Outline

Cuprins

1 Recapitulare curs anterior

Noţiuni învăţate la cursul precedent

- Alk = limbaj de reprezentare a algoritmilor
  - sintaxă
  - semantică
- calculul unui algoritm
- mărimea valorilor, timpul pentru operaţiil, timpul pentru calcule

2 Problemă rezolvată de un algoritm

Problemă computaţională

O problemă propusă pentru a fi rezolvată de un algoritm poate fi reprezentată prin:

- domeniul problemei - descrie conceptele ce apar în descrierea problemei şi relaţiile dintre acestea
- o pereche (*input*, *output*), unde *input* descrie date de intrare (o instanţă) iar *output* descrie datele de ieşire (răspunsul).

Dacă domeniul este subînţeles sau cunoscut atunci se poate renunţa la descrierea lui.

Notaţie: prin $p \in P$ notăm faptul că p este *instanţă* (componenta *input*) a problemei P şi prin $P(p)$ *rezultatul* (componenta *ouput*).

Exemplu: problema Platou 1/2

Domeniul problemei: Fiind dată o secvenţă $a = (a_0, \ldots, a_{n-1})$ de numere întregi, un *segment* $a[i..j]$ al lui a este secvenţa $(a_i, \ldots, a_j)$, unde $i \leq j$. Dacă $i > j$ putem presupune că segmentul $a[i..j]$ este secvenţa vidă. *Lungimea* unui segment $a[i..j]$ este $j + 1 - i$. Un *platou* este un segment cu toate elementele egale.

Input: O secvenţă $a = (a_0, \ldots, a_{n-1})$ de numere întregi de lungime n ordonată crescător.

Output: Lungimea celui mai lung platou.

Exemplu: problema Platou 2/2

De multe ori perechea (*input*, *output*) poate fi reprezentată cu ajutorul predicatelor. Pentru problema Platou sunt utlizate următoarele predicate:

platou(a, i, j) are loc dacă şi numai dacă $a[i..j]$ este platou: $(\forall k) i \leq k \leq j \implies a_i = a_k$;

ordonatCrescator(a) are loc dacă şi numai dacă $a_0 \leq \ldots \leq a_{n-1}$;

– dacă tabloul este ordonat crescător, atunci *platou*(a, i, j) este echivalent cu $a_i == a_j$ (am presupus $i \leq j$);

Problema Platou poate fi specificată astfel:

Input: $a == (a_0, \ldots, a_{n-1}) \wedge$ *ordonatCrescator*(a).

Output: $q \in \mathbb{Z} \wedge$
$(\exists 0 \leq i \leq j < n)$*platou*$(a, i, j) \wedge q = j + 1 - i \wedge$
$(\forall 0 \leq k \leq \ell < n)$*platou*$(a, k, \ell) \implies q \geq (\ell + 1 - k)$.

Formalizate cu formule din logica de ordinul I, componenta *input* se mai numeşte precondiţie, componenta *output* se numeşte postcondiţie iar perechea (*precondiţie*, *postcondiţie*) se numeşte specificaţie.

Problemă rezolvată de un algoritm

Spunem că un algoritm A rezolvă o problemă P dacă:

- pentru orice instanţă p a lui P, există o configuraţie $\langle A, \sigma_p \rangle$ astfel încât σ_p include structuri date ce descrie p;
- execuţia care pleacă din configuraţia iniţială $\langle A, \sigma_p \rangle$ se termină într-o configuraţie finală $\langle \cdot, \sigma' \rangle$, scriem $\langle A, \sigma_p \rangle \Rightarrow^* \langle ., \sigma' \rangle$; şi
- σ' include structuri de date ce descriu $P(p)$.

Observaţie. Definiţia de mai sus presupune că A este determinist, astfel că există o unică execuţie care pleacă din σ_p. Instrucţiunile Alk studiate până acum sunt deterministe. Vom discuta mai târziu despre programe nedeterministe.

Problemă rezolvată de un algoritm, mai formal

O aserţiune ϕ este un o formulă cu predicate descrisă cu variabilele care apar în algoritm.

Notăm cu $\sigma \models \phi$ faptul că valorile variabilelor în σ satisfac ϕ.
Exemplu: dacă $\sigma = \mathtt{x} \mapsto 3\ \mathtt{y} \mapsto 5$, atunci $\sigma \models 2 * \mathtt{x} > \mathtt{y}$ şi $\sigma \not\models \mathtt{x} + \mathtt{y} < 0$.

Presupunem că problema P este specificată de (*pre*, *post*) (i.e., (*precondiţie*, *postcondiţie*)).

A rezolvă P, specificată de (*pre*, *post*), dacă pentru orice stare σ cu proprietatea $\sigma \models$ *pre* există σ' astfel încât $\langle A, \sigma \rangle \Rightarrow^* \langle ., \sigma' \rangle$ şi $\sigma' \models$ *post*.

Rezolvare versus Corectitudine

Dacă un algoritm A rezolvă o problemă specificată prin (*precondiţie*, *postcondiţie*), se mai spune că algoritmul A este corect (relativ la specificaţia (*precondiţie*, *postcondiţie*)).

Cele două noţiuni nu sunt chiar echivalente.

Există două tipuri de corectitudine:

corectitudine totală: pentru orice stare σ, dacă $\sigma \models pre$ atunci există σ' astfel încât $\langle A, \sigma\rangle \Rightarrow^* \langle ., \sigma'\rangle$ şi $\sigma' \models post$.

corectitudine parţială: pentru orice stare σ, dacă $\sigma \models pre$ şi dacă există σ' astfel încât $\langle A, \sigma\rangle \Rightarrow^* \langle ., \sigma'\rangle$, atunci $\sigma' \models post$.

La corectitudinea totală trebuie dovedită şi terminarea execuţiei, la cea parţială se cere ca starea finală să satisfacă *postcondiţia* numai în cazurile când execuţia care pleacă din σ se termină.

Rezolvarea este echivalentă cu corectitudinea totală.

Algoritmul PlatouAlg

- Conceptele din domeniul problemei reprezentate ca structuri de date: Presupunem că secvenţa a este reprezentată de tabloul $\mathtt{a} \mapsto [a_0 \dots a_{n-1}]$. Un algoritm propus pentru a rezolva problema Platou este:

```
lg = 1;
i = 1;
while (i < n) {
  if (a[i] == a[i - lg]) lg = lg+1;
  i = i + 1;
}
```

Relaţia dintre PlatouAlg şi specificarea problemei Platou

Ca să arătăm că PlatouAlg rezolva Platou, va trebui să arătăm că orice execuţie care pleacă din *configuraţia iniţială*:

$\langle PlatouAlg,\ \mathtt{n} \mapsto n\ \mathtt{a} \mapsto [a_0, \dots, a_{n-1}]\rangle$

cu $a_0 \le \dots \le a_{n-1} \wedge n \ge 1$ (i.e., satisface precondiţia)

se opreşte în *configuraţia finală*:

$\langle \cdot,\ \mathtt{n} \mapsto n\ \mathtt{a} \mapsto [a_0, \dots, a_{n-1}]\ \mathtt{i} \mapsto n\ \mathtt{lg} \mapsto q\rangle$

şi q *reprezintă lungimea celui mai lung platou din* $\mathtt{a}$:

$$(\exists 0 \le i \le j < n) platou(a, i, j) \wedge q = j + 1 - i \wedge$$
$$(\forall 0 \le k \le \ell < n) platou(a, k, \ell) \implies q \ge (\ell + 1 - k)$$

(i.e., satisface postcondiţia)

Cum demonstrăm corectitudinea?

Cum dovedim că algoritmul PlatouAlg rezolvă într-adevăr problema Platou?

O posibilă soluţie:

- se arată că la începutul şi la sfârşitul buclei while are loc proprietatea "`lg` reprezintă lungimea celui mai lung platou din segmentul `a`$[0..i-1]$", i.e.

$(\exists 0 \leq i_0 \leq j_0 < i) platou(a, i_0, j_0) \wedge \texttt{lg} = j_0 + 1 - i_0 \wedge$
$(\forall 0 \leq k \leq \ell < i) platou(a, k, \ell) \implies \texttt{lg} \geq (\ell + 1 - k)$

- această prorietate se numeşte invariant de buclă

- la sfârşitul buclei while are loc invariantul şi negaţia condiţiei ($i \geq n$); dacă arătăm că şi $i \leq n$ este invariant, atunci la sfârşitul buclei while vom avea $i = n$ şi invariantul devine exact postcondiţia.

Cum demonstrăm invariantul?

Distingem două cazuri:

1. $j_0 = i-1$ (cel mai lung platou din $a[0..i-1]$ se termină în $i-1$). Distingem două subcazuri:

1.1 are loc $platou(a, i_0, i)$ (i.e. `a[i] == a[i - lg]`), platoul $a[i_0..j_0]$ se măreşte cu o unitate şi $a[i_0..i]$ devine cel mai lung platou din $a[0..i]$ (de ce?); de aceea `lg` este incrementat;

1.2 nu are loc $platou(a, i_0, i)$, adică $a[i-1] < a[i]$; cel mai lung platou din $a[0..i-1]$ va fi şi cel mai lung platou din $a[0..i]$; de aceea `lg` nu se modifică;

2. cel mai lung platou din $a[0..i-1]$ NU se termină în $i-1$. Rezultă că lungimea platoului care se termină în $i-1$ este $<$ `lg`, de unde lungimea platoului care se termină în i este $\leq$ `lg`; de aceea `lg` nu se modifică.

Problemă rezolvabilă (calculabilă)

O problemă P este rezolvabilă (calculabilă) dacă există un algoritm A care rezolvă P.

O problemă P este nerezolvabilă (necalculabilă) dacă NU există un algoritm A care rezolvă P.

Problemă de decizie

O problemă de decizie este o problemă P la care răspunsul (outputul) este de forma "DA" sau "NU" (echivalent, "true" sau "false"). Mai precis, pentru orice instanţă $p \in P$, $P(p) \in \{"DA", "NU"\}$ ($P(p) \in \{"true", "false"\}$).

O problemă de decizie este reprezentată în general printr-o pereche *(instance, question)* (sau (instanţă, întrebare)).

O problemă decidabilă este o problemă de decizie rezolvabilă.

O problemă nedecidabilă este o problemă de decizie nerezolvabilă.

Sunt toate problemele computaţionale rezolvabile (decidabile)?

La începutul secolului 20, matematicienii credeau că da.

În 1931, Kurt Gödel a şocat dovedind că aceasta este imposibil.

El a demonstrat că dacă avem un sistem destul de puternic şi cu o comportare bine-definită ce include raţionamentul matematic, atunci vor exista afirmaţii care nu pot fi demonstrate ca fiind adevărate cu acest sistem, chiar dacă în realitate ele sunt adevărate. (Celebra teorema de incompletitudine alui Gödel).

Câţiva ani mai târziu, Alan Turing a demonstrat acelşi lucru utilizând noţiunea de algoritm (maşină Turing).

Program universal

Pornind de la ideea de maşină Turing universală, putem defini noţiunea de program (algoritm) universal: acesta are la intrare un program (algoritm) A şi o intrare x (echivalent cu având la intare configuraţia $\langle A, \sigma_x \rangle$), simulează activitatea lui A pentru intrarea x (plecând din configuraţia $\langle A, \sigma_x \rangle$).

Programele (algoritmii) pot fi intrări pentru alţi algoritmi!

Exemplu de problemă nerezolvabilă/nedecidabilă

Problema opririi:
Instance: O configuraţie $\langle A, \sigma_0 \rangle$, unde A este un algoritm şi σ_0 codificarea unei intrări pentru A.
Question: Execuţia care pleacă din configuraţia iniţială $\langle A, \sigma_0 \rangle$ este finită?

Teoremă [Turing, 1936]

Nu există un algoritm care să rezolve Problema opririi.

Ideea de demonstrare 1/2

Prin reducere la absurd.

Presupunem că există un algoritm H care rezolvă problema opririi. Un apel al algoritmului H pentru intrarea A, x este notat prin $H(A, x)$.

Construim un alt aloritm, care apelează H:

```
NewH(A) {
  if H(A, A) return true;
  else return false;
}
```

şi un alt program care apelează NewH:

```
HaltsOnSelf (A) {
  if NewH(A) while (true) {}
  else return false;
}
```

Ideea de demonstrare 2/2

Ce se întâmplă când se apelează `HaltsOnSelf(HaltsOnSelf)`?

Execuţia lui `HaltsOnSelf(HaltsOnSelf)` nu se termină; rezultă că `NewH(HaltsOnSelf)` întoarce `true`, care implică `H(HaltsOnSelf,HaltsOnSelf)` întoarce `true`. Contradicţie.

Execuţia lui `HaltsOnSelf(HaltsOnSelf)` se termină şi întoarce `true`; rezultă că `NewH(HaltsOnSelf)` întoarce `false`, care implică `H(HaltsOnSelf,HaltsOnSelf)` întoarce `false`. Contradicţie din nou.

Rezolvarea teoremei de mai sus este strâns legată de următorul paradox logic. "Există un oraş cu un bărbier care bărbiereşte pe oricine ce nu se bărbiereşte singur. Cine bărbiereşte pe bărbier?"

Alte exemple de probleme nedecidabile

Problema echivalenţei programelor.

Totality: dacă un program dat se opreşte pentru toate intrările.

Problema corectitudinii totale.

Problema a 10-a a lui Hilbert

...

Teorema lui Rice 1/2

- proprietate (a algorimilor) = o submulţime P de algoritmi (e.g., scrişi în Alk)
- proprietate netrivială $\emptyset \neq P \neq$ mulţimea tuturor algoritmilor
- proprietate funcţională (extensională) = se referă doar la intrări/ieşiri

  formal:
  $\sigma \cong \sigma'$ ddacă reprezintă aceeaşi intrare (input) sau aceeaşi ieşire (output)
  $A_1 \equiv A_2$ ddacă $\forall\ \sigma_1 \cong \sigma_2$, $\langle A_1, \sigma_1\rangle \Rightarrow^* \langle ., \sigma_1'$ $\iff$ $\langle A_2, \sigma_2 > \Rightarrow^* \langle ., \sigma_2'\rangle$ şi $\sigma_1' \cong \sigma_2'$
  P funcţională ddacă $A_1 \cong A_2$ implică $A_1 \in P$ $\iff$ $A_2 \in P$
- exemple de proprietati funcţionale
  - algoritmii care se opresc întodeauna
  - algoritmii care întorc o constantă dată

Teorema lui Rice 2/2

- exemple de proprietati nefuncţionale (intensionale)
  - algorimii care execută exact n paşi (n dat)
  - algorimii cu acelaşi număr n de instrucţiuni (n dat)

Teoremă [Rice, 1953]
Orice proprietate (a algoritmilor) netrivială şi funcţională este nedecidabilă.

Parţial rezolvabil (calculabil, decidabil)

O problemă de decizie este parţial calculabilă, sau semidecidabilă, dacă există un algoritm care se opreşte cu răspunsul "DA" pentru toate intrările pentru care răspunsul correct este "DA".

Este Problema opririi semidecidabilă?

3 Complexitatea unui algoritm

Timpul unei execuţii

Fie $E = \langle A_0, \sigma_0 \rangle \Rightarrow \cdots \Rightarrow \langle A_n, \sigma_n \rangle$ o execuţie. Timpul consumat de această execuţie este suma timpilor paşilor de execuţie:

$$time_d(E) = \sum_{i=0}^{n-1} time_d(\langle A_i, \sigma_i \rangle \Rightarrow \langle A_{i+1}, \sigma_{i+1} \rangle)$$

unde $d \in \{log, unif\}$

Reamintim că timpul de execuţie al unui pas $time(\langle A_i, \sigma_i \rangle \Rightarrow \langle A_{i+1}, \sigma_{i+1} \rangle)$ a fost precizat când s-au descris evaluările expresiilor şi semantica instrucţiunilor.

Demo cu versiunea de Alk care calculează şi timpii uniform şi logaritmic.

Timpul necesar execuţiei unei instanţe

Un algoritm determinist este un algoritm A cu proprietatea că pentru orice configuraţie $\langle A_i, \sigma_i \rangle$ accesibilă dintr-o configuraţie iniţială $\langle A_0, \sigma_0 \rangle$ (i.e., $\langle A_0, \sigma_0 \rangle \Rightarrow^* \langle A_i, \sigma_i \rangle$) există cel mult of configuraţie succesoare, i.e., cel mult o $\langle A', \sigma' \rangle$ cu proprietatea $\langle A_i, \sigma_i \rangle \Rightarrow \langle A', \sigma' \rangle$.

Fie P o problemă, p o instanţă a lui P şi A un algoritm determinist care rezolvă P. Execuţia $\langle A, \sigma_p \rangle \Rightarrow \cdots$ este unică (deoarece A este determinist) şi finită (deoarece A rezolvă P); notăm această execuţie cu E_p.

Timpul necesar algoritmului A pentru a rezolva instanţa p este

$$time_d(A, p) = time_d(E_p)$$

unde $d \in \{log, unif\}$

Notaţii echivalente: $time_{A,d}(p)$ sau $time_A(p)$ (când d este subînţeles) sau $time(p)$ (când ambele A şi d sunt subînţelese).

4 Complexitatea în cazul cel mai nefavorabil

Dimensiunea unei instanţe

Dimensiunea unei stări σ:

$$size_d(\sigma) = \sum_{\mathbf{x} \mapsto v \in \sigma} size_d(v)$$

Dimensiunea unei configuraţii:

$$size_d(\langle A, \sigma \rangle) = size_d(\sigma)$$

Fie P o problemă rezolvată de A.

Dimensiunea lui $p \in P$:

$$size_d(p) = size_d(\langle A, \sigma_p \rangle) \quad (= size(\sigma_p))$$

unde $d \in \{log, unif\}$.

Complexitatea timp în cazul cel mai nefavorabil

Fie P o problemă şi A un algoritm determinist care rezolvă P şi $d \in \{log, unif\}$.

Grupăm instanţele p ale problemei P în clase de echivalenţă: *p şi p' sunt în aceeaşi clasă dacă* $size_d(p) = size_d(p')$.

Un număr întreg pozitiv n *poate fi privit ca fiind clasa de ecivalenţă a instanţelor de mărime* n.

Complexitatea timp în cazul cel mai nefavorabil este timpul dat de execuţia cu timpul cel mai mare peste configuraţiile iniţiale date de instanţele din aceeaşi clasă de echivalenţă:

$$T_{A,d}(n) = \max\{time_d(A, p) \mid p \in P, size_d(p) = n\}$$

Complexitatea spaţiu

Fie $E = \langle A_0, \sigma_0\rangle \Rightarrow \cdots \Rightarrow \langle A_n, \sigma_n\rangle$ o execuţie şi $d \in \{log, unif\}$.

Spaţiul consumat de această execuţie este dat de maximul dintre dimensiunile configuraţiilor din E:

$$space_d(E) = \max_{i=0}^{n} size_d(\langle A_i, \sigma_i\rangle)\}$$

Spaţiul necesar algoritmului A pentru a rezolva instanţa p este

$$space_d(A, p) = space_d(E_p)$$

Complexitatea spaţiu în cazul cel mai nefavorabil este calculată într-un mod similar complexităţii timp pentru cazul cel mai nefavorabil:

$$S_{A,d}(n) = \max\{space_d(A, p) \mid size_d(p) = n\}$$

Calcul complexităţii în cazul cel nefavorabil 1/3

- A este o expresie E care nu include apeluri de funcţii (algoritmi): $T_{A,d}(n) = time_d([\![E]\!](\sigma_p))$ pentru $p \in P$ cu $size_d(p) = n$
- A este o atribuire $X = E$; $T_{A,d}(n) = T_{E,d}(n)$
- A este `if (`E`)` S_1 `else` S_2: $T_{A,d}(n) = \max\{T_{S_1,d}(n), T_{S_2,d}(n)\} + T_{E,d}(n)$
- A o compunere secvenţială S_1 S_2: $T_{A,d}(n) = T_{S_1,d}(n) + T_{S_2,d}(n)$

Calcul complexităţii în cazul cel nefavorabil 2/3

- A este o instrucţiune iterativă (e.g.,`while, for`): de multe ori se poate calcula doar o aproximare
  - soluţia 1 (o aproximare mai fidelă):
    - se calculează numărul maxim de iteraţii $nMax$
    - se calculează complexitatea în cazul cel nefavorabil pentru fiecare iteraţie, fie acestea $T_1, \ldots, T_{nMax}$
    - se ia $T_{A,d}(n) = T_1 + \ldots + T_{nMax}$
  - soluţia 2 (approximare mai grosieră):

- se calculează numărul maxim de iteraţii $nMax$
- se calculează complexitatea în cazul cel nefavorabil pentru iteraţia cu timpul (în cazul cel nefavorabil) cel mai mare, fie acesta T_{Max}
- se ia $T_{A,d}(n) = nMax \times T_{Max}$

Calcul complexităţii în cazul cel nefavorabil 3/3

- Atenţie la liste, mulţimi, ...:

```
s = 0;
for(i = 0; i < l.size(); ++i)  // l is a linear list
  s = s + l.at(i);

s = emptySet;
forall x in a   // a is a set
  if (x % 2 == 0) s = s U singletonSet(x);
```

- Apel de funcţii (algoritmi):
  - se estimează dimensiunea argumentelor în funcţie de dimensiunea instanţei n
  - se utilizează complexitatea în cazul cel nefavorabil a algoritmului apelat, calculată cu dimensiunea argumentelor estimată

Calculul complexităţii în cazul cel nefavorabil în practică

- deobicei numai *costul uniform* este calculat
- trebuie precizată *dimensiunea unei instanţe*
- numai *o parte din operaţii* sunt considerate (e.g., comparaţii, atribuiri)
- cel mai important (şi uneori) dificil este *identificarea cazului cel mai nefavorabil*
- se calculează *aproximaţii ale lui* $T_{A,d}(n)$ utilizând notaţiile $O(f(n))$, $\Omega(f(n))$,$\Theta(f(n))$

Reamintim:

$O(f(n)) = \{g(n) \mid (\exists c > 0, n_0 \geq 0)(\forall n \geq n_0)|g(n)| \leq c \cdot |f(n)|\}$
$\Omega(f(n)) = \{g(n) \mid (\exists c > 0, n_0 \geq 0)(\forall n \geq n_0)|g(n)| \geq c \cdot |f(n)|\}$
$\Theta(f(n)) = \{g(n) \mid (\exists c_1, c_2 > 0, n_0 \geq 0)(\forall n \geq n_0)c_1 \cdot |f(n)| \leq |g(n)| \leq c_2 \cdot |f(n)|\}$

Exemplul 1

input: $n, (a_0, \ldots, a_{n-1}), z$ numere întregi.

output: $poz = \begin{cases} \min\{i \mid a_i = z\} & \text{dacă } \{i \mid a_i = z\} \neq \emptyset, \\ -1 & \text{altfel.} \end{cases}$

```
i = 0;
while (a[i] != z) and (i < n-1)
  i = i+1;
if (a[i] == z) poz = i;
else poz = -1;
```

Analiza Exemplul 1

- tip de cost: uniform
- dimensiunea unei instanţe; n
- operaţiile măsurate: comparaţii în care apar elemente ale tabloului
- cazul cel mai nefavorabil: z apare prima dată pe poziţia $n-1$ sau nu apare în a
- o bucla while: 1 comparaţie
- numărul de iteraţii pentru cazul cel mai nefavorabil: $n-1$
- timpul de execuţie pentru cazul cel mai nefavorabil: $T_A(n) = (n-1)+1 = n$

Exemplul 2

input: $n, (a_0, \ldots, a_{n-1})$ numere întregi.
output: $max = \max\{a_i \mid 0 \leq i \leq n-1\}$.

```
max = a[0];
for (i = 1; i < n; i++)
  if (a[i] > max)
    max = a[i];
```

Discuţia pe tablă.

Exemplul 3

input: $n, (a_0, \ldots, a_{n-1})$ numere întregi.
output: $(a_{i_0}, \ldots, a_{i_{n-1}})$ unde $(i_0, \ldots, i_{n-1})$ este o permutare a şirului $(0, \ldots, n-1)$ şi $a_{i_j} \leq a_{i_{j+1}}, \forall j \in \{0, \ldots, n-2\}$.

```
for (k = 1; k < n;  k++) {
  temp = a[k];
  i = k - 1;
  while (i >= 0 and a[i] > temp) {
    a[i+1] = a[i];
    i = i-1;
  }
  a[i+1] = temp;
}
```

Discuţia pe tablă.

Exemplul 4

input: $n, (a_0, \ldots, a_{n-1}), z$ numere întregi;
secvenţa $(a_0, \ldots, a_{n-1})$ este sortată crescător,

output: $poz = \begin{cases} k \in \{i \mid a_i = z\} & \text{dacă } \{i \mid a_i = z\} \neq \emptyset, \\ -1 & \text{altfel.} \end{cases}$

```
  istg = 0;
  idr = n - 1;
  while (istg <= idr ) {
    imed = (istg + idr) / 2;
    if (a[imed] == z)
      return imed
    else if (a[imed] > z)
      idr = imed-1;
    else
      istg = imed + 1;
  }
return -1
```

Discuţia pe tablă.