laborator 1

Algoritmi de sortare şi de căutare I

Conţinut

- Sortare prin interschimbare (bubble sort)
- Sortare prin inserţie (insertion sort)
- Sortare prin selecţie a minimului/maximului (selection sort)
- Căutare liniară/secvenţială
- Căutare binară

Referinţe

- **T.H. Cormen, C.E. Leiserson, R.L. Rivest.** *Introducere în algoritmi: cap 1.1 şi 1.2*, Editura Computer Libris Agora, 2000 (şi ediţiile ulterioare)
- **R. Ceterchi.** *Materiale de curs: curs 1*, Anul universitar 2012-2013
- **http://laborator.wikispaces.com/**, *Tema 1*

Problema sortării:

In: Un şir de n numere $A =< a_1, a_2, \ldots, a_n >$.

Out: O permutare $A' =< a'_1, a'_2, \ldots, a'_n >$ a şirului A astfel încât $a'_1 \leqslant a'_2 \leqslant \cdots \leqslant a'_n$.

Sortarea internă vs. sortarea externă

O sortare ce se desfăşoară complet în memoria principală (RAM) se numeşte **internă**. În cazul sortărilor **externe** se folosesc mijloace de stocare externe, precum fişierele memorate pe hard-disk. Deoarece operaţiile de citire şi scriere de date în memoria principală se realizează mai rapid decât în fişiere, algoritmii de sortare externă pot fi încetiniţi de accesul la date. Cum memoria principală disponibilă este limitată, performanţa unui algoritm va fi mai bună atunci când setul de date este suficient de mic pentru a fi stocat în ea.

În cazul algoritmilor de sortare internă, sortarea se face „pe loc", în aceleaşi locaţii, deci este necesar un număr constant (mic) de locaţii suplimentare de memorie.

Exemple de algoritmi de sortare internă:

- **Sortare prin interschimbare (bubble sort)**
- **Sortare prin inserţie (insertion sort)**
- **Sortare prin selecţie a minimului/maximului (selection sort)**
- Sortare rapidă (prin interschimbare folosind partiţii) (quick sort)
- Sortare prin inserţie cu micşorarea incrementului (shell sort)
- Sortare cu ansamble (prin selecţie folosind structuri arborescente de tip ansamblu) (heap sort)

Algoritmii de mai sus au la bază operaţia de comparare a cheilor/elementelor (folosind un operator de comparare, precum: $= < > \leqslant \geqslant \neq$). Printre cele mai utilizate operaţii este şi interschimbarea cheilor (swap), compusă din trei mutări. Mai jos sunt date trei metode de interschimbare a două variabile x şi y.

*Într-un vector $A[1..n]$, valoarile aflate la poziţia $1 \leqslant i \leqslant n$ din A le vom numi **chei**.*

Folosind o variabilă auxiliară temporară:

aux $\leftarrow$ x
x $\leftarrow$ y
y $\leftarrow$ aux

Folosind operaţiile de adunare şi scădere:

x $\leftarrow$ x + y
y $\leftarrow$ x - y
x $\leftarrow$ x - y

Folosind operaţia de xorare (disjuncţie exclusivă) pe biţi :

x $\leftarrow$ x **XOR** y
y $\leftarrow$ x **XOR** y
x $\leftarrow$ x **XOR** y

XOR	0	1
0	0	1
1	1	0

1. Sortarea prin interschimbare directă (bubble sort)

- Complexitate $O(n^2)$

La fiecare pas iterativ i (pasă), se parcurge vectorul de la dreapta la stânga şi se compară câte două elemente succesive `A[j-1]` şi `A[j]`. Dacă acestea se află în ordine crescătoare (`A[j-1]` $\leqslant$ `A[j]`), nu se efectuează nicio schimbare, altfel (`A[j-1] > A[j]`) sunt interschimbate.

La pasul iterativ i, vectorul constă în două părţi:

- **destinaţia** `A[1..i-1]`, ce conţine cele mai mici $i-1$ valori din vector, sortate în iteraţiile anterioare, şi
- **sursa** `A[i..n]`, ce conţine cele $n-i+1$ valori de sortat.

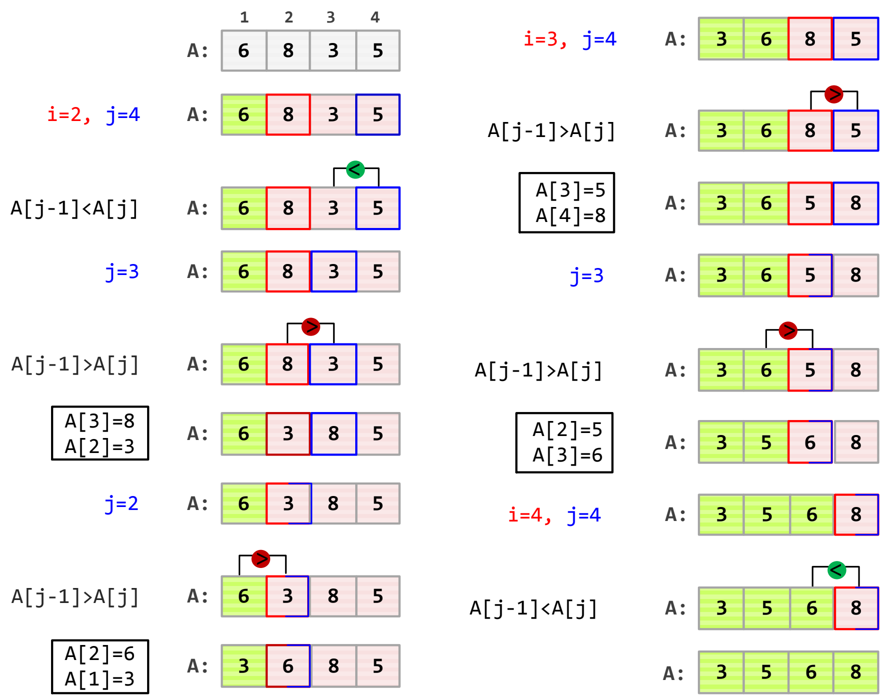
Rezultatul unei iteraţii este „împingerea” minimului din sursă pe `A[i]`. În fiecare pas iterativ dimensiunea destinaţiei va creşte cu un element, iar cea a sursei va scădea cu un element.

Un vector de dimensiune n va fi sortat după $n-1$ iteraţii.

►► SORTEAZĂ–PRIN–INTERSCHIMBARE(A[1..n])

```
for i ← 2 to n do
   for j ← n to i do
      if A[j-1]>A[j] then
         INTERSCHIMBĂ(A[j-1],A[j])
      endif
   endfor
endfor
```

►► EXEMPLIFICARE A MODULUI DE LUCRU AL ALGORITMULUI



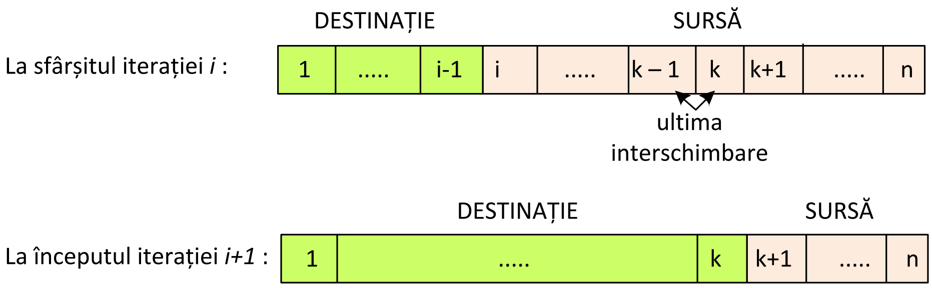
Modificare 1: reducerea numărului de paşi iterativi (reducerea numărului de iteraţii ale ciclului for exterior)

Se poate observa că dacă sursa este sortată, algoritmul nu va face nicio interschimbare. Prin urmare, atunci când nu se produce nicio interschimbare într-o pasă (o traversare a sursei), algoritmul se poate termina (vectorul este sortat).

Indicaţie: se poate folosi o variabilă booleană.

Modificare 2: scurtarea lungimii paselor (reducerea numărului de iteraţii ale ciclului for interior)

Din locul unde s-a efectuat ultima interschimbare (într-o pasă) şi până la destinaţie sursa este sortată. (Dacă nu ar fi sortată porţiunea respectivă, atunci s-ar mai fi efectuat interschimbări.) Mai exact dacă ultima interschimbare a avut loc pentru `j=k`, între elementele `A[k-1]` şi `A[k]`, atunci porţiunea `A[i..k]` este sortată. Acest lucru permite ca la următorul pas iterativ noua sursă să fie $A[k+1..n]$, deci într-o pasă se vor parcurge mai puţine elemente.



►► Sortează–Prin–Interschimbare-2(A[1..n])

```
p ← 0
for i ← 2 to n do
   for j ← n to i do
      if A[j-1]>A[j] then
         Interschimbă(A[j-1],A[j])
         p ← j {poziţia ultimei interschimbări, între p-1 şi p}
      endif
   endfor
   if p = 0 then {nu s-au efectuat interschimbări}
      {algoritmul se termină}
   else
      i ← p {actualizăm destinaţia}
      p ← 0 {reiniţializare}
   endif
endfor
```

2. Sortarea prin inserţie directă

- Complexitate $O(n^2)$

La începutul iteraţiei i vectorul constă în două părţi:

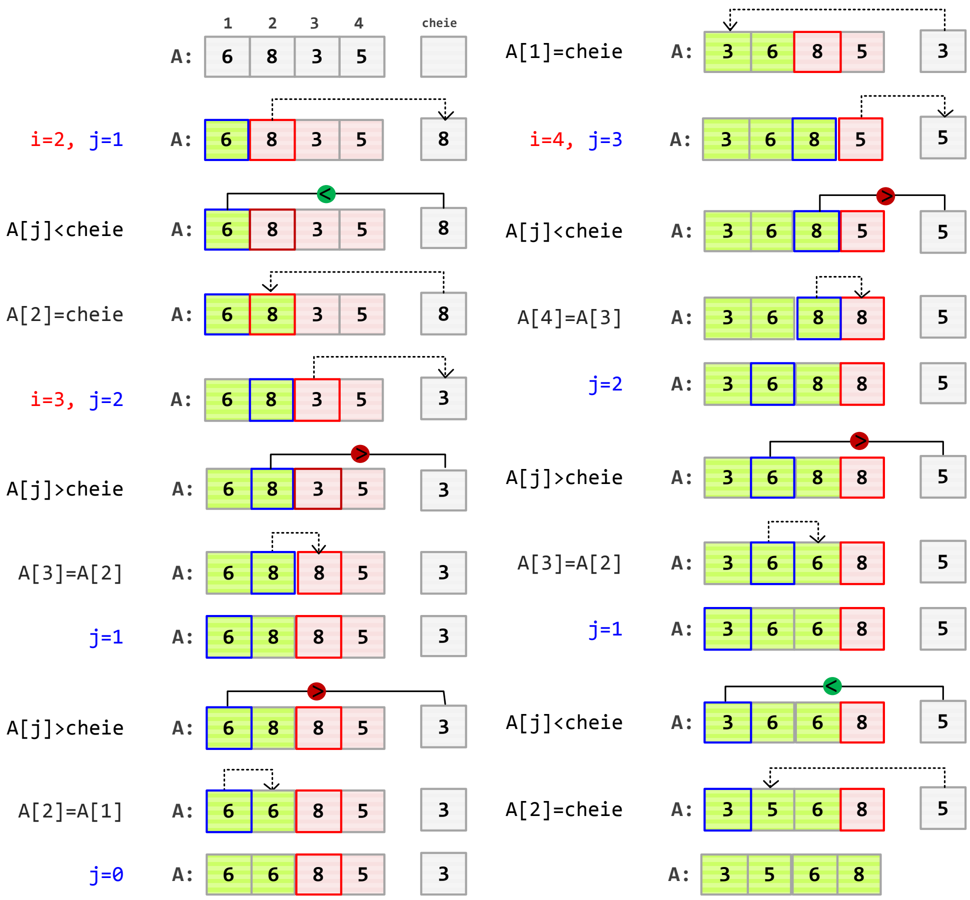
- **destinaţia** `A[1..i-1]`, ce este sortată crescător în iteraţiile anterioare, şi
- **sursa** `A[i..n]`, ce conţine cele $n - i + 1$ valori de sortat.

La fiecare pas iterativ i (pasă) se inserează cel mai din stânga element din sursă (`A[i]`) în destinaţie, astfel încât destinaţia să rămână sortată crescător: se parcurge **destinaţia** de la dreapta la stânga şi dacă elementul curent, `A[j]`, este mai mare decât valoarea de inserat `A[i]`, atunci `A[j]` este mutat cu o poziţie spre dreapta. Astfel, pe poziţia j se creează un loc liber. Reţinând valoarea de inserat `A[i]` într-o variabilă auxiliară, aceasta poate fi suprascrisă de prima mutare efectuată, când `j+1=i`. Când, în timpul unei pase, se găseşte `A[j]` $\leqslant$ `A[i]`, atunci s-a identificat poziţia unde trebuie inserat elementul `A[i]`. În fiecare pas iterativ, dimensiunea destinaţiei va creşte cu un element, iar cea a sursei va scădea cu un element.

Destinaţia nu conţine neapărat cele mai mici $i - 1$ valori din vector.

Un vector de dimensiune n va fi sortat după $n - 1$ iteraţii.

▶▶ Exemplificare a modului de lucru al algoritmului



►► SORTEAZĂ–PRIN–INSERŢIE(A[1..n])

```
1. for i ← 2 to n do
2.    cheie ← A[i]
3.    j ← i-1
4.    while (j>0) and (A[j]>cheie) do
5.       A[j+1] ← A[j]
6.       j ← j-1
7.    endwhile
8.    A[j+1] ← cheie
9. endfor
```

Modificare: utilizarea unei componente marcaj pentru eliminarea unei comparaţii

Se poate renunţa la prima condiţie din ciclul **while** (linia 4), dacă ar exista certitudinea că a doua condiţie va fi la un moment dat, pentru fiecare pas iterativ, falsă. Pentru acest lucru se măreşte dimensiunea vectorului `A` cu o poziţie, inserată la început, care va păstra mereu cheia curentă (valoarea elementului `A[i]`). Astfel, dacă în vectorul destinaţie nu se întâlnesc decât elemente mai mari decât valoarea de inserat, atunci sigur pe prima poziţie (`0`) se va întâlni un element egal.

►► SORTEAZĂ–PRIN–INSERŢIE-2(A[1..n])

```
1. for i ← 2 to n do
2.    A[0] ← A[i]
3.    j ← i-1
4.    while A[j]>A[0] do
5.       A[j+1] ← A[j]
6.       j ← j-1
7.    endwhile
8.    A[j+1] ← A[0]
9. endfor
```

3. Sortarea prin selecţie directă a minimului/maximului

- Complexitate $O(n^2)$

La începutul iteraţiei i vectorul constă în două părţi:

- **destinaţia** `A[1..i-1]`, ce conţine $i-1$ minime puse la locul lor (final) în iteraţiile anterioare, şi
- **sursa** `A[i..n]`, ce conţine cele $n-i+1$ valori de sortat.

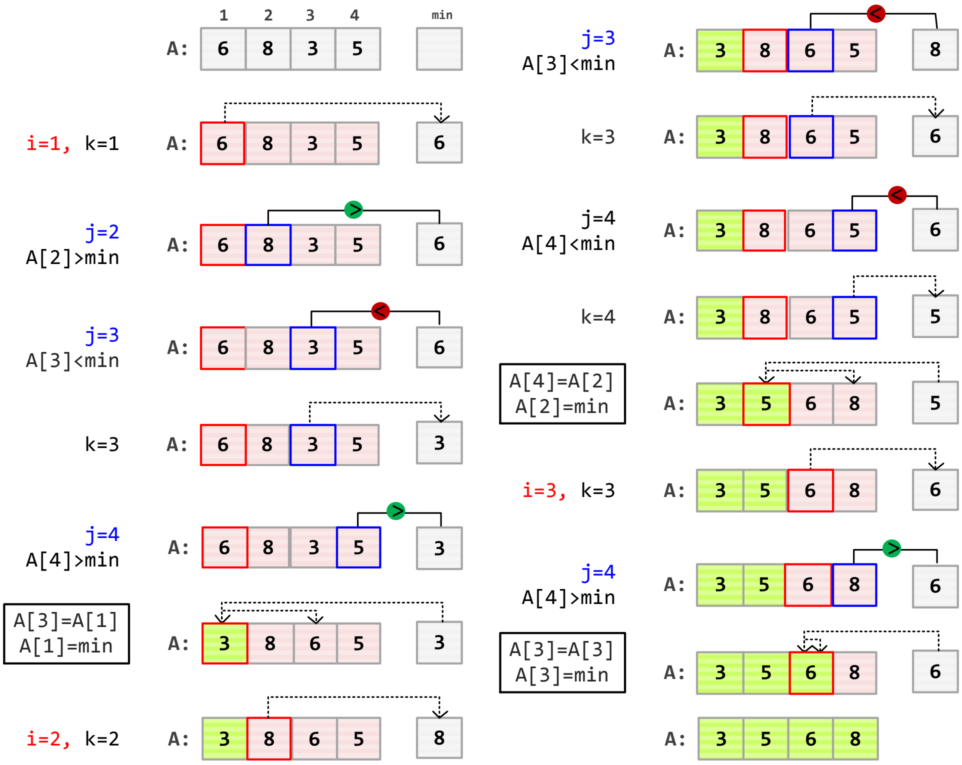
La fiecare pas iterativ i (pasă) se caută secvenţial elementul minim din sursă. Apoi acesta se interschimbă cu elementul `A[i]` (primul din sursă), obţinând pe primele i poziţii din A primele i minime ($i-1$ din destinaţie şi minimul găsit în iteraţia curentă). În fiecare pas iterativ, dimensiunea destinaţiei va creşte cu un element, iar cea a sursei va scădea cu un element.

Un vector de dimensiune n va fi sortat după $n-1$ iteraţii.

►► Sortează–Prin–Seleţie–Minim(A[1..n])

```
for i ← 1 to n-1 do
   k ← i; min ← A[i]
   for j ← i+1 to n do
      if A[j]<min then
         k ← j; min ← A[j]
      endif
   endfor
   A[k] ← A[i]
   A[i] ← min
endfor
```

►► Exemplificare a modului de lucru al algoritmului



Problema căutării:

In: Un şir de n numere $A =< a_1, a_2, \ldots, a_n >$ şi o valoare v.

Out: Un indice i (astfel încât $v = A[i]$) sau valoarea **NIL** dacă v nu apare în A.

4. Căutarea secvenţială/liniară

- Complexitate $O(n)$

Algoritmul de căutare liniară parcurge vectorul `A[1..n]` comparând elementele întâlnite cu valoarea căutată `val`. Dacă nu o găseşte (căutare cu eşec), returnează `loc=0`, altfel returnează `loc`$\in$`[1..n]`, pentru care `A[loc]=val`.

Într-un vector (structură liniară în alocare statică) accesul la oricare componentă se face în timp $O(1)$.

►► CAUTĂ-SECVENŢIAL($A[1..n]$, val, loc)

```
1.  loc ← 0
2.  i ← 1
3.  while (i ⩽ n) and (A[i] ≠ val) do
4.      i ← i+1
5.  endwhile
6.  if i ⩽ n then
7.      loc ← i
8.  endif
```

Modificare: utilizarea unei componente marcaj pentru eliminarea unei comparaţii

Se poate renunţa la prima condiţie din ciclul **while** (linia 4), dacă ar exista certitudinea că a doua condiţie va fi la un moment dat, pentru fiecare pas iterativ, falsă. Pentru acest lucru se măreşte dimensiunea vectorului `A` cu o poziţie, inserată la sfârşit, care va păstra mereu valoarea căutată (`val`). Astfel, dacă în vectorul `A[1..n]` nu se întâlneşte `val`, ciclul **while** se va opri sigur când se ajunge la poziţia $n + 1$, unde va fi reţinută componenta marcaj.

►► CAUTĂ-SECVENŢIAL-2($A[1..n]$, val, loc)

```
1.  A[n+1] ← val
2.  loc ← 1
3.  while A[loc] ≠ val do
4.      loc ← loc+1
5.  endwhile
6.  if loc = n+1 then
7.      {căutare fără succes}
8.  else
9.      {valoarea se află la poziţia loc}
10. endif
```

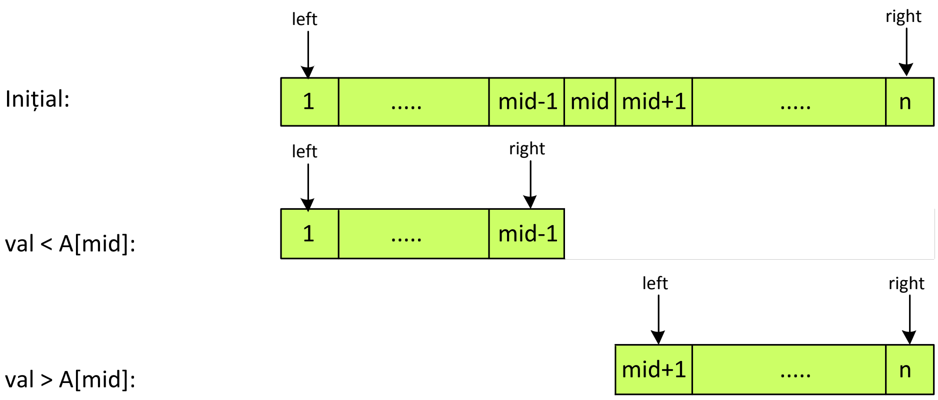
■ Complexitate $O(\log_2 n)$

Pentru a se putea utiliza căutarea binară într-un vector `A[1..n]`, acesta trebuie să fie sortat, deci **`A[1]`** $\leqslant$ **`A[2]`** $\leqslant \cdots \leqslant$ **`A[n]`**.

Se utilizează doi indecşi `left` şi `right` care indică poziţia de început şi de sfârşit a subvectorului din `A`, unde se efectuează căutarea. Iniţial cei doi indecşi sunt chiar `1` şi `n`, deci se referă la întreg vectorul `A`. La fiecare iteraţie (liniile 4-8), se calculează mijlocul subvectorului (`mid`) şi:

- dacă `val = A[mid]`, atunci se iese din ciclul **`while`** şi căutarea se termină cu succes (linia 11),
- dacă `val < A[mid]`, se continuă algoritmul de căutare pe un subvector (mai mic) `[left, mid-1]`,
- dacă `val > A[mid]`, se continuă algoritmul de căutare pe un subvector (mai mic) `[mid+1, right]`.

*Impunerea unei astfel de precondiţii asupra vectorului `A` poate îmbunătăţii şi căutarea secvenţială: condiţia ciclului **`while`** trebuie modificată, astfel încât acesta să nu mai parcurgă toate elementele diferite de `val` până la `val`, ci doar elementele mai mici strict decât `val`. La ieşirea din ciclu se va verifica dacă la poziţia `i` curentă s-a găsit `val`, altfel căutarea este fără succes.*



►► CAUTĂ-BINAR(A[1..n], val, loc)

```
1.   left ← 1; right ← n
2.   mid ← (left+right) div 2
3.   loc ← 0
4.   while ((left ≤ right) and (val ≠ A[mid])) do
5.      if val < A[mid] then
6.         right ← mid-1
7.      else
8.         left ← mid+1
9.      endif
10.     mid ← (left + right) div 2
11.  endwhile
12.  if A[mid] = val then
13.     loc ← mid
14.  else
15.     loc ← 0
16.  endif
```

Variabila `loc` va fi 0 dacă `val` nu a fost găsită în vector, iar dacă `val` a fost găsită, atunci `loc` conţine poziţia la care se află aceasta.

PROBLEME

1. **(2p)** Implementaţi algoritmul de sortare prin interschimbare pentru ordonarea crescătoare a unui vector A de dimensiune n cu elemente numere întregi.
2. Adaptaţi algoritmul de la problema anterioară, astfel încât să se reducă numărul de paşi iterativi şi să se scurteze lungimea paselor.
3. **(2p)** Implementaţi algoritmul de sortare prin inserţie pentru ordonarea crescătoare a unui vector A de dimensiune n cu elemente numere întregi.
4. **(2p)** Implementaţi algoritmul de sortare prin selecţie pentru ordonarea crescătoare a unui vector A de dimensiune n cu elemente numere întregi.
5. **(1p)** Implementaţi algoritmul de căutare liniară a unui element într-un vector A de dimensiune n cu elemente numere întregi. În caz de succes, să se afişeze poziţia i din vector unde a fost găsit prima oară elementul. În caz de eşec să se afişeze un mesaj care să indice că elementul căutat nu se află în vectorul A.
6. Adaptaţi algoritmul de la problema anterioară, astfel încât să afişaţi numărul de apariţii al elementului căutat în vector.
7. **(3p)** Implementaţi algoritmul de căutare binară a unui element într-un vector A de dimensiune n cu elemente numere întregi. În caz de succes, să se afişeze poziţia i din vector unde a fost găsit prima oară elementul. În caz de eşec să se afişeze un mesaj care să indice că elementul căutat nu se află în vectorul A.

■ TERMEN DE PREDARE: **Săptămâna 3 (15-19 octombrie) inclusiv.**

■ DETALII: **Studenţii pot obţine un maxim de 10 puncte. Problemele 1, 3, 4, 5 şi 7 sunt obligatorii. Problemele 2 şi 6 sunt facultative.**