ALGORITMICĂ. Seminar 6: Variante ale căutării binare. Aplicaţii ale interclasării şi partiţionării. Aplicaţii ale tehnicii greedy

Problema 1 (S) Se consideră un tablou $a[1..n]$ ordonat crescător şi v o valoare. Să se determine, folosind un algoritm din $\mathcal{O}(\lg n)$, poziţia unde poate fi inserată valoarea v în tabloul a astfel încât acesta să rămână ordonat crescător.

Rezolvare.

Structura generală a algoritmului:

```
cautare_pozitie(real a[1..k], v)
li ← 1; ls ← k
if (v ≤ a[li]) return(li) endif
if (v ≥ a[ls]) return(ls + 1) endif
while (ls > li)
   m ← (li + ls)/2
   if (a[m] = v) then return(m + 1) endif
   if (v < a[m]) then ls ← m − 1
      else li ← m + 1
   endif
endwhile
if (v ≤ a[li]) return(li)
if (v > a[ls]) return(ls + 1)
```

Corectitudinea poate fi demonstrată folosind ca proprietate invariantă faptul că $a[li-1] \leq v \leq a[ls+1]$ (în ipoteza că se presupune formal că $a[0] = -\infty$ şi $a[k+1] = \infty$). Intrucât structura algoritmului este similară algoritmului de căutare binară ordinul de complexitate este $\mathcal{O}(\lg n)$.

O altă variantă a algoritmului este:

```
cautare_pozitie(real a[1..k], v)
li ← 1; ls ← k
while (li ≤ ls)
   if (v ≤ a[li]) return(li) endif
   if (v ≥ a[ls]) return(ls + 1) endif
   m ← (li + ls)/2
   if (a[m] = v) then return(m + 1) endif
   if (v < a[m]) then ls ← m − 1
      else li ← m + 1
   endif
endwhile
return(li)
```

Şi în acest caz $a[li-1] \leq v \leq a[ls+1]$ este proprietate invariantă astfel că la ieşirea din ciclu (când $li = ls + 1$) are loc $a[li-1] \leq v \leq a[li]$. Prin urmare la ieşirea din ciclu trebuie returnată valoarea lui li.

Folosind acest algoritm de căutare binară a poziţiei de inserţie algoritmul de sortare prin inserţie poate fi transformat în:

```
insertie_binara(real a[1..n])
for i ← 2, n do
    aux ← a[i]
    poz ←cautare_pozitie(a[1..i − 1],v)
    for j ← i − 1, poz, −1 do a[j + 1] ← a[j] endfor
    a[poz] ← aux
endfor
return a[1..n]
```

Din punctul de vedere al numărului de comparaţii efectuate algoritmul de inserţie binară are complexitatea $\mathcal{O}(n \lg n)$. Din punctul de vedere al numărului de deplasări de elemente efectuate algoritmul de sortare prin inserţie binară aparţine lui $\mathcal{O}(n^2)$.

Problema 2 (S) *Căutare ternară.* Tehnica căutării binare poate fi extinsă prin divizarea unui subşir $a[li..ls]$ în trei subşiruri $a[li..m1]$, $a[m1+1..m2]$, $a[m2+1..ls]$, unde $m1 = li + \lfloor (ls - li)/3 \rfloor$ iar $m1 = li + 2\lfloor (ls - li)/3 \rfloor$.

```
cautare_ternara (integer a[1..n], v)
li ← 1; ls ← n;
while (li <= ls)
    m1 ← li + (ls − li)DIV3; m2 ← li + 2 ∗ (ls − li)DIV3
    if(x[m1] = v) then return(m1) endif
    if(x[m2] = v) then return(m2)endif
    if(v < x[m1]) then ls ← m1 − 1
    else if (v < x[m2])then li ← m1 + 1;ls ← m2 − 1;
        else li ← m2 + 1 endif
    endif endwhile
return(-1)
```

Notând cu $T(n)$ numărul maxim de comparaţii efectuate are loc relaţia:

$$T(n) = \begin{cases} 0 & n = 0 \\ T(n/3) + c & n > 0 \end{cases}$$

unde $c \in \{1, 2, 3, 4\}$ este numărul de comparaţii efectuate în cadrul unei iteraţii. Aplicând metoda substituţiei inverse pentru cazul particular $n = 3^k$ se obţine că $T(n) = c \log_3 n$ deci $T(n) \in \mathcal{O}(\log n)$, adică acelaşi ordin de complexitate ca în cazul căutării binare (diferă doar constantele multiplicative).

Problema 3 (S+L) *Metoda bisecţiei.* Fie $f : [a, b] \to R$ o funcţie continuă având proprietăţile: (i) $f(a)f(b) < 0$; (ii) există un unic x^* cu proprietatea că $f(x^*) = 0$. Să se aproximeze x^* cu precizia $\epsilon > 0$.

Rezolvare. A determina pe x^* cu precizia ϵ înseamnă a identifica un interval de lungime ϵ care conţine pe x^* sau chiar un interval de lungime 2ϵ dacă se consideră ca aproximare a lui x^* mijlocul intervalului. Se poate aplica exact aceeaşi strategie ca la căutarea binară ţinându-se cont ca x^* se află în intervalul pentru care funcţia f are valori de semne opuse în extremităţi .

```
bisectie(real a,b,epsilon)
li ← a; ls ← b
repeat
    m ← (li + ls)DIV2
    if f(m) = 0 then return m endif
    if f(m) * f(li) < 0 then ls ← m
        else li ← m
    endif
until |ls − li| < 2ε
return (li + ls)DIV2
```

Complexitatea este determinată de dimensiunea intervalului $[a, b]$ şi de precizia dorită a aproximării, ϵ. Notând $n = (b - a)/\epsilon$ se obţine că algoritmul are complexitatea $\mathcal{O}(\lg n)$.

Problema 4 (S+L) Fie $a[1..m]$ şi $b[1..n]$ două tablouri ordonate crescător având elemente nu neapărat distincte. Să se construiască un tablou strict crescător ce conţine elementele distincte din a şi b

Rezolvare. Se aplică tehnica interclasării verificând de fiecare dată la completarea în tabloul destinaţie dacă elementul ce ar trebui adăugat este diferit de ultimul element din tablou.

```
interclasare(integer a[1..m],b[1..n])
integer i,j, k, c[1..k]
i ← 1; j ← 1; k ← 0
if a[i] < b[j] then k ← k + 1; c[k] ← a[i]; i ← i + 1
      else if a[i] > b[j] then k ← k + 1; c[k] ← b[j]; j ← j + 1
         else k ← k + 1; c[k] ← a[i]; i ← i + 1; j ← j + 1
      endif
endif
while i ≤ m AND j ≤ n do
   if a[i] < b[j] then
      if a[i] ≠ c[k] thenk ← k + 1; c[k] ← a[i]; i ← i + 1
      else i ← i + 1 endif
   else if a[i] > b[j] then
      if b[j] ≠ c[k] thenk ← k + 1; c[k] ← b[j]; j ← j + 1
      else j ← j + 1 endif
      if a[i] ≠ c[k] thenk ← k + 1; c[k] ← a[i]; i ← i + 1;j ← j + 1
      else i ← i + 1;j ← j + 1 endif
      endif
   endif
endwhile
while i ≤ m do
   if a[i] ≠ c[k] thenk ← k + 1; c[k] ← a[i]; i ← i + 1
   else i ← i + 1 endif
endwhile
while j ≤ n do
   if b[j] ≠ c[k] thenk ← k + 1; c[k] ← b[j]; j ← j + 1
   else j ← j + 1 endif
endwhile
return c[1..k]
```

Ordinul de complexitate este în acest caz $\mathcal{O}(m+n)$.

Problema 5 (L) Se consideră două numere întregi date prin tablourile corespunzătoare descompunerilor lor în factori primi. Să se construiască tablourile similare corespunzătoare celui mai mic multiplu comun al celor două valori.

Rezolvare. Să considerăm numerele: $3415 = 3 \cdot 5^3 \cdot 7 \cdot 13$ şi $966280 = 2^3 \cdot 5 \cdot 7^2 \cdot 17 \cdot 29$. Primului număr îi corespund tablourile: $(3, 5, 7, 13)$ respectiv $(1, 3, 1, 1)$ iar celui de al doilea număr îi corespund tablourile $(2, 5, 7, 17, 29)$ respectiv $(3, 1, 2, 1, 1)$.

Tablourile corespunzătoare celui mai mic multiplu comun vor fi:

- *factori primi:* tabloul care conţine toţi factorii primi corespunzători celor două numere şi care poate fi obţinut prin interclasare ţinând cont că aceştia trebuie să fie distincţi.
- *exponenţi:* tabloul ce conţine valorile maxime ale exponenţilor.

În cazul exemplului cele două tablouri sunt: $(2, 3, 5, 7, 13, 17, 29)$ respectiv $(3, 1, 3, 2, 1, 1, 1)$

```
interclasare(integer fa[1..m],pa[1..m],fb[1..n],pb[1..n] )
integer i,j, k, fc[1..k],pc[1..k]
i ← 1; j ← 1; k ← 0
while i ≤ m AND j ≤ n do
   if fa[i] < fb[j] then
      k ← k + 1; fc[k] ← fa[i]; pc[k] ← pa[i]; i ← i + 1
   else if a[i] > b[j] then
      k ← k + 1; fc[k] ← fb[j]; pc[k] ← pb[j]; j ← j + 1
         else k ← k + 1; fc[k] ← fa[i]; pc[k] ← max(pa[i], pb[j]); i ← i + 1;j ← j + 1
      endif
   endif
endwhile
while i ≤ m do
   k ← k + 1; fc[k] ← fa[i]; pc[k] ← pa[i]; i ← i + 1
endwhile
while j ≤ n do
   k ← k + 1; fc[k] ← fb[j]; pc[k] ← pb[j]; j ← j + 1
endwhile
return fc[1..k], pc[1..k]
```

Problema 6 (S) *Problema selecţiei celui de al k-lea element.* Fie $a[1..n]$ un tablou, nu neapărat ordonat. Să se determine al k-lea element al tabloului, selectat în ordine crescătoare (pentru $k = 1$ se obţine minimul, pentru $k = n$ se obţine maximul etc.).

Rezolvare. O primă variantă de rezolvare o reprezintă ordonarea crescătoare a tabloului (prin metoda selecţiei) până ajung ordonate primele k elemente ale tabloului. Numărul de operaţii efectuate în acest caz este proporţional cu kn. Pentru k apropiat de $n/2$ aceasta conduce la un algoritm de complexitate pătratică. Un algoritm de complexitate mai mică se obţine aplicând strategia de partiţionare de la sortarea rapidă: folosind o valoare de referinţă (de exemplu cea aflată pe prima poziţie în tablou) se partiţionează tabloul în două subtablouri astfel încât elementele aflate în primul subtablou să fie toate mai mici decât valoarea de referinţă iar elementele din al doilea subtablou să fie toate mai mari decât valoarea de referinţă.

Algoritmul de partiţionare poate fi cel folosit în cadrul sortării rapide:

```
partiţionare(integer a[li..ls])
v ← a[li]; i ← li − 1; j ← lj + 1;
while i < j do
   repeat i ← i + 1 until a[i] ≥ v
   repeat j ← j − 1 until a[j] ≤ v
   if i < j then a[i] ↔ a[j] endif
endwhile
return j
```

Dacă poziţia de partiţionare este q iar $k \leq q - li + 1$ atunci problema se reduce la selecţia celui de al k-lea element din subtabloul $a[li..q]$. Dacă însă $k > q - li + 1$ atunci problema se reduce la selecţia celui de al $k - (q - li + 1)$ element din subtabloul $a[q + 1..ls]$. Valoarea parametrului k este întotdeauna cuprinsă între 1 şi numărul de elemente din subtabloul prelucrat (în cazul în care $li = ls$ valoarea lui k va fi 1). Algoritmul poate fi descris prin:

```
selectie(integer a[li..ls], k)
if li = ls then return a[li]
else
   q ←partitionare(a[li..ls])
   r ← q − li + 1
   if k ≤ r then selectie(a[li..q], k)
      else selectie(a[q + 1..ls], k − r)
   endif
endif
```

În cazul cel mai favorabil, partiţionarea este echilibrată la fiecare etapă (cele două subtablouri au aproximativ acelaşi număr de elemente). Întrucât algoritmul de partiţionare are complexitate liniară putem presupune că numărul de comparaţii efectuate asupra elementelor tabloului satisface următoarea relaţie de recurenţă:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(n/2) + n & n > 1 \end{cases}$$

Aplicând teorema master pentru estimarea ordinului de complexitate (pentru $m = 2$, $d = 1$, $k = 1$) se obţine că, în cazul cel mai favorabil, algoritmul are complexitate liniară. În cazul cel mai defavorabil partiţionarea ar conduce la fiecare etapă la descompunerea într-un subtablou constituit dintr-un singur element şi la un subtablou constituit din celelalte elemente. Relaţia de recurenţă ar fi în acest caz:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(n-1) + n & n > 1 \end{cases}$$

ceea ce conduce la o complexitate pătratică. Similar algoritmului de sortare rapidă, si algoritmul selecţiei se comportă în medie similar celui mai favorabil caz având o complexitate liniară.

Problema 7 (L) *Determinare mediană.* Mediana unui tablou cu n elemente este elementul aflat pe poziţia $\lfloor (n+1)/2 \rfloor$ în cadrul tabloului ordonat crescător (obs: în cazul în care n este par se consideră uneori ca mediană media aritmetică a valorilor aflate în mijlocul tabloului). Fie $x[1..n]$ şi $y[1..n]$ două tablouri ordonate crescător. Să se determine mediana tabloului $z[1..2n]$ care conţine toate elementele din x şi y.

Rezolvare. O primă variantă ar fi să se construiască tabloul z prin interclasare şi să se returneze elementul de pe poziţia n. Este însă suficient să se realizeze o interclasare parţială până când se obţine elementul de poziţia n. În acest scop se poate folosi interclasarea bazată pe valori santinelă mai mari decât elementele din ambele tablouri.

```
interclasare (integer x[1..n], y[1..n])
x[n + 1] ← |x[n]| + |y[n]|; y[n + 1] ← |x[n]| + |y[n]|;
i ← 1; j ← 1;
for k ← 1, n do
   if x[i] < y[j] then z[k] ← x[i]; i ← i + 1
   else z[k] ← y[j]; j ← j + 1
   endif
endfor
return z[n]
```

Este evident că algoritmul are complexitate liniară.

Problema 8 (S) *Problema selectării activităţilor.* Se consideră un set de activităţi care au nevoie de o anumită resursă şi la un moment dat o singură activitate poate beneficia de resursa respectivă (de exemplu activitatea poate fi un examen, iar resursa o sală de examen). Presupunem că pentru fiecare activitate, A_i, se cunoaşte momentul de începere, p_i şi cel de finalizare t_i ($t_i > p_i$). Presupunem că activitatea se desfăşoară în intervalul $[p_i, t_i)$. Două activităţi A_i şi A_j se consideră *compatibile* dacă intervalele asociate sunt disjuncte. Se cere să se selecteze un număr cât mai mare de activităţi compatibile.

Rezolvare. O soluţie a acestei probleme constă într-un subset de activităţi $S = (a_{i_1}, \ldots, a_{i_m})$ care satisfac $[p_{i_j}, t_{i_j}) \cap [p_{i_k}, t_{i_k}) = \emptyset$ pentru orice $j \neq k$. Pentru rezolvarea problemei se poate aplica tehnica alegerii local optimale ("greedy") folosind drept criteriu de selecţie unul dintre următoarele: (i) cea mai mică durată; (ii) cel mai mic moment de începere; (iii) cel mai mic moment de sfârşit; (iv) intervalul care se intersectează cu cele mai puţine alte intervale.

Dintre aceste variante cea pentru care se poate demonstra că asigură obţinerea soluţiei optime este a treia: la fiecare etapă se alege activitatea care se termină cel mai devreme.

Notând cu $A[i].p$, $A[i].t$ momentul de începere respectiv cel de final al activităţii $A[i]$ algoritmul bazat pe strategia greedy poate fi descris astfel:

```
selecţie_activităţi(A[1..n])
A[1..n] ← sortare_crescătoare(A[1..n]) // sortare după timpul de finalizare
S[1] ← A[1]
i ← 2; k ← 1
while i ≤ n do
   if S[k].t ≤ A[i].p
      then k ← k + 1; S[k] ← A[i]; endif
   i ← i + 1
return S[1..k]
```

Se poate demonstra că algoritmul de mai sus conduce la soluţia optimă. După ordonarea crescătoare după timpul de final avem $t_1 \leq t_2 \leq \ldots \leq t_n$. Considerăm o soluţie optimă $O = ((p_{i_1}, t_{i_1}), \ldots, (p_{i_m}, t_{i_m})$. Evident $t_{i_1} \geq t_1$ prin urmare înlocuind A_{i_1} cu A_1 în O, se obţine o soluţie în care există acelaşi număr de activităţi compatibile dar prima activitate este aleasă conform strategiei greedy. Deci problema satisface proprietate alegerii local optimale. O dată aleasă prima activitate, problema se reduce la planificarea optimă a activităţilor compatibile cu prima. Dacă O este soluţia optimă a problemei iniţiale atunci $O' = ((p_{i_2}, t_{i_2}), \ldots, (p_{i_m}, t_{i_m})$ este soluţie optimă a subproblemei la care se ajunge după stabilirea primei activităţi (întrucât toate activităţile sunt compatibile între ele). Astfel problema satisface şi proprietatea substructurii optime.

Să considerăm cazul $A = ((1, 8), (3, 5), (1, 4), (4, 6))$. Sortând A în ordinea crescătoare a duratei activităţilor se obţine ca soluţie: $S = ((3, 5))$ sau $S = ((4, 6))$. Prin selecţie după momentul de începere a activităţilor se obţine $((1, 8))$ sau $((1, 4))$, în schimb prin selecţie după momentul de finalizare se obţine soluţia, $S = ((1, 4), (4, 6))$.

Problema 9 (S) *Problema planificării activităţilor.* Se consideră un set de n prelucrări ce trebuie executate de către un procesor. Durata execuţiei prelucrărilor este aceeaşi (se consideră egală cu 1). Fiecare prelucrare i are asociat un termen final de execuţie, $t_i \leq n$ şi un profit, p_i. Profitul unei prelucrări intervine în calculul profitului total doar dacă prelucrarea este executată (dacă

prelucrarea nu poate fi planificată înainte de termenul final de execuţie profitul este nul). Se cere să se planifice activităţile astfel încât să fie maximizat profitul total (acesta este corelat cu numărul de prelucrări planificate).

Rezolvare. O soluţie constă în stabilirea unui "orar" de execuţie a prelucrărilor $S = (s_1, s_2, \ldots, s_n)$, $s_i \in \{1, \ldots, n\}$ fiind indicele prelucrării planificate la momentul i. Pentru rezolvarea problemei se poate aplica o tehnică de tip greedy caracterizată prin:

- se sortează activităţile în ordinea descrescătoare a profitului;
- fiecare activitate se planifică într-un interval liber cât mai apropiat de termenul final de execuţie.

```
planificare(A[1..n])
A[1..n] ← sortare_descrescătoare(A[1..n]) // sortare după profit (A[i].p)
for i ← 1, n do S[i] ← 0 endfor
for i ← 1, n do
   poz ← A[i].t // termenul final de execuţie
   while S[poz] ≠ 0 ∧ poz ≥ 1 do poz ← poz − 1 endwhile
   if poz ≠ 0 then S[poz] ← i endif // se planifica activitatea
   // altfel activitatea i nu poate fi planificată
return S[1..n]
```

Se observă că dacă pentru fiecare $i \in \{1, \ldots, n\}$ numărul activităţilor care au termenul final cel mult i este cel mult egal cu i ($\text{card}\{j | t_j \leq i\} \leq i$) atunci toate activităţile vor fi planificate, altfel vor exista activităţi ce nu pot fi planificate.

Să considerăm $n = 4$ activităţi având termenele finale: $(2, 3, 4, 2)$ şi profiturile corespunzătoare $(4, 3, 2, 1)$. Atunci aplicând tehnica greedy se obţine soluţia $(4, 1, 2, 3)$. Dacă însă termenele de execuţie sunt: $(2, 4, 1, 2)$ şi aceleaşi profituri se obţine soluţia $(3, 1, 0, 2)$ iar activitatea 4 nu este planificată.

Problema 10 *Problema impachetarii.* (S) Se consideră un set de numere $a_1, a_2, \ldots, a_n$ cu proprietatea că $a_i \in (0, C]$. Se cere să se grupeze în cât mai puţine subseturi (k) astfel încât suma elementelor din fiecare subset să nu depăşească valoarea C. Este cunoscută şi sub numele de "bin-packing problem". Există mai multe variante ce folosesc principiul alegerii greedy însă toate sunt *sub-optimale* (nu garantează obţinerea optimului ci doar a unei soluţii suficient de apropiate de cea optimă).

Varianta 1. Se construiesc succesiv submulţimile: se transferă (în ordinea în care se află în set) în primul subset atâtea elemente cât este posibil, din elementele rămase se transferă elemente în al doilea subset etc. (nu e garantată optimalitatea soluţiei însă s-a demonstrat că $k < 2k_{opt}$, k fiind numărul de subseturi generat prin strategia greedy de mai sus, iar k_{opt} este numărul optim de subseturi).

Varianta 2. Se iniţializează n subseturi vide. Se parcurge setul de numere şi fiecare număr se transferă în primul subset în care "încape". Numărul de subseturi nevide astfel constituite k, are proprietatea $k < 1.7k_{opt}$.

Varianta 3. Dacă se ordonează descrescător setul iniţial şi se aplică varianta 2 se obţine o îmbunătăţire: $k < 11/9k_{opt} + 4$.

Indicaţie. Considerăm că soluţia va fi reprezentată printr-o matrice $R[1..n, 1..n]$ în care linia i corespunde subsetului i iar $R[i, j]$ conţine fie 0 fie indicele unui element din a care trebuie plasat în subsetul i. Pentru a controla modul de completare a subseturilor se pot folosi tablourile $K[1..n]$ ($K[i]$ specifică indicele ultimului element completat în linia i) şi $S[1..n]$ care conţine suma curentă a elementelor de pe linia i.

Problema 11 (L) Fie A o mulţime de valori naturale. Să se descompună mulţimea A în două submulţimi B şi C astfel încât $A = B \cup C$, $B \cap C = \emptyset$ şi suma elementelor din B este cât mai apropiată de suma elementelor din C.

Rezolvare. Se ordonează descrescător elementele lui A. Se transferă primul element din A în B după care se transferă elemente în C până când suma elementelor din C devine cel puţin egală cu suma elementelor din B. Procesul de transfer continuă până când se epuizează elementele din A:

```
descompunere(integer a[1..n])
integer k1, k2, b[1..k1], c[1..k2], S1, S2, i
k1 ← 1; b[k1] ← a[1]; S1 ← a[1]; k2 ← 0; S2 ← 0
a[1..n] ← sortare(a[1..n])
for i ← 2, n
   if S1 < S2 then k1 ← k1 + 1; b[k1] ← a[i]; S1 ← S1 + a[i];
   else k2 ← k2 + 1; c[k2] ← a[i]; S2 ← S2 + a[i];endif
return b[1..k1], c[1..k2]
```

În acest caz tehnica greedy folosită mai sus nu conduce întotdeauna la o soluţie optimă ci doar la una suboptimală. De exemplu aplicând strategia propusă setului de valori $\{10, 8, 7, 5, 4, 2\}$ se obţine descompunerea în $\{10, 5, 4\}$ respectiv $\{8, 7, 2\}$ pe când o soluţie optimă ar fi $\{10, 8\}$ şi $\{7, 5, 4, 2\}$.

Teme seminar/laborator

1. Scrieţi varianta recursivă a algoritmului de la problema 1.
2. Fie $a[1..m]$ şi $b[1..n]$ două tablouri ordonate strict crescător. Propuneţi un algoritm de complexitate liniară pentru determinarea mulţimii elementelor comune celor două tablouri.
3. Se consideră două numere întregi date prin tablourile corespunzătoare descompunerilor lor în factori primi. Să se construiască tablourile similare corespunzătoare celui mai mare divizor comun al celor două valori.
4. Se consideră un caroiaj de dimensiuni $2^k \times 2^k$ (pentru $k = 3$ se obţine o tablă de şah clasică) în care unul dintre pătrăţele este marcat. Propuneţi o strategie bazată pe tehnica divizării care acoperă întreaga tablă (cu excepţia pătrăţelului marcat) cu piese constând din 3 pătrăţele aranjate în forma de L (indiferent de orientare).
5. Propuneţi un algoritm de complexitate medie $\mathcal{O}(n \log n)$ pentru a verifica dacă elementele unui tablou sunt distincte sau nu.
6. Modificaţi algoritmul de la problema 2 astfel încât tabloul z să nu fie construit efectiv.
7. Propuneţi un algoritm de complexitate $\mathcal{O}(n \log n)$ pentru a determina numărul de perechi $(a[i], a[j])$ ale unui tablou $a[1..n]$ (tabloul are elemente distincte) având proprietatea că $i < j$ şi $a[i] > a[j]$.

Indicaţie. Se foloseşte ideea de la sortarea prin interclasare doar că nu se modifică tabloul ci doar se numără de câte ori o valoarea mai mare se află înaintea unei valori mai mici.

8. Se consideră un tablou $a[1..n]$ care conţine numere întregi (atât pozitive cât şi negative). Propuneţi un algoritm de complexitate liniară (şi care foloseşte spaţiu suplimentar de dimesniune constantă) pentru a transforma tabloul iniţial astfel încât toate valorile negative să fie înaintea celor pozitive.