
sortat=0;
}
}
}
```

5. Sortarea prin partiționare şi interschimbare: Quicksort

Se poate încadra într-o tehnică mai generală, "*Divide et impera"* (împarte si stăpâneste), deoarece se bazează pe divizarea sirului inițial în siruri din ce în ce mai scurte.

Algoritmul Quicksort, fundamemtat în 1962 de C. A. R. Hoare, este un exemplu de algoritm performant de sortare, fiind de ordinul *n*log(n)* .

Este un algoritm recursiv, usor de implementat în C.

Funcția care realizaeză sortarea primeste ca date de intrare o parte a tabloului ce trebuie sortat, prin adresa de început si doi indici *left* si *right* . Inițial funcția se apelează cu indicii 0 si n-1.

Se alege un element arbitrar al tabloului **v**, numit **pivot**, care se notează cu mark, uzual `mark = v[(left+right)/2]`. Se divide tabloul în două părți în raport cu **mark**: toate elementele mai mici decât **mark** trec în stânga iar cele mai mari decât **mark** trec în dreapta. În acest moment elementul **mark** se află pe poziția finală iar tabloul este partiționat în două tablouri de dimensiuni mai mici.

Dacă notăm cu k indicele pivotului, putem apela aceeasi funcție de sortare cu limitele **left** si **k-1** pentru partea stângă si cu limitele **k+1** si **right** pentru partea dreaptă.

Când se realizează condiția left>=right, algoritmul se încheie. Există mai multe variante de implementare; în exemplul de mai jos, se utilizează doi indici, i si j, care sunt inițializați cu left, respectiv cu right. Cât timp v[i]<mark, incrementăm i, apoi cât timp v[j]>mark, decrementăm j. Acum, dacă i<=j, se face interschimbarea v[i] cu v[j], actualizând similar indicii i si j. Procesul continuă până când i>j.

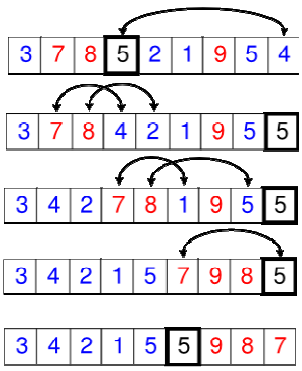
S-a obținut următoarea partiționare:

- toate elementele **v[left],v[left+1],...,v[i-1]** sunt mai mici ca mark;
- toate elementele **v[j+1],v[j+2],...,v[right]** sunt mai mari ca mark;
- toate elementele **v[i],...,v[j]** sunt egale cu mark;

Algoritmul în preudocod este prezentat în continuare:

```
function creare_partitii(array, left, right, pivotIndex)
     ValoarePivot := array[pivotIndex]
     schimba array[pivotIndex] si array[right] // pivotul se muta la sfarsit
     Index := left
     for i de la left la right - 1
         if array[i] ≤ ValoarePivot
             schimba array[i] si array[Index]
             Index := Index + 1
     schimba array[Index] si array[right] // Muta pivotul la locul lui
     return Index
```

Exemplu:



Acum se apelează aceeasi funcție cu indicii **left,j**, respectiv **i, right** (dacă left<j si i<right).

Pentru eficiență, variabilele intens folosite în etapa de partiționare (i si j) sunt declarate în clasa register.

```
void quick_sort(BYTE*v,size_t size,int left,
int right,PFCMPcmp)
{
register int i, j;
BYTE *mark;
```

```
i=left; j=right;
switch(j-i) {
case 0: return;
case 1: if(cmp(v+i*size, v+j*size)>0)
swap(v, size, i, j);
return;
default: break;
}
mark=(BYTE*)malloc(size);
copy(mark, v+((left+right)/2)*size, size);
do {
while(cmp(v+i*size, mark)<0) i++;
while(cmp(v+j*size, mark)>0) j--;
if(i<=j) swap(v, size, i++, j--);
} while (i<=j);
if(left<j) quick_sort(v, size, left, j, cmp);
if(i<right) quick_sort(v, size, i, right, cmp);
free(mark);
}
void Quick(void *v, size_t n, size_t size, PFCMP
cmp)
{
quick_sort(v, size, 0, (int) n-1, cmp);
}
```

Analiza algoritmului Quicksort arată că, în cazul datelor aleatoare, el este de ordin O(nlog(n)).

TEMA:

Utilizând algoritmul BubbleSort realizați un program care citeşte un vector cu *n* elemente de la tastatură, îl împarte în doi vectori ce conţin elementele 0..*i* şi respectiv *i+1..n* din vectorul inițial, sortează cei doi vectori, primul în ordine crescătoare şi al doilea în ordine descrescătoare şi afişează rezultatele (*n, i* se citesc de la tastatură).