Ordonări şi căutări

Capitolul

9

- **Algoritmi de ordonare**
- **Algoritmi de căutare**
- **Interclasarea a două tablouri unidimensionale**
- **Implementări sugerate**
- **Probleme propuse**
- **Soluțiile problemelor**

9.1. Ordonări

Fie $A = (a_1, a_2,...,a_n)$ un tablou unidimensional cu n elemente. A *ordona* (sorta) tabloul înseamnă a ordona crescător sau descrescător elementele tabloului.

În cartea lui *D. Knuth* „Arta programării calculatoarelor", vol. III (*Căutări şi ordonări*) sunt prezentate 33 de metode de ordonare. Dintre acestea vom învăța doar câteva.

9.1.1. Metoda bulelor

Să presupunem că dorim o ordonare crescătoare a şirului. Algoritmul constă în parcurgerea tabloului A de mai multe ori, până când devine ordonat. La fiecare pas se compară două elemente alăturate. Dacă $a_i > a_{i+1}$, ($i = 1, 2, ..., n - 1$), atunci cele două valori se interschimbă între ele. Controlul acțiunii repetitive este dat de variabila booleană *ok*, care la fiecare reluare a algoritmului primește valoarea inițială *adevărat*, care se schimbă în *fals* dacă s-a efectuat o interschimbare de două elemente alăturate. În momentul în care tabloul A s-a parcurs fără să se mai efectueze nici o schimbare, *ok* rămâne cu valoarea inițială *adevărat* şi algoritmul se termină, deoarece tabloul este ordonat.

Interschimbarea a două elemente se realizează prin intermediul variabilei auxiliare *aux* care are acelaşi tip ca şi elementele tabloului.

```
Subalgoritm Metoda_bulelor(a,n):
  repetă
    ok ← adevărat
    pentru i=1,n-1 execută:
      dacă a[i] > a[i+1] atunci
        ok ← fals
        aux ← a[i]
```

```
        a[i] ← a[i+1]
        a[i+1] ← aux
      sfârşit dacă
    sfârşit pentru
  până când ok
sfârşit subalgoritm
```

Fie de exemplu tabloul *A* având 5 elemente numere reale: 0.0, 1.1, 1.0, 1.2 şi 0.08.

Prima parcurgere a tabloului (*ok* este iniţializat cu *adevărat*):

$a_1 = 0.0$	$a_2 = 1.1$	$a_3 = 1.0$	$a_4 = 1.2$	$a_5 = 0.08$	*ok*
0.0	1.0 ↔	↔ 1.1	1.2	0.08	*fals*
0.0	1.0	1.1	0.08 ↔	↔ 1.2	*fals*

0.0 < 1.1, rămân neschimbate, 1.1 > 1.0, le interschimbăm. Deoarece 1.1 < 1.2, avansăm şi constatăm că 1.2 > 0.0.8, deci din nou avem interschimbare. În consecinţă, la ieşire din structura **pentru** *ok* este *fals*. Observăm că 1.2 a ajuns pe locul lui definitiv. Urmează a doua parcurgere a tabloului (*ok* primeşte din nou valoarea *adevărat*).

$a_1 = 0.0$	$a_2 = 1.0$	$a_3 = 1.1$	$a_4 = 0.08$	$a_5 = 1.2$	*ok*
0.0	1.0	0.08 ↔	↔ 1.1	1.2	*fals*

Am avut interschimbare şi de data aceasta, deci ieşim cu *ok* = *fals*. La acest pas 1.1 a ajuns pe locul său definitiv. A treia parcurgere a tabloului începe cu reiniţializarea lui *ok* cu valoarea *adevărat*.

$a_1 = 0.0$	$a_2 = 1.0$	$a_3 = 0.08$	$a_4 = 1.1$	$a_5 = 1.2$	*ok*
0.0	0.08 ↔	↔ 1.0	1.1	1.2	*fals*

Am interschimbat 0.08 cu 1.0, cel din urmă astfel a ajuns pe locul său în şirul ordonat. A patra parcurgere a tabloului se finalizează cu valoarea *ok* = *adevărat*, deoarece nu am efectuat nici o interschimbare, ceea ce înseamnă că procesul de ordonare s-a încheiat.

$a_1 = 0.0$	$a_2 = 0.08$	$a_3 = 1.0$	$a_4 = 1.1$	$a_5 = 1.2$	*ok*
0.0	0.08	1.0	1.1	1.2	*adevărat*

Observaţia cu privire la faptul că la fiecare parcurgere a ajuns cel puţin un element pe locul său definitiv în şirul ordonat poate fi fructificată, deoarece constatăm că astfel, la următorul pas nu mai sunt necesare verificările în care intervine acest element şi cele care se află după el în şir. Rezultă că la fiecare parcurgere am putea micşora cu 1 numărul elementelor verificate. Dar este posibil ca la o parcurgere să ajungă mai multe

elemente în locul lor definitiv. Rezultă că vom ține minte indicele ultimului element care a intervenit în interschimbare şi verificările le vom efectua doar până la acest element. Similar, se poate restrânge şi indicele primului element care a intervenit în interschimbări. Astfel, ajungem la următorul subalgoritm îmbunătățit ca performanță:

```
Subalgoritm Bule_Îmbunătățit(n,a):
  stvechi ← 1  { marginea din stânga a subşirului care urmează să fie verificat }
  drvechi ← n - 1                                  { marginea din dreapta }
  repetă
    ok ← adevărat                          { presupunem că şirul este ordonat }
    dr ← 0                               { indicele ultimului element interschimbat }
                                         { verificăm elementele din subşirul curent }
    pentru i=stvechi,drvechi execută:
      dacă a[i] > a[i+1] atunci
        Interschimb(a[i],a[i+1])
        ok ← fals
        dacă i > dr atunci          { am interschimbat elementul de indice i }
          dr ← i                    { indicele ultimului element interschimbat }
        sfârşit dacă
      sfârşit dacă
    sfârşit pentru
    dacă nu ok atunci                        { a avut loc o interschimbare }
      drvechi ← dr                       { actualizăm marginea din dreapta }
      st ← n                                          { marginea din stânga }
      { verificăm elementele din subşirul curent avansând din dreapta spre stânga }
      pentru i=drvechi,stvechi execută:
        dacă a[i] > a[i+1] atunci
          Interschimb(a[i],a[i+1])
          ok ← fals
          dacă i < st atunci        { am interschimbat elementul de indice i }
            st ← i                  { indicele primului element interschimbat }
          sfârşit dacă
        sfârşit dacă
      sfârşit pentru
      stvechi ← st                       { actualizăm marginea din stânga }
    sfârşit dacă
  până când ok
sfârşit subalgoritm
```

Metoda bulelor nu este cea mai performantă modalitate de a ordona un şir cu multe elemente, dar în cazul şirurilor „aproape ordonate”, cu optimizările de mai sus, poate deveni mai eficientă decât alte metode.

9.1.2. Sortare stabilind poziția definitivă prin numărare

Această metodă constă în construirea unui nou tablou *B* care are aceeaşi dimensiune ca şi tabloul *A* în care depunem elementele din *A*, ordonate crescător.

Vom analiza fiecare element şi îl vom compara cu fiecare alt element din şir pentru a putea reține în variabila *k* numărul elementelor care sunt mai mici decât elementul considerat. Astfel, vom afla poziția pe care trebuie să-l punem pe acesta în şirul *B*. Dacă în problemă avem nevoie de şirul ordonat tot în tabloul *A*, vom copia în *A* întreg tabloul *B*.

```
Subalgoritm Numărare(n,a):
  pentru i=1,n execută:
    k ← 0
    pentru j=1,n execută:
      dacă (a[i] ≥ a[j]) şi (i ≠ j) atunci
        k ← k + 1  { numărăm câte elemente sunt mai mici sau egale cu a[i] }
      sfârşit dacă
    sfârşit pentru
    b[k+1] ← a[i]            { pe a[i] îl punem pe următoarea poziție din b }
  sfârşit pentru
  a ← b                                  { copiem peste şirul a întreg şirul b }
sfârşit subalgoritm
```

Exemplu

Fie tabloul *A* cu 4 elemente: 7, 2, 3, –1.

i	*j*	*Relația*	*k*	b_{k+1}
1	1	$i = j$	0	
1	2	$7 > 2$	1	
1	3	$7 > 3$	2	
1	4	$7 > -1$	3	$b_4 \leftarrow 7$
2	1	$2 < 7$	0	
2	2	$i = j$	0	
2	3	$2 < 3$	0	
2	4	$2 > -1$	1	$b_2 \leftarrow 2$
3	1	$3 < 7$	0	
3	2	$3 > 2$	1	
3	3	$i = j$	1	
3	4	$3 > -1$	2	$b_3 \leftarrow 3$
4	1	$-1 < 7$	0	
4	2	$-1 < 2$	0	
4	3	$-1 < 3$	0	
4	4	$i = j$	0	$b_1 \leftarrow -1$

9.1.3. Sortare prin selecție directă

Metoda precedentă are dezavantajul că necesită de două ori mai multă memorie decât tabloul *A*. Dacă dorim să evităm această risipă, putem aplica metoda de ordonare prin selectarea unui element şi plasarea lui pe poziția sa finală.

De exemplu, în caz de ordonare crescătoare, pornind de la primul element se caută valoarea minimă din tablou. Aceasta se aşează pe prima poziție printr-o interschimbare între elementul de pe prima poziție şi elementul minim. Apoi, se reia algoritmul, pornind de la a doua poziție şi se caută minimul între elementele a_2, ..., a_n. Acesta se interschimbă cu al doilea dacă este cazul. Procedeul se continuă până la ultimul element.

Pseudocodul algoritmului de sortare prin selecția minimului este:

```
Subalgoritm Selecţie(n,a):
  pentru i=1,n-1 execută:
    min ← a[i]
    pentru j=i+1,n execută:
      dacă min > a[j] atunci
        min ← a[j]
        k ← j
      sfârşit dacă
      dacă min ≠ a[i] atunci
        aux ← a[i]
        a[i] ← a[k]
        a[k] ← aux
      sfârşit dacă
    sfârşit pentru
  sfârşit pentru
sfârşit subalgoritm
```

Algoritmul se poate scrie şi prin determinarea valorilor maxime şi mutarea lor în tablou de la dreapta la stânga, astfel rezultând, de asemenea, un şir ordonat crescător.

Exemplu

Fie tabloul $A = (5, 0, 8, 7, 3)$.

Pas	*Tabloul A*	*Element minim*	*Poziția minimului*	*Noul tablou A*
1	(5, **0**, 8, 7, 3)	0	2	(**0**, 5, 8, 7, 3)
2	(0, 5, 8, 7, **3**)	3	5	(0, **3**, 8, 7, 5)
3	(0, 3, 8, 7, **5**)	5	5	(0, 3, **5**, 7, 8)
4	(0, 3, 5, **7**, 8)	7	4	

9.1.4. Sortarea prin inserție directă

Ideea algoritmului este de a considera pe rând fiecare element al tabloului şi de a-l insera în subtabloul ordonat până la momentul respectiv pe locul său. Acest loc s-ar putea să nu fie definitiv, deoarece este posibil ca următorul element să se insereze undeva în fața acestuia.

Dacă s-a ajuns la elementul a_j, (j = 1, 2, ..., n) şi toate elementele a_1, a_2, ..., a_{j-1} au fost ordonate, înseamnă că trebuie să-l inserăm pe a_j între aceste elemente. Îl comparăm pe rând cu a_{j-1}, a_{j-2}, ... până când ajungem la un prim element a_i având proprietatea $a_i < a_j$. Prin urmare a_j se inserează după a_i. Această inserare provoacă deplasarea elementelor care-l succed pe a_i cu o poziție spre dreapta.

```
Subalgoritm Inserţie(n,a):
  pentru i=2,n execută:
    copie ← a[i]  { salvăm al i-lea element (s-ar putea pierde cu translatările) }
    p ← 0                                  { în p vom avea indicele unde îl vom pune }
    j ← i - 1            { vom căuta locul printre elementele din fața lui a[i] }
    găsit ← fals
    cât timp (j ≥ 1) şi nu găsit execută:
      dacă a[j] ≤ copie atunci
        p ← j                  { dacă a[j] ≤ a[i], aici trebuie să-l punem }
        găsit ← adevărat                                  { am găsit locul }
      altfel
        a[j+1] ← a[j] { până nu am găsit locul, mutăm elementele la dreapta }
      sfârşit dacă
      j ← j - 1
    sfârşit cât timp
    a[p+1] ← copie                  { îl punem pe a[i] pe locul „eliberat" }
  sfârşit pentru
sfârşit subalgoritm
```

Exemplu

Fie tabloul A = (3, 3, 2, 0, 0, 5).

i	$j = i - 1$	$copie = a_i$	*Relație*	*găsit*	p	A
2		3		*fals*	0	(**3**, 3, 2, 0, 0, 5)
2	1	3	$3 \le 3$	*adevărat*	1	(**3**, **3**, 2, 0, 0, 5)
3		2		*fals*	0	(**3**, **3**, 2, 0, 0, 5)
3	2	2	$2 \le 3$	*adevărat*	2	(**2**, **3**, **3**, 0, 0, 5)
4	3	0	$3 > 0$	*fals*	0	(**2**, **3**, **3**, **3**, 0, 5)
4	2	0	$3 > 0$	*fals*	0	(**2**, **3**, **3**, **3**, 0, 5)

4	1	0	2 > 0	*fals*	0	(**2**, **2**, **3**, **3**, 0, 5)
4	0	0			0	(**0**, **2**, **3**, **3**, 0, 5)
5	4	0	3 > 0	*fals*	0	(**0**, **2**, **3**, **3**, **3**, 5)
5	3	0	3 > 0	*fals*	0	(**0**, **2**, **3**, **3**, **3**, 5)
5	2	0	2 > 0	*fals*	0	(**0**, **2**, **2**, **3**, **3**, 5)
5	1	0	0 = 0	*adevărat*	1	(**0**, **0**, **2**, **3**, **3**, 5)

9.1.5. Sortare prin numărarea aparițiilor

Algoritmii prezentați anterior sunt relativ mari consumatori de timp. De exemplu, pentru a ordona un şir de 1000 de elemente, numărul comparațiilor pe care le va executa oricare dintre algoritmii prezentați va fi aproximativ de 1 milion. În clasa a 10-a vom învăța alte metode de ordonare cu un timp de execuție mai bun. Dar, considerăm important să se învețe şi cel puțin o metodă de ordonare care se implementează cu un algoritm *liniar*. Un algoritm liniar execută un număr de operații proporțional cu numărul elementelor, adică pentru a ordona 1000 de elemente vor fi necesare $c \cdot 1000$ de operații, unde c este o constantă.

Dacă avem un şir de elemente de tip ordinal care sunt dintr-un interval de cardinalitate nu foarte mare, vom putea realiza o *ordonare liniară*. Corespunzător fiecărei valori întâlnite în şir în timpul prelucrării mărim cu 1 valoarea elementului având indicele (în acest şir de contoare) egal cu valoarea elementului în şirul de ordonat. În final, vom suprascrie în şirul dat atâtea elemente cu valori ai indicilor elementelor diferite de 0 cât este valoarea elementului în acest şir a numerelor de apariții.

Important este să reținem particularitățile pe care trebuie să le aibă şirul dat pentru ca această metodă să se poată aplica:

- valorile elementelor trebuie să fie de tip ordinal;
- numărul elementelor mulțimii din care şirul primeşte valori trebuie să fie relativ mic (dacă în program nu există alte structuri de date, din cei 64KB ai memoriei, scădem spațiul necesar şirului de ordonat şi putem calcula spațiul pe care îl avem la dispoziție);
- valorile posibile în şirul de ordonat trebuie să fie din intervalul $[x..y]$, unde $y - x + 1$ va fi dimensiunea şirului de contoare.

Exemplu

```
var sir:array[1..20000] of Integer;          { cel mult 20000 de numere }
    frecv:array[-500..500] of Integer; { valori posibile între -500 şi 500 }

Subalgoritm Ordonare_cu_Şir_de_Frecvențe(n,a,x,y):
  pentru i=x,y execută:
    frecv[i] ← 0
  sfârşit pentru
```

```
  pentru i=1,n execută:
    frecv[a[i]] ← frecv[a[i]] + 1
  sfârşit pentru
  k ← 0
  pentru i=x,y execută:
    pentru j=1,frecv[i] execută:
      k ← k + 1
      a[k] ← i
    sfârşit pentru
  sfârşit pentru
sfârşit subalgoritm
```

Fie şirul (–2, 0, –1, 0, –2, 3, 0, –2). Pentru acesta declaraţia cea mai „strânsă” în Pascal este:

```
var sir:array[1..8] of Integer;
    frecv:array[-2..3] of Byte;
```

Şirul de frecvenţe va fi (3, 1, 3, 0, 0, 1).

Corespunzător ultimei secvenţe din algoritm se vor scrie trei valori –2 consecutive în a, apoi 1 element va fi egal cu –1, urmează trei bucăţi de 0, urmat de o valoare 3: –2, –2, –2, –1, 0, 0, 0, 3.

9.2. Algoritmi de căutare

9.2.1. Căutare secvenţială

Se dă un tablou unidimensional $A = (a_1, a_2, ..., a_n)$ şi o valoare p. Se pune problema să verificăm dacă valoarea p se află între elementele tabloului sau nu.

Dacă şirul *nu* este ordonat trebuie să începem căutarea, comparând primul element din şir cu elementul căutat şi să continuăm fie până când îl găsim, fie ajungem la sfârşitul şirului, aşa cum am învăţat în capitolul 6. Dacă trebuie să aflăm toate poziţiile din şir pe care se află valoarea căutată, va trebui să îl parcurgem în întregime.

9.2.2. Căutare binară

Dacă tabloul A este sortat în ordine crescătoare, atunci în schimb, se poate realiza o căutare mai rapidă. Presupunem aşadar că elementele tabloului A sunt aranjate în ordine crescătoare.

Între elementele tabloului A avem relaţiile $a_1 \le a_2 \le ... \le a_{m-1} \le a_m \le a_{m+1}, ... \le a_n$, unde a_m reprezintă elementul din mijloc, poziţie calculată cu formula $m = [(n + 1)/2]$. Valoarea căutată este p. De asemenea, convenim să notăm cu s extremitatea stângă şi cu d extremitatea dreaptă a intervalului în care efectuăm căutarea. Iniţial $s = 1$ şi $d = n$.

Comparăm valoarea p cu valoarea elementului din şir de pe poziția din mijloc m. Dacă $p = a_m$, atunci căutarea se încheie cu succes. Dacă $p \neq a_m$, atunci vom decide în care parte a şirului (împărțit de a_m în două subşiruri) vom continua căutarea. Dacă $p < a_m$, atunci căutarea se va continua în prima jumătate a şirului, iar dacă $p > a_m$, atunci căutarea se transferă în a doua jumătate a tabloului. Astfel eliminăm un număr considerabil de comparații şi timpul de lucru al programului se reduce.

Să presupunem că $p < a_m$. În acest caz particularizăm capetele intervalului în care vom căuta în continuare astfel: $s = 1$ şi $d = m - 1$, ținând cont de faptul că elementul a_m nu mai necesită verificări.

Dacă $p > a_m$, atunci $s = m + 1$ va fi indicele extremității stângi al intervalului nou şi n va fi extremitatea dreaptă.

În funcție de relația dintre p şi a_m se alege unul dintre subşiruri pentru care se determină indicele elementului din mijloc în mod asemănător: $[(s + d)/2]$.

Algoritmul se continuă până când fie la un moment dat $a_m = p$, ceea ce înseamnă că am încheiat căutarea cu succes, fie ajungem la un subşir curent vid, adică având marginea din stânga mai mare decât marginea din dreapta ($s > d$), ceea ce înseamnă că avem o căutare fără succes.

Algoritmul descris mai sus este:

```
Subalgoritm Căutare_binară(n,a,p):
  s ← 1
  d ← n
  este ← fals
  cât timp s ≤ d şi nu este execută
    m ← [(s+d)/2]
    dacă p = a[m] atunci
      este ← adevărat
    altfel
      dacă p < a[m] atunci
        d ← m - 1
      altfel
        s ← m + 1
      sfârşit dacă
    sfârşit dacă
  sfârşit cât timp
  dacă este atunci
    scrie 'Valoarea ',p,' se afla pe pozitia ',m
  altfel
    scrie 'Valoarea ',p,' nu se afla in sir.'
  sfârşit dacă
sfârşit subalgoritm
```

Exemplu

Fie tabloul de numere întregi (1, 3, 5, 7, 9, 11) şi $p = 1$.

s	d	m	p	$a_m = p$?	Relaţia dintre a_m şi p	*este*
1	6	3	1	$a_m = 5 \neq p = 1$	$p < a_m$ (1 < 5) trecem în subşirul stâng	*fals*
1	2	1	1	$a_m = 1 = p = 1$	$p = a_m$ (1 = 1) $\Rightarrow$ căutare cu succes	*adevărat*

Vom căuta în acelaşi tablou valoarea $p = 12$.

s	d	m	p	$a_m = p$?	Relaţia dintre a_m şi p	*este*
1	6	3	12	$a_m = 5 \neq p = 12$	$p > a_m$ (12 > 5) trecem în subşirul drept	*fals*
4	6	5	12	$a_m = 9 \neq p = 12$	$p > a_m$ (12 > 9) trecem în subşirul drept	*fals*
6	6	6	12	$a_m = 11 \neq p = 12$	$p > a_m$ (12 > 11) trecem în subşirul drept	*fals*
7	6	6	12	$a_m = 11 \neq p = 12$	$p > a_m$ (12 > 11) trecem în subşirul drept	*fals*
$s > d$ şi *este* = *fals* $\Rightarrow$ algoritmul se termină cu căutare fără succes						

9.3. Interclasarea a două tablouri unidimensionale

În capitolul 6 am văzut cum procedăm în cazul în care dorim să creăm un şir nou din două şiruri date, astfel încât fiecare valoare să apară o singură dată, deoarece cu acele şiruri am simulat mulţimi. În cazul în care şirurile sunt ordonate, problema „intersecţiei" şi a „reuniunii" se prezintă altfel.

Fie $A = (a_1, a_2, ..., a_n)$ având elementele $a_1 \leq a_2 ... \leq a_n$ şi $B = (b_1, b_2, ..., b_m)$ având elementele $b_1 \leq b_2 \leq ... \leq b_m$. Se cere să se construiască tabloul $C = (c_1, c_2, ..., c_{n+m})$ având elementele $c_1 \leq c_2 \leq ... \leq c_{n+m}$ care să conţină toate elementele lui A şi B.

Exemplu

$A = (2, 4, 6, 10)$, $n = 4$
$B = (1, 3, 5, 9, 10, 11)$, $m = 6$
$C = (1, 2, 3, 4, 5, 6, 9, 10, 10, 11)$, $k = 10$

În rezolvare vom avansa în paralel în cele două şiruri date. Primul element în şirul nou va fi fie a_1, fie b_1. Pentru a şti care dintre cele două numere va fi ales, trebuie să le comparăm. Dacă a_1 este mai mic decât b_1, atunci pe acesta îl „consumăm" din şirul a şi pregătim indicele elementului următor din acesta. În caz alternativ, procedăm la fel cu indicele j, după ce aşezăm elementul b_1 în şirul c. (Dacă ni s-ar cere ca fiecare valoare să apară o singură dată, în caz de egalitate, am copia unul dintre aceştia în c, şi am avansa în ambele şiruri.) Vom efectua aceşti paşi până când careva din şiruri se sfârşeşte, moment în care copiem elementele rămase din celălalt şir în şirul nou. Din moment ce nu putem şti care şir s-a „consumat" integral, vom scrie secvenţa respectivă pentru ambele şiruri.

```
Subalgoritm Interclasare1(n,a,m,b,buc,c):
  buc ← 0
  i ← 1
  j ← 1
  cât timp (i ≤ n) şi (j ≤ m) execută:
    buc ← buc + 1
    dacă a[i] < b[j] atunci
      c[buc] ← a[i]
      i ← i + 1
    altfel
      c[buc] ← b[j]
      j ← j + 1
    sfârşit dacă
  sfârşit cât timp
  cât timp i ≤ n execută:
    buc ← buc + 1
    c[buc] ← a[i]
    i ← i + 1
  sfârşit cât timp
  cât timp j ≤ m execută:
    buc ← buc + 1
    c[buc] ← b[j]
    j ← j + 1
  sfârşit cât timp
sfârşit subalgoritm
```

Analizând acest algoritm, observăm că dacă am fi avut „norocul" ca a_n să fie egal cu b_m, atunci ultimele două structuri **cât timp** nu se executau niciodată. Vom exploata această observație, aşezând două elemente fictive (numite frecvent *santinele*) după ultimele elemente din cele două şiruri. Deoarece şirurile sunt ordonate crescător, vom fi atenți să aibă valorile mai mari decât ultimul (cel mai mare) din celălalt şir (l-am notat cu `infinit`).

```
Subalgoritm Interclasare2(n,a,m,b,buc,c):
  buc ← 0
  i ← 1
  j ← 1
  a[n+1] ← infinit
  b[m+1] ← infinit
  cât timp (i < n+1) sau (j < m+1) execută:
    buc ← buc + 1
```

```
    dacă a[i] < b[j] atunci
      c[buc] ← a[i]
      i ← i + 1
    altfel
      c[buc] ← b[j]
      j ← j + 1
    sfârşit dacă
  sfârşit cât timp
sfârşit subalgoritm
```

Deoarece se ştie că numărul elementelor în şirul *c* este *n* + *m*, (se admit şi elemente identice în şirul rezultat) structura repetitivă utilizată va fi de tip **pentru**.

```
Subalgoritm Interclasare3(n,a,m,b,buc,c):
  i ← 1
  j ← 1
  a[n+1] ← infinit
  b[m+1] ← infinit
  pentru buc=1,n+m execută:
    dacă a[i] < b[j] atunci
      c[buc] ← a[i]
      i ← i + 1
    altfel
      c[buc] ← b[j]
      j ← j + 1
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

9.4. Implementări sugerate

Pentru a fi pregătiți să rezolvați cu uşurință probleme în care, de multe ori cheia succesului este o ordonare sau o căutare corect şi eficient implementată, vă recomandăm să rezolvați următoarele exerciții:

1. verificarea faptului că un şir este ordonat;
2. verificarea faptului că elementele unui şir sunt în progresie aritmetică/geometrică;
3. ordonare cu metoda bulelor;
4. sortare prin selecție;
5. sortare prin inserție;
6. sortare prin numărare;
7. căutare binară;
8. interclasare a două şiruri;
9. căutare binară cu determinarea extremităților aparițiilor;

10. inserarea unui element într-un şir ordonat cu păstrarea ordinii;
11. determinarea celei mai lungi secvenţe care este formată din elemente având o proprietate dată;
12. determinarea celei mai lungi secvenţe formată din elemente având suma maximă.

9.5. Probleme propuse

9.5.1. Lucrări codificate

La sfârşitul examenului de capacitate, înainte să se desfacă lucrările, trebuie afişate două liste care să conţină codurile cu care s-au numerotat lucrările şi mediile scrise pe acestea. Prima listă va fi ordonată crescător după coduri, a doua va conţine aceleaşi informaţii, dar lista va fi ordonată descrescător după medii.

Date de intrare
Din fişierul de intrare **ELEVI.IN** se citesc mai multe linii, fiecare linie cuprinzând date despre câte o lucrare. Pe aceste linii vom avea un număr întreg, reprezentând codul lucrării şi un număr real, reprezentând media.

Date de ieşire
Fişierul de ieşire **CODURI.OUT** va conţine lista lucrărilor aranjată crescător după coduri, iar fişierul **MEDII.OUT** va conţine lista de lucrări ordonată descrescător după medii. În ambele fişiere se vor scrie perechi de cod-medie, despărţite printr-un spaţiu.

Restricţii şi precizări
- afişarea în fişierele de ieşire a mediilor se va face cu două zecimale exacte;
- numărul lucrărilor ≤ 5000.

Exemplu

ELEVI.IN	**CODURI.OUT**	**MEDII.OUT**
1 9.60	1 9.60	4 10.00
2 8.77	2 8.77	1 9.60
5 9.20	3 8.90	5 9.20
4 10.00	4 10.00	3 8.90
3 8.90	5 9.20	2 8.77

9.5.2. Unificarea listelor

La inspectoratul şcolar s-au strâns *n* liste din şcolile judeţului de la examenul de capacitate. Din păcate, unele liste sunt ordonate crescător după medii, altele descrescător. Ajutaţi inspectoratul şcolar şi scrieţi un program care creează o singură listă din cele *n* liste în care mediile să fie ordonate descrescător.

Date de intrare
Pe prima linie a fişierului de intrare **LISTE.IN** se află un număr natural, reprezentând numărul listelor. Pe a doua linie se află un număr natural reprezentând dimensiunea d_1 a primei liste. Pe următoarele d_1 linii se află perechi de cod-medie separate printr-un spaţiu. Pe următoarele linii se află descrierea celorlalte liste în mod similar cu prima.

Date de ieşire
Fişierul **LISTE.OUT** va conţine componentele listei unificate. Pe fiecare linie se va afla câte o pereche cod-medie despărţite printr-un spaţiu.

Restricţii şi precizări
- numărul de elemente din fiecare tablou este cel mult 50.

Exemplu

LISTE.IN
```
4
3
1 9.50
2 8.75
3 8.40
5
6 7.25
8 7.76
5 7.95
4 8.35
11 9.50
2
13 10.00
12 4.50
4
14 8.65
10 4.80
7 4.75
9 4.50
```

LISTE.OUT
```
13 10.00
11  9.50
 1  9.50
 2  8.75
14  8.65
 3  8.40
 4  8.35
 5  7.95
 8  7.76
 6  7.25
10  4.80
 7  4.75
 9  4.50
12  4.50
```

9.5.3. Alegeri prezidenţiale

Se apropie alegerile şi organizatorii se pregătesc cu pachete de programe cu care să-şi uşureze munca în viitorul apropiat. Se prevede că la alegerile prezidenţiale vor fi foarte mulţi candidaţi. Scrieţi un program care va stabili pe baza voturilor dacă există sau nu un câştigător la aceste alegeri. Se consideră că pentru a câştiga, un candidat trebuie să totalizeze un număr de voturi cel puţin cu 1 mai mult decât jumătate din numărul total de voturi.

Date de intrare
Fişierul de intrare **VOT.IN** conține atâtea linii câte voturi există. Pe fiecare linie este scris un număr natural reprezentând numărul de ordine al unui candidat.

Date de ieşire
În fişierul de ieşire **VOT.OUT** se va scrie cuvântul `'DA'` în cazul în care există câştigător la alegeri şi `'NU'` în caz contrar. În cazul în care există câştigător, pe următoarea linie din fişier se va scrie numărul de ordine al câştigătorului.

Restricții şi precizări
- 1 ≤ *numărul voturilor* ≤ 1000000000;
- 1 ≤ *număr de ordine* ≤ 1000000000.

Exemple

VOT.IN
```
4
3
3
1
3
3
```

VOT.OUT
```
DA
3
```

VOT.IN
```
1
1
3
3
```

VOT.OUT
```
NU
```

9.5.4. Plăți şi încasări

Casierul unei firme trebuie să parcurgă zilnic un traseu liniar prin cele *n* magazine ale sale. Magazinele sunt numerotate de la 1 la *n* şi se parcurg zilnic în ordinea numerelor lor de ordine, începând cu primul magazin şi terminând cu al *n*-lea. (Evident casierul va trece doar o singură dată prin fiecare magazin.) Scopul parcurgerii acestui traseu este efectuarea unor plăți, respectiv a unor încasări. Se ştie că există cel puțin un magazin de unde casierul va încasa bani. De asemenea se cunosc sumele care trebuie plătite, respectiv încasate.

Într-o zi, patronul nu are bani pentru plăți şi cere casierului să aleagă două magazine m_i şi m_j de pe traseu, astfel încât parcurgând traseul începând cu m_i şi încheindu-l cu m_j, încasările să reprezinte suma maximă posibilă de adus înapoi la firmă. Casierul trebuie să intre în toate magazinele aflate pe traseul ales, chiar dacă în unele trebuie să efectueze plăți. În plus, pentru a nu crea suspiciuni printre angajații magazinelor, patronul doreşte să fie vizitate cât mai multe magazine.

Scrieți un program care determină suma maximă posibilă de încasat din magazinele firmei. De asemenea, determinați secvența cea mai lungă de magazine care permite obținerea acestei sume. Dacă aceeaşi sumă maximă se poate obține pentru mai multe alegeri diferite, vizitând acelaşi număr maxim de magazine, atunci în fişierul de ieşire se va scrie una singură, şi anume cea care începe de la magazinul cu numărul de ordine cel mai mic.

Date de intrare
Pe prima linie a fişierului de intrare **CASIER.IN** se află numărul natural *n*, reprezentând numărul magazinelor de pe traseul inițial. Pe următoarea linie se află *n* numere întregi, despărțite prin câte un spațiu, reprezentând sumele pe care casierul ar trebui să le plătească sau să le încaseze în/din cele *n* magazine; numerele pozitive reprezintă sume care se vor încasa, iar numerele negative corespund sumelor care se vor plăti.

Date de ieşire
Pe prima linie a fişierului de ieşire **CASIER.OUT** se va scrie un număr natural nenul, reprezentând suma maximă cu care casierul se întoarce la firmă. Pe următoarea linie se vor scrie două numere naturale nenule, aparținând intervalului [1, *n*] care reprezintă numerele de ordine ale magazinelor alese de casier pentru începutul şi sfârşitul porțiunii de traseu.

Restricții
- $1 \le n \le 1000000000$;
- $-1000000 \le Suma_i \le 1000000$, $i = 1, 2, \ldots, n$;
- $1 \le SumaMax \le 1000000000$.

Exemplu

CASIER.IN	**CASIER.OUT**
10	13
2 3 -6 5 -6 6 7 -2 2 -1	6 9

9.6. Soluțiile problemelor propuse

9.6.1. Lucrări codificate

Vom lucra cu două tablouri: *cod* şi *med* în care vom reține pe poziții identice codurile lucrărilor şi mediile scrise pe ele.

Citirea datelor din fişierul de intrare continuă până la apariția mărcii de sfârşit de fişier. Concomitent, vom număra şi numărul perechilor de date citite pentru a şti numărul lucrărilor (*n*).

Pentru sortarea după coduri folosim metoda bulelor îmbunătăţită. Este important să reţinem că atunci când interschimbăm două elemente din tabloul de coduri, trebuie să interschimbăm şi elementele respective lor din tabloul de medii pentru a menţine corespondenţa dintre elemente.

Algoritmul care formează cea de-a doua listă trebuie să realizeze o sortare descrescătoare a elementelor din tabloul de medii. Vom aplica metoda sortării prin selectarea maximului şi aşezarea lui pe poziţia sa definitivă. Ca şi în cazul precedent, la o interschimbare de medii trebuie realizată şi interschimbarea codurilor.

9.6.2. Liste

Să luăm exemplul din enunţ. Pentru o afişare mai potrivită, vom scrie listele pe coloane:

```
3               5               2               4
1 9.50           6 7.25         13 10.00        14 8.65
2 8.75           8 7.76         12  4.50        10 4.80
3 8.40           5 7.95                          7 4.75
                 4 8.35                          9 4.50
                11 9.50
```

Avem de interclasat, lăsând la o parte pentru moment codurile, 4 tablouri având dimensiunile 3, 5, 2, 4.

Ne propunem să interclasăm la fiecare pas câte două tablouri, păstrând în primul rezultatul interclasării. Trebuie să interclasăm şiruri ordonate, dar din nefericire acestea nu sunt ordonate în acelaşi sens. Din acest motiv, înainte de a interclasa două dintre ele mai întâi trebuie să vedem dacă sunt ordonate în acelaşi sens. În plus, rezultatul trebuie să fie ordonat descrescător după medie. În caz de ordonări necorespunzătoare, algoritmul de interclasare va parcurge şirul respectiv în ordine inversă.

Revenind la exemplul considerat, dacă la primul pas interclasăm primele două tablouri, obţinem:

```
9.50 9.50 8.75 8.40 8.35 7.95 7.76 7.25
```

Ne-am atins de 3 plus 5 elemente şi a rezultat un şir de **8** elemente. Acesta se va interclasa cu următorul, prelucrând în total 8 + 2 = **10** elemente, urmând ca în final acest şir să se interclaseze cu ultimul, prelucrând 10 + 4 = **14** elemente, ceea ce înseamnă că în total am efectuat **32** de prelucrări de elemente. Se observă uşor că dacă am alege să interclasăm mereu cele mai scurte două şiruri, numărul elementelor prelucrate în total scade: interclasăm şirul de lungime 2 cu cel de lungime 3 (**5** elemente „mişcate"), şirul rezultat cu cel de lungime 4 (avem **7** mişcări) şi în final mai avem 7 + 4 = **11** prelucrări. Numărul total de prelucrări de elemente acum este **23**.

Strategia pe care o vom aborda în rezolvare este de a afla la fiecare pas cele mai scurte două tablouri în vederea interclasării lor.

În reprezentarea algoritmului am folosit următoarele notaţii:
med: şirul în care fiecare element este un şir de medii;
cod: şirul în care fiecare element este un şir de coduri;
dim: şirul lungimilor listelor;
amed, *bmed*: două şiruri de medii din două liste care urmează să fie interclasate;
acod, *bcod* două şiruri de coduri din două liste care urmează să fie interclasate în paralel cu mediile;
adim, *bdim*: dimensiunile şirurilor *amed*, respectiv *bmed*;
cmed: lista de medii rezultată în urma interclasării lui *amed* şi *bmed*;
ccod: lista de coduri rezultată în urma interclasării lui *acod* şi *bcod*;
n: este numărul listelor.

Algoritmul urmează strategia de interclasare sugerată prin exemplul tratat:

```
Algoritm Liste:
  citirea datelor de intrare:                          { n,med,cod,dim }
  cât timp n ≤ 2 execută:        { până la interclasarea ultimelor două şiruri }
    indsir1 ← Minim(n,dim)          { indicele şirului de dimensiune minimă }
    adim ← dim[indsir1]  { în adim păstrăm dimensiunea celui mai scurt şir }
    acod ← cod[indsir1]       { în acod păstrăm cel mai scurt şir de coduri }
    amed ← med[indsir1]        { în amed păstrăm cel mai scurt şir de medii }
    Elimin(n,indsir1,dim,cod,med)            { eliminăm cel mai scurt şir }
    indsir2 ← Minim(n,dim)  { procedăm la fel cu al doilea cel mai scurt şir }
    bdim ← dim[indsir2]
    bcod ← cod[indsir2]
    bmed ← med[indsir2]
    Elimin(n,indsir2,dim,cod,med)
    dacă amed[1] < amed[2] atunci          { verificăm sensul de ordonare }
      Invers(adim,acod,amed)       { dacă este în sens crescător, îl inversăm }
    sfârşit dacă
    dacă bmed[1]  <bmed[2] atunci
      Invers(bdim,bcod,bmed)
                  { interclasăm acod cu bcod în ccod şi amed cu bmed în cmed }
    sfârşit dacă
    Inter(acod,bcod,ccod,amed,bmed,cmed,adim,bdim,adim+bdim)
    Adaug(n,ccod,cmed,adim+bdim,cod,med,dim)
  sfârşit cât timp
                                { rezultatul este în ultimul rezultat al interclasării }
  Afişare(ccod,cmed,adim+bdim)
sfârşit algoritm
```

În algoritmul principal se apelează câțiva subalgoritmi dintre care `Minim` şi `Inter` îi considerăm cunoscuți. În algoritmul de interclasare santinelele vor fi două valori străine de şirul de medii egale cu 0, deoarece şirurile sunt ordonate descrescător:

```
a[n+1] ← 0
b[m+1] ← 0
```

Subalgoritmul de eliminare a şirului având indicele *ind* (care se va interclasa la un moment dat) dintre şirurile care urmează să fie interclasate:

```
Subalgoritm Elimin(n,ind,dim,cod,med):
                                        { elimină şirurile care se vor interclasa }
  i ← 1
  cât timp (i ≤ n) şi (i ≠ ind) execută:
    i ← i + 1
  sfârşit cât timp
  pentru j=i,n-1 execută:      { începând cu al i+1-lea translatăm la stânga }
    dim[j] ← dim[j+1]
    cod[j] ← cod[j+1]
    med[j] ← med[j+1]
  sfârşit pentru
  n ← n - 1                                    { şirul devine mai scurt }
sfârşit subalgoritm
```

Şirul obținut în urma interclasării (având lungimea *d*) devine unul care urmează să fie interclasat cu restul şirurilor. Îl adăugăm la sfârşitul şirului de şiruri:

```
Subalgoritm Adaug(n,ccod,cmed,d,cod,med,dim):
                                                { adăugăm şirul interclasat }
  n ← n + 1                                        { şirul devine mai lung }
  dim[n] ← d
  cod[n] ← ccod
  med[n] ← cmed
sfârşit subalgoritm
```

Şirurile care nu sunt descrescător ordonate trebuie transformate astfel încât să fie corespunzătoare cerințelor. Evident, se putea scrie şi o interclasare care pe acestea să le parcurgă în sens invers, dar s-ar fi lungit întreg codul programului. În cazul în care timpul de execuție trebuie redus la minim, recomandăm implementarea unei interclasări în acest sens.

```
Subalgoritm Invers(d,cod,med):
                    { dacă şirul de medii este ordonat crescător, inversăm ordinea }
  pentru i=1,[d/2] execută:
    aux1 ← med[i]
    med[i] ← med[d-i+1]
    med[d-i+1] ← aux1
    aux2 ← cod[i]
    cod[i] ← cod[d-i+1]
    cod[d-i+1] ← aux2
  sfârşit pentru
sfârşit subalgoritm
```

9.6.3. Alegeri prezidenţiale

Această problemă face parte din categoria acelora care necesită o prelucrare a datelor pe segmente de lungime variabilă. Lungimea unui astfel de subşir depinde de valoarea elementelor şirului. De altfel, datorită lungimii maxime posibile este exclus să citim toate datele şi să le păstrăm în memorie. În concluzie, vom acţiona pe baza valorii unui singur număr.

Numărul curent îl notăm cu *nr*, iar candidatul curent cu *candidat*. Dacă un număr citit din fişier este egal cu *candidat*, mărim contorul *câte* în care ţinem evidenţa apariţiilor candidatului. Dacă numărul citit diferă de *candidat*, indiferent de valoarea lui, scădem 1 din contorul *câte*. După fiecare astfel de scădere verificăm contorul *câte*, deoarece în momentul în care acesta devine 0, înseamnă că s-a sfârşit un segment şi începe altul cu un *candidat* nou.

```
Algoritm Alegeri_prezidenţiale:
  citeşte nr
  candidat ← nr
  câte ← 1            { în câte numărăm voturi egale pe „segmente" de date }
  cât timp nu urmează marca de sfârşit de fişier execută:
    citeşte nr
    dacă nr = candidat atunci  { dacă a apărut vot pentru acelaşi candidat }
      câte ← câte + 1
    altfel                        { dacă votul nu este pentru candidatul curent }
      câte ← câte - 1      { micşorăm numărul voturilor primite de candidat }
                           { dacă prin micşorare numărul voturilor devine 0 }
      dacă câte = 0 atunci
        candidat ← nr              { schimbăm candidatul cu ultimul votat }
        câte ← 1                      { pentru acesta a apărut primul vot }
      sfârşit dacă
    sfârşit dacă
  sfârşit cât timp
```

```
                                { am terminat prelucrarea fişierului; am ieşit cu un candidat }
                                { deoarece nu este sigur că acesta are suficiente voturi }
  câte ← 0                                                { parcurgem fişierul din nou }
  n ← 0
  cât timp nu urmează marca de sfârşit de fişier execută:
    citeşte nr
    n ← n + 1                                             { numărăm numărul voturilor }
    dacă nr = candidat atunci    { şi numărul voturilor primite de candidat }
      câte ← câte + 1
    sfârşit dacă
  sfârşit cât timp
  dacă câte > [n/2] atunci
    scrie 'DA'
    scrie candidat
  altfel
    scrie 'NU'
sfârşit algoritm
```

9.6.4. Plăți şi încasări

Înainte de toate observăm că trebuie să determinăm o subsecvență (subşir format din elemente consecutive din şir) având proprietatea că este cea mai lungă printre acelea care au sumă maximă.

Vom considera exemplul din enunț.

Fie $n = 10$ şi şirul: 1, 2, –6, 3, 4, 5, –2, 10, –5, –6.

Rezolvând problema manual pentru acest exemplu, obținem:

- Suma maximă: 20;
- Lungimea subsecvenței: 5;
- Subsecvența căutată: 3, 4, 5, –2, 10.

Ideea cea mai simplă de rezolvare constă în generarea tuturor perechilor posibile de numere *stânga* şi *dreapta*, astfel încât: $1 \le stânga \le dreapta \le n$, urmând calcularea sumelor subsecvențelor corespunzătoare (T[*stânga*..*dreapta*]). Pe parcursul generării vom păstra acea subsecvență care are suma maximă.

```
Subalgoritm Subsecvență_1(n,T):
  Max ← T[1]                  { T este şirul dat, Max este suma maximă căutată }
  pentru stânga=1,n execută:    { stânga este primul indice al subsevenței }
                                { dreapta este ultimul indice al subsevenței }
    pentru dreapta=stânga,n execută:
      suma ← 0                                          { suma subsecvenței actuale }
      pentru i=stânga,dreapta execută:
        suma ← suma + T[i]
      sfârşit pentru
```

```
      dacă Max < suma atunci
        Max ← suma
      sfârşit dacă
    sfârşit pentru
  sfârşit pentru
sfârşit subalgoritm
```

Acest algoritm conține trei structuri repetitive incluse una în cealaltă (imbricate). Dacă $n = 1000$, un calculator de performanțe medii ar lucra aproximativ o oră. Această complexitate este de neacceptat, deci căutăm un algoritm mai eficient.

Observăm, de exemplu, că în suma subsecvenței $T[1..3]$ există deja suma aferentă subsecvenței $T[1..2]$. Adică $T[1..3] = T[1..2] + T[3]$ etc. Generalizând, suma subsecvenței $T[stânga..dreapta]$ se poate calcula cu ajutorul sumei determinate la pasul precedent, corespunzător subsecvenței $T[stânga..dreapta - 1]$.

Vom compara sumele corespunzătoare fiecărei subsecvențe $T[stânga..dreapta]$ cu suma maximă determinată până în momentul respectiv, *imediat* după calcularea lor. Astfel algoritmul va conține doar două cicluri imbricate.

```
Subalgoritm Subsecvență_2(n,T):
  Max ← T[1]                  { T este şirul dat, Max este suma maximă căutată }
  pentru stânga=1,n execută:    { stânga este primul indice al subsevenței }
    suma ← 0                 { începând cu fiecare stânga nou, avem sumă nouă }
    pentru dreapta=stânga,n execută:
      suma ← suma + T[dreapta]              { adăugăm un element nou }
      dacă Max < suma atunci
        Max ← suma                                { actualizăm Max }
      sfârşit dacă
    sfârşit pentru
  sfârşit pentru
sfârşit subalgoritm
```

Dacă am reuşi să rezolvăm problema astfel încât să parcurgem şirul o singură dată, algoritmul va fi *liniar*. În plus, să ne aducem aminte că problema garantează că în şir există cel puțin un număr pozitiv.

Deci, dacă şirul conține $n - 1$ elemente negative şi un singur element pozitiv, atunci *subsecvența având suma maximă ar fi constituită din acest singur element pozitiv*.

Rezultă că atâta vreme cât o sumă actuală este pozitivă, continuăm adunările. Dacă, la un moment dat această sumă devine negativă, atunci începem calcularea unei sume noi pe segmentul care începe cu primul număr pozitiv.

Dacă am observat că problema poate fi rezolvată pe segmente, următoarea optimizare constă în a renunța la citirea tuturor datelor şi reținerea lor în memorie. La un moment dat, vom citi un singur număr şi îl vom prelucra.

```
Algoritm Subşir_de_suma_maxima_3:
  citeşte n
  citeşte T
  MaxS_Caut ← T                              { iniţializarea sumei maxime }
  MaxS_Aici ← MaxS_Caut                      { iniţializarea sumei curente }
  poz_inc ← 1
  sfârşit ← 1
  început ← 1
  pentru i=2,n execută:                      { continuăm citirea numerelor }
    citeşte T
    dacă MaxS_Aici < 0 execută:              { dacă MaxS_Aici este negativă }
      MaxS_Aici ← T                          { aceasta se reiniţializează cu T }
      poz_inc ← i                            { noul subşir posibil începe în poziţia i }
    altfel
      MaxS_Aici ← MaxS_Aici + T              { lui MaxS_Aici i se adaugă T }
      dacă MaxS_Caut ≤ MaxS_Aici atunci      { se actualizează MaxS_Caut }
        dacă MaxS_Caut < MaxS_Aici atunci
          MaxS_Caut ← MaxS_Aici
          început ← poz_inc
          sfârşit ← i
        altfel                               { în caz de egalitate }
          dacă început = poz_inc atunci
            sfârşit ← i
          sfârşit dacă
        sfârşit dacă
      sfârşit dacă
    sfârşit dacă
  sfârşit pentru
  scrie MaxS_Caut
  scrie început,' ',sfârşit
sfârşit algoritm
```