

Outline

Cuprins

1	Problemă rezolvată de un algoritm	1
2	Complexitatea unui algoritm	6
3	Complexitatea în cazul cel mai nefavorabil	7

1 Problemă rezolvată de un algoritm

Problemă computațională

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

- domeniul problemei - descrie conceptele ce apar în descrierea problemei și relațiile dintre acestea
- o pereche (*input*, *output*), unde *input* descrie date de intrare (o instanță) iar *output* descrie datele de ieșire (răspunsul).

Dacă domeniul este subînțeles sau cunoscut atunci se poate renunța la descrierea lui.

Notăție: prin $p \in P$ notăm faptul că p este *instanță* (componenta *input*) a problemei P și prin $P(p)$ *rezultatul* (componenta *output*).

Exemplu: problema Platou 1/2

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

Input: O secvență $a = (a_0, \dots, a_{n-1})$ 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 utilizate următoarele predicate:

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

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

– 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: $a == (a_0, \dots, a_{n-1}) \wedge \text{ordonatCrescator}(a)$.

Output: $q \in \mathbb{Z} \wedge$

$$\begin{aligned} & (\exists 0 \leq i \leq j < n) \text{platou}(a, i, j) \wedge q = j + 1 - i \wedge \\ & (\forall 0 \leq k \leq \ell < n) \text{platou}(a, k, \ell) \implies q \geq (\ell + 1 - k). \end{aligned}$$

Formalizate cu formule din logica de ordinul I, componenta *input* se mai numește precondiție, componenta *output* se numește postcondiție iar perechea (*precondiție*, *postcondiție*) se numește specificație.

Problemă rezolvată de un algoritm

Spunem că un algoritm A rezolvă o problemă P dacă:

- pentru orice instanță p a lui P , există o configurație $\langle A, \sigma_p \rangle$ astfel încât σ_p include structuri date ce descrie p ;
- execuția care pleacă din configurația inițială $\langle A, \sigma_p \rangle$ se termină într-o configurație finală $\langle \cdot, \sigma' \rangle$, scriem $\langle A, \sigma \rangle \Rightarrow^* \langle \cdot, \sigma' \rangle$; și
- σ' include structuri de date ce descriu $P(p)$.

Observație. Definiția de mai sus presupune că A este determinist, astfel că există o unică execuție care pleacă din σ_p . Instrucțiunile Alk studiate până acum sunt deterministe. Vom discuta mai târziu despre programe nedeterministe.

Problemă rezolvată de un algoritm, mai formal

O aserțiune 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 $\sigma = x \mapsto 3 \ y \mapsto 5$, atunci $\sigma \models 2 * x > y$ și $\sigma \not\models x + y < 0$.

Presupunem că problema P este specificată de $(pre, post)$ (i.e., (*precondiție*, *postcondiție*)).

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

Rezolvare versus Corectitudine

Dacă un algoritm A rezolvă o problemă specificată prin $(precondiție, postcondiție)$, se mai spune că algoritmul A este corect (relativ la specificația $(precondiție, postcondiție)$).

Cele două noțiuni nu sunt chiar echivalente.

Există două tipuri de corectitudine:

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

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

La corectitudinea totală trebuie dovedită și terminarea execuției, la cea parțială se cere ca starea finală să satisfacă *postcondiția* numai în cazurile când execuția care pleacă din σ se termină.

Rezolvarea este echivalentă cu corectitudinea totală.

Algoritmul PlatouAlg

Presupunem că secvența a este reprezentată de tabloul $\mathbf{a} \mapsto [a_0 \dots a_{n-1}]$. 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;
}
```

Relația dintre PlatouAlg și specificarea problemei Platou

Ca să arătăm că PlatouAlg rezolva Platou, va trebui să arătăm că orice execuție care pleacă din *configurația inițială*:

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

cu $a_0 \leq \dots \leq a_{n-1} \wedge n \geq 1$ (i.e., satisface precondiția)

se oprește în *configurația finală*:

$\langle \cdot, \mathbf{n} \mapsto n \ \mathbf{a} \mapsto \{0 \mapsto a_0 \dots n-1 \mapsto a_{n-1}\} \ \mathbf{i} \mapsto n \ \mathbf{lg} \mapsto q \rangle$

și q reprezintă lungimea celui mai lung platou din \mathbf{a} :

$(\exists 0 \leq i \leq j < n) \text{platou}(a, i, j) \wedge q = j + 1 - i \wedge$

$(\forall 0 \leq k \leq \ell < n) \text{platou}(a, k, \ell) \implies q \geq (\ell + 1 - k)$

(i.e., satisface postcondiția)

Cum demonstrăm corectitudinea?

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

O posibilă soluție:

- se arată că la începutul și la sfârșitul buclei while are loc proprietatea "lg reprezintă lungimea celui mai lung platou din segmentul $\mathbf{a}[0..i-1]$ ", i.e.

$(\exists 0 \leq i_0 \leq j_0 < i) \text{platou}(a, i_0, j_0) \wedge \mathbf{lg} = j_0 + 1 - i_0 \wedge$

$(\forall 0 \leq k \leq \ell < i) \text{platou}(a, k, \ell) \implies \mathbf{lg} \geq (\ell + 1 - k)$

- această proprietate se numește invariant de buclă

- la sfârșitul buclei while are loc invariantul și negația condiției ($i \geq n$); dacă arătăm că și $i \leq n$ este invariant, atunci la sfârșitul buclei while vom avea $i = n$ și invariantul devine exact postcondiția.

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 mărește cu o unitate și $a[i_0..i]$ 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 și 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 instanță $p \in P$, $P(p) \in \{"DA", "NU"\}$ ($P(p) \in \{"true", "false"\}$).

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

O problemă decidabilă este o problemă de decizie rezolvabilă.

O problemă nedecidabilă este o problemă de decizie nerezolvabilă.

Sunt toate problemele computaționale rezolvabile (decidabile)?

La începutul secolului 20, matematicienii credeau că da.

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

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

Câțiva ani mai târziu, Alan Turing a demonstrat acelși lucru utilizând noțiunea de algoritm (mașină Turing).

Program universal

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

Programele (algoritmii) pot fi intrări pentru alți algoritmi!

Exemplu de problemă nerezolvabilă/nedecidabilă

Problema opririi:

Instance: O configurație $\langle A, \sigma_0 \rangle$, unde A este un algoritm și σ_0 codificarea unei intrări pentru A .

Question: Execuția care pleacă din configurația inițială $\langle A, \sigma_0 \rangle$ este finită?

Teoremă

Nu există un algoritm care să rezolve Problema opririi.

Ideea de demonstrație 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 algoritm, care apelează H :

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

și un alt program care apelează NewH:

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

Ideea de demonstrație 2/2

Ce se întâmplă când se apelează $\text{HaltsOnSelf}(\text{HaltsOnSelf})$?

Execuția lui $\text{HaltsOnSelf}(\text{HaltsOnSelf})$ nu se termină; rezultă că $\text{NewH}(\text{HaltsOnSelf})$ întoarce **true**, care implică $H(\text{HaltsOnSelf}, \text{HaltsOnSelf})$ întoarce **true**. Contradicție.

Execuția lui $\text{HaltsOnSelf}(\text{HaltsOnSelf})$ se termină și întoarce **true**; rezultă că $\text{NewH}(\text{HaltsOnSelf})$ întoarce **false**, care implică $H(\text{HaltsOnSelf}, \text{HaltsOnSelf})$ întoarce **false**. Contradicție din nou.

Rezolvarea teoremei de mai sus este strâns legată de următorul paradox logic. “Există un oraș cu un bărbier care bărbiereste pe oricine ce nu se bărbiereste singur. Cine bărbiereste pe bărbier?”

Alte exemple de probleme nedecidabile

Problema echivalenței programelor.

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

Problema corectitudinii totale.

Problema a 10-a a lui Hilbert

...

Parțial rezolvabil (calculabil, decidabil)

O problemă de decizie este parțial calculabilă (semidecidabilă) dacă există un algoritm care se oprește cu răspunsul "DA" pentru toate intrările pentru care răspunsul corect este "DA".

Este Problema opririi semidecidabilă?

2 Complexitatea unui algoritm

Timpul unei execuții

Fie $E = \langle A_0, \sigma_0 \rangle \Rightarrow \dots \Rightarrow \langle A_n, \sigma_n \rangle$ o execuție. Timpul consumat de această execuție este suma timpilor pașilor de execuție:

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

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

Reamintim că timpul de execuție 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 și semantica instrucțiunilor.

Demo cu versiunea de Alk care calculează și timpii uniform și logaritmici.

Timpul necesar execuției unei instanțe

Un algoritm determinist este un algoritm A cu proprietatea că pentru orice configurație $\langle A_i, \sigma_i \rangle$ accesibilă dintr-o configurație inițială $\langle A_0, \sigma_0 \rangle$ (i.e., $\langle A_0, \sigma_0 \rangle \Rightarrow^* \langle A_i, \sigma_i \rangle$) există cel mult o configurație 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 instanță a lui P și A un algoritm determinist care rezolvă P . Execuția $\langle A, \sigma_p \rangle \Rightarrow \dots$ este unică (deoarece A este determinist) și finită (deoarece A rezolvă P); notăm această execuție cu E_p .

Timpul necesar algoritmului A pentru a rezolva instanța p este

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

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

3 Complexitatea în cazul cel mai nefavorabil

Dimensiunea unei instanțe

Dimensiunea unei stări σ :

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

Dimensiunea unei configurații:

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

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 și A un algoritm determinist care rezolvă P și $d \in \{log, unif\}$.

Grupăm instanțele p ale problemei P în clase de echivalență: p și p' sunt în aceeași clasă dacă $size_d(p) = size_d(p')$.

Un număr întreg pozitiv n poate fi privit ca fiind clasa de echivalență a instanțelor de mărime n .

Complexitatea timp în cazul cel mai nefavorabil este timpul dat de execuția cu timpul cel mai mare peste configurațiile inițiale date de instanțele din aceeași clasă de echivalență:

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

Complexitatea spațiu

Fie $E = \langle A_0, \sigma_0 \rangle \Rightarrow \dots \Rightarrow \langle A_n, \sigma_n \rangle$ o execuție și $d \in \{log, unif\}$.

Spațiul consumat de această execuție este dat de maximum dintre dimensiunile configurațiilor din E :

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

Spațiul necesar algoritmului A pentru a rezolva instanța p este

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

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

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

Calcul complexității în cazul cel nefavorabil 1/3

- A este o expresie E care nu include apeluri de funcții (algoritmi): $T_{A,d}(n) = time_d(\llbracket E \rrbracket(\sigma_p))$ 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 secvențială S_1 S_2 : $T_{A,d}(n) = T_{S_1,d}(n) + T_{S_2,d}(n)$

Calcul complexității în cazul cel nefavorabil 2/3

- A este o instrucțiune iterativă (e.g., **while**, **for**): de multe ori se poate calcula doar o aproximare
 - soluția 1 (o aproximare mai fidelă):
 - * se calculează numărul maxim de iterații $nMax$
 - * se calculează complexitatea în cazul cel nefavorabil pentru fiecare iterație, fie acestea T_1, \dots, T_{nMax}
 - * se ia $T_{A,d}(n) = T_1 + \dots + T_{nMax}$
 - soluția 2 (aproximare mai grosieră):
 - * se calculează numărul maxim de iterații $nMax$
 - * se calculează complexitatea în cazul cel nefavorabil pentru iterația cu timpul (în cazul cel nefavorabil) cel mai mare, fie acesta T_{itMax}
 - * se ia $T_{A,d}(n) = nMax \times T_{itMax}$

Calcul complexității în cazul cel nefavorabil 3/3

- Atenție la liste, mulțimi, ...:

```
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 funcții (algoritmi):
 - se estimează dimensiunea argumentelor în funcție de dimensiunea instanței n
 - se utilizează complexitatea în cazul cel nefavorabil a algoritmului apelat, calculată cu dimensiunea argumentelor estimată

Calculul complexității în cazul cel nefavorabil în practică

- de obicei numai *costul uniform* este calculat
- trebuie precizată *dimensiunea unei instanțe*
- numai *o parte din operații* sunt considerate (e.g., comparări, atribuirii)
- cel mai important (și uneori) dificil este *identificarea cazului cel mai nefavorabil*
- se calculează *aproximații ale lui* $T_{A,d}(n)$ utilizând notațiile $O(f(n))$, $\Omega(f(n))$, $\Theta(f(n))$

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: $n, (a_0, \dots, a_{n-1}), z$ 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;
```

Discuția pe tablă.

Exemplul 2

input: $n, (a_0, \dots, a_{n-1})$ 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];
```

Discuția pe tablă.

Exemplul 3

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

```
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;
}
```

Discuția pe tablă.

Exemplul 4

input: $n, (a_0, \dots, a_{n-1}), z$ numere întregi;
secvența (a_0, \dots, 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
```

Discuția pe tablă.