# 8. *Analiza Algoritmilor*

- ***Corectitudine***
- ***Eficienţă***
- ***Posibilitate de îmbunătăţire***
- ***Alte calităţi …***

- ***Complexitatea** algoritmilor*
- ***Documentarea** programelor*
- *Inspectare*
- *Testare si depanare*
- *Verificare si validare*
- *Stil in programare*

# *Complexitatea Algoritmilor*

Ne interesează să comparăm între ei mai mulţi algoritmi care rezolvă aceeaşi problemă, despre care ştim că sunt corecţi, precizând ***care dintre aceştia este mai bun***.

Putem compara doi algoritmi în funcţie de :

- cantitatea de **memorie** necesară;
- **viteza** de lucru, deci timpul necesar rezolvării problemei.

Timpul necesar execuţiei unui program depinde de numărul operaţiilor ce trebuiesc executate, care depinde de datele de intrare, deci se schimbă de la o execuţie la alta.

Există însă ***un cel mai rău caz***, pentru acele date de intrare pentru care numărul operaţiilor efectuate este maxim. În acest caz vorbim de ***complexitate în cel mai rău caz***.

De asemenea, putem vorbi de numărul ***mediu de operaţii*** efectuate într-o execuţie. Dacă numărul execuţiilor posibile este finit atunci acest număr mediu este egal cu numărul operaţiilor efectuate în toate execuţiile, împărţit la numărul execuţiilor → ***complexitate medie***.

# Calculul complexitatii

***Complexitatea*** măsoară *eficienţa* unui algoritm din punct de vedere al ***vitezei*** de lucru, deci a ***timpului*** necesar rezolvării unei probleme, care depinde de numărul operaţiilor necesare, iar acesta depinde de volumul datelor de intrare.

Putem determina ***complexitatea***

a) în ***cel mai rău caz*** (numărul operaţiilor este maxim), sau

b) ***medie*** (nr. operaţiilor pentru toate variantele posibile / nr. total de posibilităţi).

Pentru un algoritm care are domeniul $D$ şi datele de intrare $d \in D$, dacă notăm cu

- $n_d$ numărul de operaţii efectuate, şi cu
- $p_d$ probabilitatea de a alege setul d,

vom defini ***complexitatea***

a) în ***cel mai rău caz*** : $C_r = \max_{d \in D} n_d$

b) ***medie :*** $C_m = \sum_{d \in D} p_d \cdot n_d$

În general este suficient să determinăm ordinul de mărime al ***complexităţii*** în funcţie de numărul datelor de intrare ($n$). Se spune că o complexitate $C$ este de ordinul $n^k$ şi se notează aceasta cu $O(n^k)$, dacă există două constante c1, c2 pentru care: $c1 \cdot n^k < C < c2 \cdot n^k$ .

# *Căutare, Sortare, Interclasare*

***Căutarea*** şi ***Sortarea*** sunt două dintre cele mai des întâlnite subprobleme în programare. Ele constituie o parte esenţială din numeroasele procese de prelucrare a datelor. Operaţiile de căutare şi sortare sunt executate frecvent de către oameni în viaţa de zi cu zi, ca de exemplu căutarea unui cuvânt în dicţionar sau căutarea unui număr în cartea de telefon.

***Căutarea*** este mult simplificată dacă datele în care efectuăm această operaţie sunt *sortate* (ordonate, aranjate) într-o anumită ordine (cuvintele în ordine alfabetică, numerele în ordine crescătoare sau descrescătoare).

***Sortarea*** datelor constă în rearanjarea colecţiei de date astfel încât un câmp al elementelor colecţiei să respecte o anumită ordine. De exemplu în cartea de telefon fiecare element (abonat) are un câmp de nume, unul de adresă şi unul pentru numărul de telefon. Colecţia aceasta respectă ordinea alfabetică după câmpul de nume.

Dacă datele pe care dorim să le ordonăm, adică să le sortăm, sunt în memoria internă, atunci procesul de rearanjare a colecţiei îl vom numi *sortare internă*, iar dacă datele se află într-un fişier (colecţie de date de acelaşi fel aflate pe suport extern), atunci procesul îl vom numi *sortare externă.*

Fiecare element al colecţiei de date se numeşte *articol* iar acesta la rândul său este compus din unul sau mai multe componente. O *cheie C* este asociată fiecărui articol şi este de obicei una dintre componente. Spunem că o colecţie de *n* articole este *ordonată crescător* după cheia C dacă $C(i) \leq C(j)$ pentru $1 \leq i < j \leq n$, iar dacă $C(i) \geq C(j)$ atunci este *ordonată descrescător*.

# Căutare

Vom prezenta câteva tehnici elementare de căutare şi vom presupune că datele se află în memoria internă, într-un şir de articole.

Vom căuta un articol după un câmp al acestuia pe care îl vom considera cheia de căutare. În urma procesului de căutare va rezulta poziţia elementului căutat (dacă acesta există).

Notând cu $k_1$, $k_2$, ...., $k_n$ cheile corespunzătoare articolelor şi cu $a$ cheia pe care o căutăm, problema revine la a găsi (dacă există) poziţia $p$ cu proprietatea

$$a = k_p.$$

De obicei articolele sunt păstrate în ordinea crescătoare a cheilor, deci vom presupune

$$k_1 < k_2 < .... < k_n .$$

Uneori este util să aflăm nu numai dacă există un articol cu cheia dorită ci şi să găsim în caz contrar locul în care ar trebui inserat un nou articol având cheia specificată, astfel încât să se păstreze ordinea existentă.

# *Problema Căutării*

*Problema căutării* are următoarea specificaţie:

*Date a,n,(ki, i=1,n);* $\{ n \in N, n \geq 1$ *şi* $k_1 < k_2 < .... < k_n \}$

*Rezultate p;* *{ p = poziţia de inserare }*

unde: $p = \begin{cases} 1 & \text{dacă } a \leq k1 , \\ i & \text{dacă } \exists\, i \in \{2, 3, ... , n\} : k_{i-1} < a \leq k_i , \\ n+1 & \text{dacă } a > k_n . \end{cases}$

Pentru rezolvarea acestei probleme vom descrie mai mulţi subalgoritmi. O primă metodă este ***căutarea secvenţială***:

***Subalgoritmul CautSecv(a,n,K,p) este:***

***Fie*** $p:=1;$

***Câttimp*** $p \leq n$ *şi* $a > k_p$ ***execută*** $p:=p+1$ ***sfcât***

***sf-CautSecv.***

O altă metodă, numită ***căutare binară***, care este mult mai eficientă, utilizează metoda ***divide et impera***. Se determină în ce relaţie se află cheia articolului aflat în mijlocul colecţiei cu cheia de căutare. În urma acestei verificări căutarea se continuă doar într-o jumătate a colecţiei. În acest mod, prin înjumătăţiri succesive se micşorează volumul colecţiei rămase pentru căutare. Căutarea binară se poate realiza prin apelul funcţiei *BinarySearch(a,n,K,1,n)*, descrisă mai jos, folosită în subalgoritmul dat în continuare.

```
Subalgoritmul CautBin(a,n,K,p) Este:
  Dacă a≤k1 Atunci p:=1 Altfel
  Dacă a>kn Atunci p:=n+1 Altfel
          {ki-1<a≤ ki}          p:=BinarySearch(a,K,1,n) Sfdacă  Sfdacă
Sf-CautBin.
Funcţia BinarySearch (a,K,St,Dr) Este:
   Dacă St≤Dr Atunci  m:=(St+Dr) Div 2;
   Dacă a=Km Atunci  BinarySearch:=m Altfel
   Dacă a<Km Atunci  BinarySearch:= BinarySearch(a,K,St,m-1) Altfel
        { a>Km}          BinarySearch:= BinarySearch(a,K,m+1,Dr)
       { St>Dr} Altfel  BinarySearch:=St
Sf-BinarySearch.
```

În funcţia *BinarySearch* descrisă mai sus, variabilele *St* şi *Dr* reprezintă capetele intervalului de căutare, iar *m* reprezintă mijlocul acestui interval. Prin această metodă, într-o colecţie având *n* elemente, rezultatul căutării se poate furniza după cel mult $log_2 n$ comparări. Deci complexitatea în cel mai rău caz este direct proporţională cu $log_2 n$. Fără a insista asupra demonstraţiei, menţionăm că ordinul de mărime al complexităţii medii este acelaşi.

Se observă că funcţia *BinarySearch* se apelează recursiv. Se poate înlătura uşor recursivitatea, aşa cum se poate vedea în următoarea funcţie:

```
Funcţia BinarySearch (a,K,St,Dr) Este:
   Repetă      m:=(St+Dr) Div 2;
      Dacă     a=Km Atunci  BinarySarch:=m  Altfel
      Dacă     a<Km Atunci  Dr:=m-1         Altfel
               { a>Km }     St:=m+1
   PânăCând (St>Dr)  Sau  (a=Km);
   Dacă  St>Dr  Atunci  BinarySarch:=St
Sf-BinarySearch.
```

Complexitatea acestor algoritmi de căutare este dată în tabelul alăturat :

| Algoritmul | Complexitatea | |
|---|---|---|
| | în cel mai rău caz | medie |
| CautSecv | O(n) | O(n) |
| CautBin | $O(log_2 n)$ | $O(log_2 n)$ |

# Sortare internă

Prin sortare internă vom înţelege o rearanjare a unei colecţii aflate în memoria internă astfel încât cheile articolelor să fie ordonate crescător (eventual descrescător). Problema de ***sortare internă*** revine la ordonarea (rearanjarea) cheilor $K=(k_1,k_2,...,k_n)$ astfel încât $k_1 \le k_2 \le ... \le k_n$ .

# Selection Sort

O primă tehnică numită "*Selecţie*" se bazează pe următoarea idee: se determină poziţia elementului cu cheie de valoare minimă (respectiv maximă), după care acesta se va interschimba cu primul element. Acest procedeu se repetă pentru subcolecţia rămasă, până când mai rămâne doar elementul maxim. Deci la fiecare pas $i = 2, 3, ... , n-1$ se va obţine subşirul $k_1 , k_2 , ... , k_i$ corect aranjat (aşa cum aşa cum trebuie să fie în poziţia finală) punând pe poziţia $i$ elementul minim $k_m$ (respectiv maxim) din subşirul $k_i , k_{i+1} , ... , k_n$. După ce şi elementul $k_{n-1}$ este completat corect şirul va fi ordonat (implicit şi ultima poziţie va fi corectă).

Subalgoritmul de ordonare prin ***selecţie*** este următorul:

***Subalgoritmul Selectie*** *(n,K)* ***este:***

***Pentru*** *i:=1, n-1* ***execută*** *{ Completează corect fiecare poziţie i = 1, 2 ,... , n-1 }*

***Fie*** *m:=i;* *{ m = poziţia elementului minim din subşirul $k_i$ , $k_{i+1}$ , ... , $k_n$ }*

***Pentru*** *j:=i+1; n* ***execută***

***Dacă*** *$k_j < k_m$* ***atunci*** *m:=j* ***sfdacă***

***sfpentru***

***Dacă*** *i<m* ***atunci*** *v:=$k_i$; $k_i$:=$k_m$; $k_m$:=v* ***sfdacă*** *{Interschimbă $k_i$ cu $k_m$ }*

***sfpentru***

***sf****-Selectie.*

Se observă că numărul de comparări este :

$$(n-1)+(n-2)+...+2+1 = n(n-1)/2$$

indiferent de natura datelor.

Deci complexitatea medie, dar şi în cel mai rău caz este ***$O(n^2)$***.

# *Insertion Sort*

O altă posibilitate constă în parcurgerea cheilor cu ordonare parţială: întotdeauna porţiunea parcursă rămâne ordonată. Deci cheia la care am ajuns va fi inserată în secvenţa parcursă astfel ca aceasta să-şi păstreze proprietatea de ordonare. Deci la fiecare pas $i = 2, 3, \ldots, n$ se va obţine subşirul $k_1, k_2, \ldots, k_i$ ordonat crescător *inserând* elementul de pe poziţia $i$ în subşirul $k_1, k_2, \ldots, k_{i-1}$ (care este ordonat crescător de la pasul precedent) astfel încât şi acest subşir să rămână ordonat crescător.

Elementul $k_i$ care trebuie pus pe poziţia corectă $p$ se reţine într-o variabilă temporară $v$ pentru a putea muta spre dreapta (cu o poziţie) toate elementele mai mari decât el. Pe locul rămas liber ($p$=poziţia căutată) se va depune v obţinând astfel ordinea dorită în subşirul $k_1,k_2,\ldots,k_n$ adică $k_1 \le k_2 \le \ldots \le k_n$. Evident că la ultimul pas ($i=n$), după ce şi $k_n$ va fi inserat corect se va obţine $k_1 \le k_2 \le \ldots \le k_n$ deci tot şirul va fi ordonat.

Obţinem un al doilea algoritm de sortare, numit ***Inserţie***:

***Subalgoritmul Insertie*** *(n,K)* ***este:***

*Pentru i:=2, n execută*

*Fie v:=$k_i$; p:=i-1;*

*Câttimp p>0 şi $k_p$ >v execută*

$k_{p+1} := k_p$ ;

*p:=p-1*

*sfcât ;*

$k_{p+1}$:=v

*sfpentru*

***sf-Inserţie.***

Algoritmul prezentat mai sus va executa în cel mai nefavorabil caz $n(n-1)/2$ comparări şi mutări, iar în cazul cel mai favorabil, când şirul este ordonat va executa $2(n-1)$ mutări şi $n-1$ comparări, ceea ce înseamnă că acest algoritm are complexitatea $O(n^2)$.

# Bubble Sort

A treia metodă care va fi prezentată, numită "*BubbleSort*", compară două câte două elemente consecutive iar în cazul în care acestea nu se află în relaţia dorită, ele vor fi interschimbate. Procesul de comparare se va încheia în momentul în care toate perechile de elemente consecutive sunt în relaţia de ordine dorită.

***Subalgoritmul BubbleSort** (n,K) este:*
*t:=1;* *{Traversarea t =1 (urmează prima traversare) }*
*Repetă*
*Fie Sortat:=True;* *{Ipoteza "este sortat"}*
*Pentru i:=1, n-t execută*
*Dacă* $k_{i-1} > k_i$ *atunci*
$v := k_i$; $k_i := k_{i+1}$; $k_{i+1} := v$; *{ Interschimbă* $k_i$ *cu* $k_{i+1}$ : $k_i \leq k_{i+1}$ *}*
*Sortat:=False* *{Nu a fost sortat }*
*sfdacă*
*sfpentru;* *t:=t+1*
*până când Sortat* *{Este sortat}*
***sf-BubbleSort.***

Acest algoritm execută în cel mai nefavorabil caz (n-1)+(n-2)+ ... +2+1 = n(n-1)/2 comparări, deci complexitatea lui este ***O(n2)***.

# *Quick Sort*

O metodă mai performantă de ordonare, prezentată în continuare, se numeşte ***QuickSort*** şi se bazează pe tehnica *divide et impera*. Metoda este prezentată sub forma unei proceduri care realizează ordonarea unui subşir precizat prin limita inferioară (*St*) şi limita superioară (*Dr*) a indicilor acestuia.

Apelul procedurii pentru ordonarea întregului şir este : *QuickSort(K,*1*,n)*, unde *n* reprezintă numărul de articole ale colecţiei date este:

... *Cheamă* ***QuickSort(K,*1*,n);*** ...

Procedura ***QuickSort*** *(K,St,Dr)* va realiza ordonarea subşirului $k_{St},k_{St+1},...,k_{Dr}$. Acest subşir va fi rearanjat astfel încât $k_{St}$ să ocupe poziţia lui finală (când şirul este ordonat). Dacă *p* este această poziţie, şirul va fi rearanjat astfel încât următoarea condiţie să fie îndeplinită: $k_i \le k_p \le k_j$, pentru: $St \le i < p < j \le Dr$.

Odată realizat acest lucru (adică toate elementele mai mici decât *kp* se află în stânga şi cele mai mari în dreapta), în continuare va trebui doar să ordonăm subşirul $k_{St},k_{St+1},...,k_{p-1}$ prin apelul recursiv al procedurii ***QuickSort(K,St,p-1)*** şi apoi subşirul $k_{p+1},...,k_{Dr}$ prin apelul ***QuickSort(K,p+1,Dr)***. Ordonarea efectivă a acestui subşir este necesară doar dacă acesta conţine cel puţin două elemente (un subşir având mai puţin de douăa elemente este deja ordonat).

Procedura **QuickSort** este prezentată în continuare :

```
Subalgoritmul QuickSort (K,St,Dr) Este:
  Dacă St<Dr Atunci   p:=Split (K,St,Dr);      { Pune K_St pe poziţia finală p }
          Cheamă QuickSort(K,St,p-1);
          Cheamă QuickSort(K,p+1,Dr)
  Sfdacă
Sf-QuickSort.
Funcţia Split (K,St,Dr) Este:
  Fie i:=St; j:=Dr; v:=k_i;
  Câttimp i<j Execută
      Câttimp i<j şi v≤kj Execută j:=j-1 Sfcât ;  {Elelentele mai mari rămân în dreapta}
      k_i:= k_j;                                  {Elementul mai mic se mută în stânga}
      Câttimp i<j şi k_i≤v Execută i:=i+1 Sfcât ; {Elementele mici rămân în stânga}
      k_j:= k_i                                   {El. mai mare se mută în dreapta}
  Sfcât;
  Fie k_i := v;    Split:=i;      { i = p = j }
Sf-Split.
```

Complexitatea algoritmului prezentat este $O(n^2)$ în cel mai nefavorabil caz, dar complexitatea medie este de ordinul $O(nLog_2n)$.

# *Tree Sort*

Un alt algoritm performant pentru sortare internă este "***Tree Sort***" (sortare arborescentă) care se bazează pe construirea unui *arbore binar ordonat*. *Arborele binar ordonat* are proprietatea că în orice subarbore, subarborele stâng al acestuia conţine articole cu chei mai mici decât cheia conţinută în rădăcină, iar subarborele drept conţine chei mai mari decât rădăcina. Prin parcurgerea în **inordine** a arborelui astfel construit elementele vor fi în ordine crescătoare. Construirea unui astfel de arbore se va realiza prin adăugări succesive ale elementelor de sortat în subarborele stâng sau drept după cum cheia articolului care se adaugă este mai mică respectiv mai mare decât cheia aflată în rădăcina subarborelui în care se efectuează adăugarea.

Complexitatea acestor algoritmi de sortare este dată în tabelul care alăturat:

| ***Algoritmul*** | ***Complexitatea*** | |
|---|---|---|
| | ***în cel mai rău caz*** | ***medie*** |
| ***Selecţie*** | $O(n^2)$ | $O(n^2)$ |
| ***Inserţie*** | $O(n^2)$ | $O(n^2)$ |
| ***Bubblesort*** | $O(n^2)$ | $O(n^2)$ |
| ***QuickSort*** | $O(n^2)$ | $O(n*\log_2 n)$ |

# *Counting Sort*

Algoritmul ***Sortare prin Numărare*** realizează ordonarea unui şir $x_1, x_2, ..., x_n$ având termenii dintr-o mulţime finită {0, 1, ... ,M} ( $0 \le x_i \le M, \ \forall$ i=1,*n* ). Acest algoritm păstrează ordinea dată pentru elementele nedistincte (Dacă xi reprezintă chei de ordonare neunice, de exemplu notele unor studenti $\in$ {1, 2, ... , 10} ).

Algoritmul calculează poziţia pe care se va afla fiecare element prin numărarea valorilor mai mici sau egale decât acesta.

Subalgoritmul este următorul:

***Subalgoritmul*** *Sort_Num (X,n)* ***este****:*

***Pentru*** *j:=0, m* ***Execută*** *Pj := 0* ***Sf_Pentru;*** *{ Iniţializează Pj = frecvenţa valorii j }*

***Pentru*** *i:=0, n* ***Execută*** $Px_i := Px_i + 1$ ***Sf_Pentru;*** *{ Det.* $Px_i$ *= frecv. Val.* $x_i$ *(*)}*

***Pentru*** *j:=1, m* ***Execută*** $P_j := P_j + P_{j-1}$ ***Sf_Pentru;*** *{ Recalc. Pj=frecv. Val.* $\le j$ *(**)}*

***Pentru*** *i:= n,*1,-1 ***Execută*** *{ Se trav. invers pentru a păstra ord. elem. egale}*

$y_{Pxi} := x_i$ ; *{ Depune elementul xi în şirul Y pe poziţia corectă Pxi }*

$Px_i := Px_i$ - 1 *{ Pregăteşte următoarea poziţie pentru un element egal cu xi }*

***Sf_Pentru;***

*X:=Y { Depune în şirul X şirul ordonat din vectorul temporar Y }*

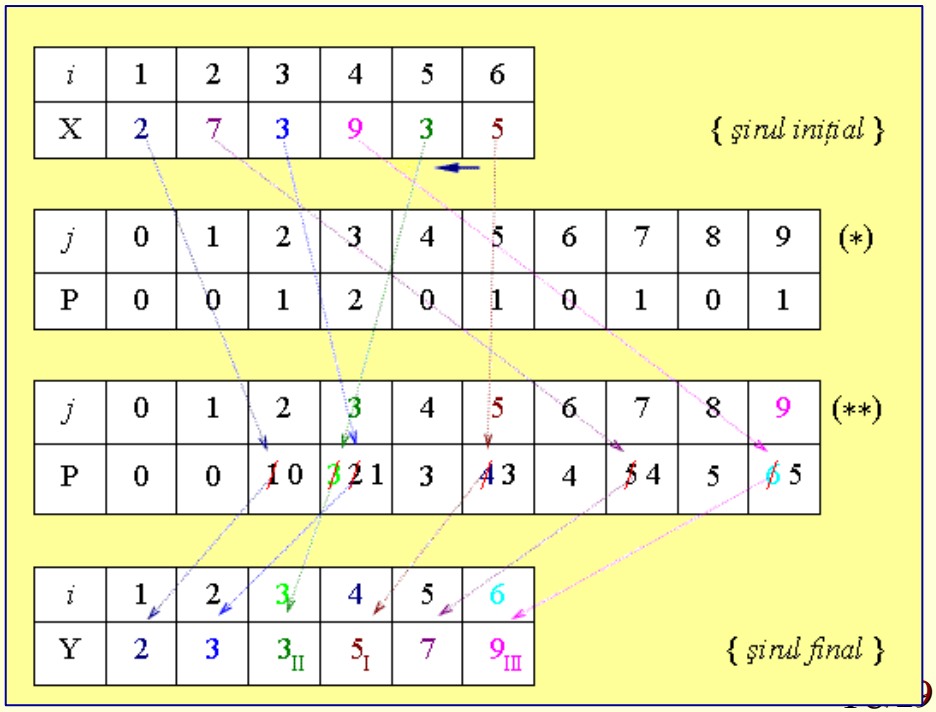
***Sf-*** *Sort_Num.*

Calitatea acestui algoritm constă în **complexitatea** lui: $O(Max(m,n))$, deci pentru noi este important volumul de date, deci practic complexitatea este $O(n)$. Evident că nu putem ordona astfel orice şir ci doar şiruri pentru care mulţimea valorilor cheilor este finită. De exemplu, pentru o ordonare după medii vom avea limita M=1000 (media ***a.bc*** se va număra la ***abc***).

Pentru a renunţa la şirul temporar Y se va interschimba fiecare element cu elementul de pe poziţia corectă (finală). Elementul $x_i$ este completat corect dacă $i=P_{x_i}$ ($i$ este poziţia iniţială iar $P_{x_i}$ este poziţia finală, egală cu numărul de elemente mai mici sau egale decât $x_i$ (**)).

În tabelul următor se poate vedea modificarea directă a vectorului X (prin interschimbări succesive până se ajunge la forma finală ordonată):

| *i* | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| X | 2 | 7 | 3 | 9 | 3 | 5 |
| | 2 | 7 | 3 | 5 | 3 | 9 |
| | 2 | 7 | 3 | 5 | 3 | 9 |
| | 2 | 3 | 3 | 5 | 7 | 9 |
| | 2 | 3 | 3 | 5 | 7 | 9 |



# *Merge Sort*

Un ultim algoritm care va fi prezentat se numeşte "*Merge Sort*" (sortare prin interclasare) şi se bazează pe tehnica "divide et impera". Şirul ce urmează a fi ordonat se împarte în două subşiruri care se ordonează, după care acestea se vor interclasa obţinându-se întregul şir ordonat. Fiecare subşir se va ordona tot prin despărţirea lui în două subşiruri urmată de interclasare şi aşa mai departe până când ordonarea unui subşir se poate rezolva elementar fără a mai fi necesară despărţirea lui în alte două subşiruri (lungimea subşirului este cel mult 2).

Algoritmul corespunzător este prezentat în paragraful următor apelând o procedură recursivă care ordonează un subşir precizând limitele acestuia astfel: :

```
Subalgoritmul MergeSort (A,St,Dr) este:  {Sort. prin intercl. a elem. A_St,A_St+1,...,A_Dr}
  Dacă St < Dr atunci
    Fie m:=(St+Dr) Div 2;                          { m:=Split(A,St,Dr)}
    Cheamă MergeSort (A,St,m);                      {OrdoneazăASt,...,Am }
    Cheamă MergeSort (A,m+1,Dr);                    {OrdoneazăAm+1,...,ADr }
    Cheamă Interclasare (A,St,m, A,m+1,Dr, A,St,Dr);    {Combin rezultatele}
  sfdacă
sf- MergeSort .
```

# *Interclasare*

Fiind date două colecţii de date, ordonate crescător (sau descrescător) după o cheie, se cere să se obţină o colecţie care să fie de asemenea ordonată crescător (respectiv descrescător) după aceeaşi cheie şi care să fie formată din articolele colecţiilor date. Acest lucru se poate obţine direct (fără o sortare a colecţiei finale) prin parcurgerea secvenţială a celor două colecţii, simultan cu generarea colecţiei cerute. Prin compararea a două elemente din listele de intrare se va decide care element va fi adăugat în lista de ieşire. Deci ne interesează un algoritm de rezolvare a următoarei probleme:

*Se dă un şir X ordonat crescător* ($x_1 \le x_2 \le ... \le x_m$) *şi un şir Y de asemenea ordonat crescător* ($y_1 \le y_2 \le ... \le x_n$).

*Se cere şirul Z ordonat crescător* ($z_1 \le z_2 \le .... \le z_{m+n}$) *format din termenii şirului X şi Y.*

O soluţie posibilă ar fi depunerea componentelor vectorului *X* şi a componentelor vectorului *Y* în vectorul *Z*, urmată de ordonarea vectorului *Z* obţinând rezultatul dorit. Acest algoritm este ineficient deoarece la o singură traversare a vectorilor *X* şi *Y* se poate obţine vectorul *Z* aşa cum se poate vedea în următorul algoritm (*de interclasare*).

***Subalgoritmul*** *Interclasare(m,X,n,Y,k,Z)* ***este***: *{X are m componente ord. crescător}*
*{La fel Y cu n componente. }*

***Fie*** *i:=1; j:=1; k:=0;* *{ Cele m+n valori se depun în Z ordonate crescător }*

***Câttimp*** *(i≤m) şi (j≤n)* ***execută*** *{Există comp. în X şi în Y }*

***Dacă*** $x_i \le y_j$ ***Atunci Cheamă*** *Pune(i,*$x_i$*,k,Z)* *{* $x_i \rightarrow z_k$ *}*

***Altfel Cheamă*** *Pune(j,*$y_j$*,k,Z)* *{* $y_j \rightarrow z_k$ *}*

***Sfdacă***

***Sfcât;***

***Câttimp*** *(i≤m)* ***Execută Cheamă*** *Pune(i,xi,k,Z)* ***Sfcât;*** *{Mai există comp. în X }*

***Câttimp*** *(j≤n)* ***Execută Cheamă*** *Pune(j,yj,k,Z)* ***Sfcât;*** *{Mai există comp. în Y }*

***Sf****-Interclasare.*

Aici s-a folosit subalgoritmul ***Pune(ind,val,k,Z)*** care pune în vectorul Z valoarea v şi măreşte indicele i cu 1, subalgortim dat în continuare.

***Subalgoritmul Pune(i,v,k,Z) este***: *{Adaugă val în vectorul Z cu k componente }*

*k:=k+1;* *{ şi măreşte ind cu 1 }*

$z_k$*:=v;*

*i:=i+1*

***Sf****-Pune.*

Subalgoritmul de *sortare* bazat pe *interclasare* se poate vedea în continuare .

```
Subalgoritmul Sortare_Prin_Interclasare(n,K) este:  {Sortare prin interclasare}
    Cheamă MergeSort (K,St,Dr);
sf-SortInter.

Subalgoritmul MergeSort (A,St,Dr) este:   {Sortare prin interclasare a elementelor A_St,A_St+1,...,A_Dr}
  Dacă St < Dr atunci
    Fie m:=(St+Dr) Div 2;                                  { m:=Split(A,St,Dr)}
    Cheamă MergeSort (A,St,m);                             {OrdoneazăASt,...,Am }
    Cheamă MergeSort (A,m+1,Dr);                           {OrdoneazăAm+1,...,ADr }
    Cheamă Interclasare (A,St,m, A,m+1,Dr, A,St,Dr);       {Combin rezultatele}
  sfdacă
sf- MergeSort.

Subalgoritmul Interclasare (X,i,m, Y,j,n, Z,p,k) este:     { Interclasare }
 Fie k:=p-1;
 Câttimp (i≤m) şi (j≤n) execută                            {Există comp. în X şi în Y}
    Dacă xi≤yj      Atunci Cheamă Pune(i,xi,k,Z)           { xi → zk }
                    Altfel  Cheamă Pune(j,yj,k,Z)          { yj → zk }
    Sfdacă
 Sfcât;
 Câttimp (i≤m) Execută  Cheamă Pune(i,xi,k,Z) Sfcât        {Mai există componente în X }
 Câttimp (j≤n) Execută  Cheamă Pune(j,yj,k,Z) Sfcât        {Mai există componente în Y }
sf-Inter .
```

# Sortare externă

O problemă cu care ne confruntăm adesea este sortarea unei colecţii de date aflate pe un suport extern, de volum relativ mare faţă de memoria internă disponibilă. În această secţiune o astfel de colecţie de date o vom numi *fişier*. În acest caz nu este posibil transferul întregii colecţii în memoria internă pentru a fi ordonată şi apoi din nou transferul pe suport extern. Dacă datele ce urmează a fi sortate ocupă un volum de $n$ ori mai mare decât spaţiul de memorie internă de care dispunem, atunci colecţia se va împărţi în $n$ subcolecţii ce vor fi transferate succesiv în memoria internă, se vor sorta pe rând şi vor fi stocate din nou pe suportul extern sortate. Din acest moment prin operaţii de interclasare două câte două se pot obţine colecţii de dimensiuni superioare până se obţine toată colecţia ordonată.

La aceste interclasări, pentru a efectua un număr cât mai mic de operaţii de transfer se recomandă interclasarea colecţiilor de dimensiuni minime, apoi din datele obţinute din nou vor fi alese două colecţii de dimensiuni minime şi aşa mai departe până se obţine o singură colecţie care va fi colecţia cerută, adică sortată.

Sortarea externă presupune parcurgerea a două etape importante:

a) *Împărţirea* fişierului de sortat $F$, în $n$ fişiere $H_1,H_2,...,H_n$ şi sortarea internă a acestora;

b) *Interclasarea* fişierelor de dimensiuni minime rezultând un alt fişier ordonat, operaţie care se repetă de $n$-1 ori pentru a ajunge la fişierul dorit $G$.

# *Documentarea programelor*

În paralel cu elaborarea programului trebuie elaborată şi o documentaţie. Aceasta va conţine toate deciziile luate în crearea programului. Documentarea este activitatea de prezentare a programului celor care vor fi interesaţi să obţină informaţii despre el. Aceştia sunt în primul rând persoanele care au realizat programul, apoi persoanele care-l vor folosi şi persoanele solicitate să facă întreţinerea acestuia. Rezultă de aici că avem două tipuri de documentaţie:

- **documentaţie de *realizare*,**
- **documentaţie de *utilizare*** .

**Documentaţia de *realizare*** trebuie să redea în cele mai mici detalii modul în care a fost realizat programul. Adeseori se întâlnesc programe fără nici o altă documentaţie în afara textului propriu-zis al programului. În graba de a termina cât mai repede, programul nu este însoţit de nici o documentaţie şi frecvent nu sunt folosite nici comentarii în textul programului.

Sigur că însăşi textul programului constituie o autodocumentare. Iar comentariile prezente în program dau explicaţii suplimentare despre program. Este însă necesară o documentaţie completă, scrisă, care va conţine documentaţia de realizare, ce constă din:

- *enunţul iniţial* al problemei;
- *specificaţia*;
- *documentaţia de proiectare* (metoda de rezolvare aleasă şi proiectarea algoritmilor folosiţi. Pentru fiecare algoritm va fi prezentată subproblema corespunzătoare, cu specificaţia ei şi rolul fiecărei variabile);
- *documentaţia de programare*, care va include textul programului;
- *datele de test* folosite;
- *modificări făcute în timpul întreţinerii* programului,

şi documentaţia privind exploatarea programului.

O primă documentaţie a oricărui program este textul sursă propriu-zis. Este bine ca acest text să poată fi citit cât mai uşor, iar comentariile dau informaţii suplimentare despre program, constituie o autodocumentare a programului. Referitor la ***autodocumentare***, folosirea comentariilor, alegerea cu grijă a denumirii variabilelor, cât şi claritatea textului, obţinută prin indentare şi grijă asupra structurii programului, este utilă nu numai pe timpul elaborării programului, dar mai ales pe timpul întreţinerii şi modificărilor ulterioare.

- Denumirea variabilei să fie astfel aleasă încât să reflecte semnificaţia ei.
- Se recomandă şi scrierea cuvintelor cheie cu litere mari pentru a le diferenţia de celelalte denumiri din program.
- Comentariile vor fi prezente:

- în capul programului, pentru a prezenta titlul şi scopul programului, perioada realizării lui şi numele programatorului;
- în definiţii, pentru a descrie semnificaţia notaţiilor folosite (a variabilelor, constantelor, subalgoritmilor, etc);
- în dreapta unor instrucţiuni, pentru a descrie rolul acestora, sau cazul în care se atinge acea instrucţiune;
- între părţile unui modul mai lung, pentru a explica rolul fiecărei părţi.

- Comentariile sunt recomandate, fiind un mijloc de autodocumentare a programului sursă.

Este însă nevoie şi de o documentaţie însoţitoare scrisă, care constituie documentarea propriu-zisă a programului. Aceasta trebuie să redea toate deciziile făcute în timpul proiectării, să prezinte diagrama de structură a întregului produs şi fiecare parte separat. Pentru fiecare modul documentaţia va conţine: numele acestuia, datele de intrare, datele de ieşire, funcţia realizată de modulul respectiv, variabilele folosite şi semnificaţia lor, algoritmul propriu-zis.

## documentaţia de *Utilizare*

Este necesară şi o **documentaţie de *folosire*** a produsului realizat. Beneficiarul nu este interesat de modul în care a fost realizat programul ci de modul în care îl poate folosi.

Menţionăm că cele mai recente produse realizate de firmele consacrate au, pe lângă documentaţia scrisă, şi o autodocumentaţie (funcţii HELP).

O documentare completă a unui program poate fi utilă nu numai pentru folosirea şi întreţinerea programului. Componente ale unui produs existent pot fi utile şi în realizarea altor produse. Este însă necesar să se înţeleagă cât mai uşor ce funcţii realizează aceste componente şi cu ce performanţe. Folosirea acestor componente existente, testate şi documentate, duce evident la creşterea productivităţii în realizarea noului produs.

## Tema:

*Ce linie şi ce coloană se pot tăia dintr-o matrice data, astfel încât suma elementelor rămase să fie maximă?*

**Exemplu:**

2 6 6 5
5 5 1 3
4 9 7 8
2 2 6 2

**O soluţie** (2,1):

Suma Max.= 51

0 6 6 5
0 0 0 0
0 9 7 8
0 2 6 2

Contraexemplu:

```
 2  6  6  5     19          17
 5  5  1  3     14           9
 4  9  7  8     28          24
 2  2  6  2     12   -->    --
11 20 14 16    min_lc   max=50!
 |
Suma max.= 50!
 0  6  6  5
 0  5  1  3
 0  9  7  8
 0  0  0  0
```

Matricea A:

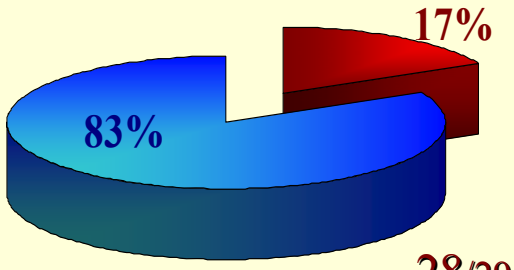
7 5 5 8
9 1 1 4
4 5 2 1
1 3 7 1

Suma Max.= 44

7 5 0 8
9 1 0 4
4 5 0 1
0 0 0 0

Suma max.= 43

7 0 5 8
9 0 1 4
0 0 0 0
1 0 7 1



# …Tema:

*Ce linie şi ce coloană se pot tăia dintr-o matrice data, astfel încât suma elementelor rămase să fie maximă?*

```
void  main ()
{                          clrscr();     randomize();
  Mat a,b,c; int n,Sa,Sb, Nt=10000, Ce=0;
  for (int t=1; t<=Nt; t++) {
     Gen(a,n);
     Sa=Alb(a,b,n);
     Sb=L_c(a,c,n);
     if (Sa!=Sb) { Ce++;
         g << endl <<" Matricea A:"<<    endl;  Tipf(a,n);
         g <<        " Suma Max.= "<<Sa<<endl;  Tipf(b,n);
         g <<        " Suma max.= "<<Sb<<endl;  Tipf(c,n);
     }
  }
  if (!Ce)  cout << " Ok dupa " << Nt << " teste." << endl;
      else  cout << " See ContraEx.Txt "<<Ce<<" din "<<Nt<<endl;
  g.close();
  getche();
}
```

```
typedef int Mat[10][10];
void Gen(Mat a, int& n)
{
  n=random(4)+4;
  for (int i=1; i<=n; i++)
  for (int j=1; j<=n; j++)
a[i][j]=random(9)+1;
}
```

```
int Alb (Mat a, Mat b,  int n)
{
  int s,sij, sm=0,im=0,jm=0; int X[10],Y[10];
  s=Sum(a,n); SumCol(a,n,X); SumLin(a,n,Y);
  for (int i=1; i<=n; i++)
  for (int j=1; j<=n; j++) {
       sij=s-Y[i]-X[j]+a[i][j];
       if (sij>sm) { sm=sij; im=i; jm=j; }
  }
  Mut(a,b,n);
  Lin(b,n,im);Col(b,n,jm);
  return sm;
}
```

```
int L_c (Mat a, Mat c,  int n)
{
  int im, jm, Sl[10], Sc[10];
  Mut(a,c,n);
  SumLin(c,n,Sl);   Lin(c,n,Pm(Sl,n));
  SumCol(c,n,Sc);   Col(c,n,Pm(Sc,n));
  return Sum(c,n);
}
```

```
ofstream g("ContraEx.Txt");
void Tipf(Mat a, int n)
{
     for (int i=1; i<=n; i++) { g << endl;
     for (int j=1; j<=n; j++)
       g << setw(3)<<setfill(' ')<<a[i][j];
     }    g << endl << endl;
}
```

```
int Pm(int* X, int n)
{
      int m=1;
  for (int i=2; i<=n; i++) if (X[i]<X[m]) m=i;
  return m;
}
```

```
int Sum(Mat a, int n)
{
  int s=0;
  for (int i=1; i<=n; i++)
  for (int j=1; j<=n; j++) s+=a[i][j];
  return s;
}
```

```
void Lin(Mat a, int n, int i)
{
    for (int j=1; j<=n; j++) a[i][j]=0;
}
```

```
void SumLin(Mat a, int n, int* Y)
{
 for (int i=1; i<=n; i++) {
       Y[i]=0;
       for (int j=1; j<=n; j++) Y[i]+=a[i][j];
}}
```

```
void Col(Mat a, int n, int j)
{
    for (int i=1; i<=n; i++) a[i][j]=0;
}
```

```
void SumCol(Mat a, int n, int* X)
{
  for (int j=1; j<=n; j++) {
       X[j]=0;
       for (int i=1; i<=n; i++) X[j]+=a[i][j];
}}
```

```
void Mut(Mat a, Mat b, int n)
{
  for (int i=1; i<=n; i++)
  for (int j=1; j<=n; j++) b[i][j]=a[i][j];
}
```

… C7 / 16.11.2012