#1 Analiza complexităţii algoritmilor

Obiective:

1. Prezentarea noţiunilor necesare pentru efectuarea analizei complexităţii unui algoritm. Complexitatea temporală. Complexitatea spaţială
2. Introducerea notaţiilor asimptotice. Ordine de complexitate
3. Exersarea analizei complexităţii pentru algoritmii:
   - Determinarea valorii maxime a unei mulțimi de numere întregi.
   - Sortarea prin inserţie a unei mulţimi de numere întregi.
   - Generarea permutărilor de ordin `n`.

1.1 Algoritmi. Analiza algoritmilor. Complexitate

Definiţie

Un **algoritm** reprezintă o procedură de calcul bine definită care primeşte o mulţime de valori drept date de intrare şi care, prin aplicarea unui set finit de reguli şi operaţii asupra acestora, produce o mulţime de valori drept date de ieşire (Cormen et al., 2000) (p. 1).

Echivalent, un algoritm poate fi privit ca un şir finit de operații sau pași care transformă datele de intrare în date de ieşire. Conform lui (Knuth, 2002) (p. 22-23), un algoritm prezintă cinci caracteristici importante:

1. **Caracter finit**: un algoritm se încheie întotdeauna după parcurgerea unui număr finit de paşi.
2. Un algoritm este **bine definit**: fiecare pas este bine specificat astfel încât să nu existe ambiguităţi în ceea ce priveşte acţiunea ce trebuie executată la un anumit moment. În acest sens, algoritmii sunt descriși folosind limbaje de programare în cadrul cărora sensul fiecărui cuvânt și fiecărei propoziţii sunt bine stabilite dinainte (e.g., există cuvinte cheie care sunt folosite doar cu un singur sens).
3. **Date de intrare**: un algoritm primeşte o mulţime de valori drept date de intrare. Se poate întâmpla ca această mulţime să conţină un singur element sau, la fel de bine, să fie nulă. Spre exemplu, un algoritm de sortare primeşte drept date de intrare numărul n de elemente de sortat precum şi valorile celor n elemente; un algoritm de testare a primalităţii unui număr primeşte ca intrare o mulţime alcătuită dintr-un singur număr; un algoritm de generare de numere aleatoare poate să nu primească nimic drept intrare.
4. **Date de ieşire**: un algoritm furnizează o mulţime de valori drept date de ieşire, mulțime ce a fost obţinută ca rezultat al aplicării unor reguli şi operaţii asupra datelor de intrare.
5. **Eficienţă**: operaţiile algoritmului sunt realizate corect şi eficient în raport cu resursele disponibile, cum ar fi timpul de calcul sau memoria sistemului.

Un algoritm trebuie să fie **general** în sensul capacităţii acestuia de a rezolva o clasă de probleme şi nu doar o anumită instanţă a problemei (reprezentată de exemplu doar de anumite date de intrare). Un algoritm poate fi **determinist** în sensul că execuţia sa este predictibilă: pentru aceleaşi date de intrare, algoritmul va furniza întotdeauna aceleaşi date de ieşire prin parcurgerea acelorași etape şi acelorași paşi la fiecare rulare. O categorie aparte este reprezentată de algoritmii **aleatori** sau **randomizați** a căror execuţie este ghidată şi influenţată de un generator de numere (pseudo-)

aleatoare, acești algoritmi având execuții diferite, timpi de execuție diferiți și chiar rezultate diferite cu fiecare rulare.

Pentru o anumită problemă putem dispune de mai multe variante de rezolvare, respectiv de mai mulţi algoritmi. Drept urmare, apare necesitatea alegerii celui mai potrivit algoritm care va fi implementat pentru a rezolva problema, această decizie luându-se în funcţie de cerinţele specifice ale aplicaţiei și de resursele disponibile (e.g., timp de calcul și memorie). În continuare sunt prezentate spre exemplificare câteva variante posibile de rezolvare pentru o problemă simplă: determinarea valorii lipsă dintr-o mulţime cu `n−1` numere distincte ce pot lua valori în domeniul `{1,2,...n}`.

Problemă exemplu

Elementul lipsă

Fie o mulţime A de numere întregi de dimensiune `n−1`, ale cărei elemente iau valori distincte în domeniul `{1,2,...n}`.
Să se determine elementul lipsă.
Datele vor fi citite dintr-un fişier ce conţine pe prima linie numărul de elemente `n` iar pe a doua linie cele `n−1` elemente separate printr-un spaţiu. Fişierul de ieşire va conține o singură linie cu elementul lipsă.

Intrare	Ieşire
6 5 1 6 3 2	4

O primă variantă de rezolvare ar putea fi verificarea prezenţei fiecărei valori posibile din domeniul `{1,2,...n}` în mulţimea `A` cu `n−1` elemente. Fiecare valoare este căutată în tabloul unidimensional folosit pentru reprezentarea mulţimii `A`. Algoritmul se încheie atunci când am identificat o valoare care nu se regăseşte în tablou. Acest prim algoritm este descris de procedura `ValoareLipsă_v1(...)`.

O altă variantă de rezolvare a problemei constă în folosirea unui tablou suplimentar de dimensiune `n` (pe care îl vom numi tablou de prezenţă) în care vom memora la poziţia `i−1` prezenţa valorii[1] `i` în mulţimea `A` folosind valorile binare `0/1` (unde `0` codifică lipsa elementului `i` iar `1` faptul că valoarea `i` este prezentă în mulţimea `A`). Tabloul de prezenţă va fi completat prin parcurgerea mulţimii `A` o singură dată, atribuind codul `1` elementului aflat la indicele `A[i]−1`. Parcurgem apoi tabloul de prezenţă pentru a identifica indicele elementului având valoarea `0`.

[1] Tabloul de prezenţă este indexat începând de la 0.

Pseudocod

```
int ValoareLipsă_v1(int[] A, int n)
      *) pentru fiecare valoare posibilă v
      pentru v ← 1, n execută
            *) căutăm apariţia lui v în tabloul A
            amGăsit ← FALS
            pentru i ← 0, n-2 execută
                  dacă A[i] == v atunci
                        amGăsit ← ADEVĂRAT
                        break


            *) valoarea v lipseşte din şir?
            dacă amGăsit == FALS atunci
                  întoarce v


sf.procedură
```

Pseudocod

```
int ValoareLipsă_v2(int[] A, int n)
      *) alocăm memorie pentru tabloul prezent
      prezent ← new int[n]
      *) iniţializăm tabloul de prezenţă cu 0
      pentru i ← 0, n-1 execută
            prezent[i] ← 0

      *) pentru fiecare valoare din mulţimea A
      *) memorăm prezenţa ei în tabloul prezent
      pentru i ← 0, n-2 execută
            prezent[A[i]-1] ← 1

      *) căutăm elementul lipsă
      pentru i ← 0, n-1 execută
            dacă prezent[i] == 0 atunci
                  întoarce i + 1


sf.procedură
```

O a treia variantă de rezolvare ar fi să calculăm suma elementelor mulţimii `A` şi, cunoscând faptul că suma primelor `n` numere naturale este `n×(n+1)/2`, să efectuăm diferenţa `n×(n+1)/2−suma` pentru a identifica elementul lipsă.

Pseudocod

```
int ValoareLipsă_v3(int[] A, int n)
    *) calculăm suma elementelor mulţimii A
    suma ← 0
    pentru i ← 0, n-2 execută
        suma ← suma + A[i]
    ■
    *) determinăm elementul lipsă
    întoarce n*(n+1)/2 - suma
sf.procedură
```

Vedem astfel cum pentru o problemă simplă dispunem de multiple variante de rezolvare. Întrebarea legitimă pe care putem să o formulăm în acest moment este: Care va fi varianta pe care o vom alege pentru implementare? Iar această întrebare ascunde de fapt problema analizei algoritmilor: **Care sunt criteriile folosite pentru a compara diverşi algoritmi?**

Analiza unui algoritm se realizează determinând **complexitatea** sa în raport cu:

1. Numărul de operaţii elementare pe care algoritmul le execută. În acest caz vorbim despre **complexitate temporală** întrucât numărul de operaţii determină timpul de execuţie.
2. Memoria necesară pentru variabilele folosite în cadrul algoritmului sau **complexitate spaţială**.

Important

Facem deosebire între timpul de execuţie al unui program (timp de rulare sau timp procesor) ce depinde de sistemul de calcul pe care rulează o implementare a algoritmului şi timpul de execuţie al unui algoritm reprezentat de numărul de operaţii elementare sau numărul de paşi de executat. Când vorbim despre timpul de execuţie al unui algoritm nu ne referim la o anumită implementare a sa şi nici la rularea unei implementări pe o anumită maşină, ci înţelegem strict numărul de paşi necesari obţinerii rezultatului.

Un pas al unui algoritm este reprezentat de execuţia unei instrucţiuni din pseudocod. Diferite instrucţiuni pot avea, bineînţeles, timpi de execuţie diferiți (spre exemplu, o înmulţire de numere reale este mai costisitoare decât o adunare de numere întregi) însă, pentru simplificarea discuţiei, vom considera ca având timp constant de execuţie (respectiv o unitate de timp) toate operaţiile cu caracter elementar: operaţii aritmetice, iniţializări şi atribuiri, indexări în tablouri, apeluri de procedură şi întoarcerea rezultatelor din proceduri. Timpul de execuţie al unui algoritm va fi dat de suma timpilor de execuţie corespunzători fiecărei instrucţiuni executate (Cormen et al., 2000) (p. 6).

De regulă, există un compromis în practică între cele două criterii: timp de execuție și memorie necesară. Putem obţine un timp de execuţie mai mic dacă folosim memorie suplimentară, spre exemplu pentru stocarea rezultatelor parţiale într-o problemă de calcul matematic şi, respectiv, vom avea un timp de execuţie mai ridicat în cazul în care cerinţele legate de memorie sunt mai exigente. **Complexitatea temporală rămâne însă criteriul folosit cel mai frecvent pentru analiza algoritmilor.**

O serie de factori au un efect direct asupra timpului de execuţie:

1. **Mărimea datelor de intrare** reprezentată de numărul de elemente furnizate spre intrare algoritmului. De exemplu, dimensiunea unui tablou de numere n, dimensiunea unei matrici $m \times n$, numărul n de noduri sau numărul m de muchii ale unui graf, etc. La nivel intuitiv, cu cât dimensiunea datelor de intrare este mai mare cu atât timpul de execuţie va creşte, însă vom vedea ulterior care poate fi forma acestei dependenţe.
2. **Conţinutul şi organizarea datelor de intrare**, în funcţie de care distingem următoarele situaţii:
   - **Cazul cel mai favorabil (CF)** pentru care datele de intrare sunt organizate de aşa natură încât algoritmul parcurge un număr minim de paşi pentru obţinerea rezultatului. Spre exemplu, cel mai favorabil caz pentru algoritmul de sortare prin inserţie a unui vector de numere întregi apare atunci când elementele vectorului sunt furnizate deja sortate crescător.
   - **Cazul cel mai defavorabil (CDF)** pentru care datele sunt organizate astfel încât algoritmul va parcurge numărul maxim de paşi pentru a obţine rezultatul. Pentru acelaşi exemplu, cazul cel mai defavorabil al algoritmului de sortare prin inserţie apare atunci când elementele vectorului sunt furnizate în ordine descrescătoare.
   - **Cazul mediu (CM)** cu o organizare aleatoare a datelor de intrare.

Timpul de execuţie al unui algoritm va fi acelaşi la fiecare rulare pentru aceleaşi date de intrare și aceeași mașină (observaţie valabilă pentru algoritmii de tip determinist, exceptând cazul algoritmilor aleatorii).

În practică, pentru analiza timpului de execuţie suntem interesaţi prioritar de:

1. **Timpul în cazul cel mai defavorabil (T-CDF)** întrucât acesta ne oferă o margine superioară privind timpul de execuție al algoritmului, indiferent de organizarea datelor de intrare pentru o anumită dimensiune `n` a acestora.
2. **Timpul mediu de execuţie (T-CM)** întrucât acesta ne oferă informaţii privind comportarea algoritmului în cazul mediu presupunând că toate datele de o anumită dimensiune `n` au aceeaşi probabilitate de apariţie.

Definiţie

Algoritmul A_1 este mai eficient temporal decât algoritmul A_2 dacă timpul de execuţie al algoritmului A_1 în cazul cel mai defavorabil este mai mic decât timpul de execuţie al algoritmului A_2:

$$\text{TE-CDF}(A_1) < \text{TE-CDF}(A_2)$$

Această definiţie ne asigură de faptul că, oricum ar fi prezentate datele la intrare, performanţa algoritmului A_1 va fi mai bună decât cea a algoritmului A_2. Cu alte cuvinte, algoritmul A_1 se încheie mai repede.

Pentru a exemplifica discuțiile anterioare, vom calcula timpul de execuţie pentru cei trei algoritmi propuşi drept soluţie pentru problema identificării valorii lipsă. Întrucât prezintă structura cea mai simplă, începem analiza cu ultima variantă:

```
  int ValoareLipsă_v3(int[] A, int n)
1       *) calculăm suma elementelor mulţimii A
2       suma ← 0
3       pentru i ← 0, n-2 execută
4       |      suma ← suma + A[i]
5       ■
6       *) determinăm elementul lipsă
7       întoarce n*(n+1)/2 - suma
8 sf.procedură
```

Linie	Timp
2	1
3	2n
4	3(n-1)
7	5
	$T_3(n)=5n+3$

Linia 3 necesită `2n` operaţii elementare întrucât execută: atribuirea `i←0` o singură dată, `n` comparaţii `i≤n−2` şi `n−1` incrementări ale variabilei `i`. Linia 4 va fi executată de `n−1` ori, realizându-se de fiecare dată trei operaţii: o indexare, o adunare şi o atribuire. Linia 7 presupune o adunare, o înmulţire, o împărţire, o scădere şi întoarcerea unei valori deci cinci operaţii. Timpul total necesar execuției procedurii `ValoareLipsă_v3` va fi:

$$T_3(n) = 1 + 2n + 3(n-1) + 5 = 5n+3$$

Observăm că pentru acest algoritm nu există cazul cel mai favorabil sau cel mai defavorabil, numărul de operaţii executate fiind aceleaşi pentru orice organizare a datelor de intrare. Continuăm discuţia cu algoritmul `ValoareLipsă_v2`.

```
    int ValoareLipsă_v2(int[] A, int n)                      Timp
 1        *) alocăm memorie pentru prezent
 2        prezent ← new int[n]
 3        *) iniţializăm tabloul cu 0
 4        pentru i ← 0, n-1 execută                         2n+2
 5              prezent[i] ← 0                              2n
 6        ■
 7        *) pentru fiecare valoare din A
 8        *) memorăm prezenţa ei
 9        pentru i ← 0, n-2 execută                         2n
10              prezent[A[i]-1] ← 1                         4(n-1)
11        ■
12        *) căutăm elementul lipsă
13        pentru i ← 0, n-1 execută                         2 .. 2n+2
14              dacă prezent[i] == 0 atunci                 1..n
15                    întoarce i + 1                        2
16              ■
17        ■
18  sf.procedură                                            T2(n) ≤ 13n+2
```

Linia 4 necesită `2n+2` operaţii elementare: atribuirea `i←0`, `n+1` comparaţii `i≤n−1` şi `n` incrementări ale variabilei `i`. Indexarea și atribuirea din linia 5 vor fi repetate de `n` ori, rezultând `2n` operații. Linia 9 necesită o atribuire `i←0`, `n` comparaţii `i≤n−2` şi `n−1` incrementări ale variabilei `i`, deci un total de `2n` operaţii. Linia 10 presupune o scădere, două indexări şi o atribuire repetate de `n−1` ori (pentru fiecare element din mulţimea `A`). Linia 14 va fi executată de maxim `n` ori şi cel puţin o dată, iar linia 15 o singură dată. Rezultă prin urmare timpul necesar execuţiei procedurii `ValoareLipsă_v2` este încadrat de:

$$10n+3 \le T_2(n) \le 13n+2$$

Cum execuţia algoritmului depinde de modul în care sunt prezentate datele de intrare (ceea ce determină un număr diferit de operaţii în liniile 13 şi 14), trebuie discutate cazurile cel mai favorabil şi cel mai defavorabil. Cazul cel mai favorabil apare atunci când algoritmul execută numărul minim de operaţii (`10n+3`) şi anume când valoarea `1` este cea care lipseşte din mulțimea A. Cel mai defavorabil caz apare atunci când lipseşte valoarea `n`, algoritmul executând `13n+2` operaţii. În cazul mediu, ne aşteptăm ca linia 13 să se execute pentru jumătate din valori, deci estimăm timpul de execuţie în cazul mediu ca fiind:

$$T_2\text{-}CM(n)=11.5n$$

Continuăm analiza pentru algoritmul `ValoareLipsă_v1`.

```
   int ValoareLipsă_v1(int[] A,int n)
1    *) pentru fiecare valoare v
2    pentru v ← 1, n execută
3       *) căutăm v în A
4       amGăsit ← FALS
5       pentru i ← 0, n-2 execută
6            dacă A[i] == v atunci
7               amGăsit ← ADEVĂRAT
8               break
9            ■
10      ■
11      *) valoarea v lipseşte?
12      dacă amGăsit == FALS atunci
13           întoarce v
14      ■
15   ■
16 sf.procedură
```

Linia	Timp CF	Timp CDF	Timp CM
2	2	2n+2	n
4	1	n	0.5n
5	2n	n×2n	$0.5n^2$
6	2n-2	$.5n^2+2n$	$0.5n^2$
7		n-1	0.5n
8		n-1	0.5n
12	1	n	0.5n
13	1	1	1
Total	4n+3	$2.5n^2+8n$	n^2+3n+1

Algoritmul `ValoareLipsă_v1` se încheie atunci când găseşte valoarea v care nu se află în mulţimea `A`. Cazul cel mai favorabil apare atunci când elementul care lipseşte este `1` iar cazul cel mai defavorabil când lipseşte valoarea `n`.

Cazul cel mai favorabil

Linia 2 presupune doar două operaţii: atribuirea `v←1` şi comparaţia `v≤n`. Cum elementul `1` nu se află în mulţimea `A`, linia 5 va necesita `2n` operaţii iar linia 6 va fi executată de n−1 ori, de fiecare dată realizându-se câte o indexare și o comparație. Liniile 7 şi 8 nu se vor executa iar linile 12-13 vor fi executate o singură dată. Timpul necesar algoritmului în cazul cel mai favorabil va fi astfel:

$$T_1\text{-CF}(n)=4n+3$$

Cazul cel mai defavorabil

Linia 2 presupune 2n+2 operaţii: atribuirea v←1, n+1 comparaţii v≤n şi n incrementări ale variabilei v. Linia 4 va fi executată de n ori. Căutarea valorii v în mulţimea A are loc în liniile 5-9. Cum ne aflăm în cazul cel mai defavorabil în care valoarea n este cea care lipsește, restul valorilor 1,2,...n−1 se vor regăsi în A (de fiecare dată când găsim un element părăsim ciclul pentru din liniile 5-9 prin instrucţiunea break din linia 8). Rezultă că vom executa un număr de n(n+1)/2−1 comparaţii[2] în linia 6 şi de n^2+2n operaţii în linia 5: de n ori va fi executată atribuirea i←1, de n(n+1)/2 ori comparaţia i≤n−2 şi de n(n+1)/2 ori incrementarea variabilei i. Liniile 7 şi 8 vor fi executate de n−1 ori, câte o execuţie pentru fiecare valoare v care se regăseşte în mulţimea A. Linia 12 este executată de n ori iar linia 13 o singură dată. Avem timpul în cazul cel mai defavorabil:

$$T_1\text{-CDF}(n)= 2.5n^2+8n$$

Cazul mediu

În cazul mediu vom căuta în medie n/2 valori în mulţimea A. Deci, linia 2 va presupune o atribuire, n/2 comparaţii şi n/2 incrementări, linia 4 va fi executată de n/2 ori, vom avea un număr aproximativ de n/2×n/2 comparaţii în linia 6 şi n/2+n/2×n/2×2 operaţii necesare ciclului pentru în linia 5. Liniile 7-8 se execută de n/2−1 ori iar linia 12 de n/2 ori. Avem timpul în cazul mediu:

$$T_1\text{-CM}(n)= n^2+3n+1$$

Tabelul 1.1 rezumă rezultatele pe care le-am obţinut efectuând analiza temporală a fiecăruia dintre cei trei algoritmi propuşi pentru rezolvarea problemei elementului lipsă. Doi algoritmi au necesitat analiza timpului de execuţie pe cazuri în funcţie de structura datelor de intrare, în timp ce ValoareLipsă_v3 efectuează acelaşi număr de operaţii în toate cazurile. Putem trage concluzia că algoritmul ValoareLipsă_v3 execută cele mai puţine operaţii în CDF şi în CM însă este întrecut de algoritmul ValoareLipsă_v1 în cazul cel mai favorabil. Acest lucru are loc întrucât algoritmul

[2] Elementele mulţimii A sunt distincte şi căutăm fiecare valoare din 1, 2, ... n. Primele n-1 valori se vor regăsi în A pe locaţii distincte deci fiecare locaţie din A va genera părăsirea ciclului pentru. Drept urmare, vor fi executate 1 + 2 + ... + n-1 comparaţii pentru aceste elemente. Valoarea n nu se află în A deci linia 6 va necesita n-1 comparaţii. Rezultă un total de 1 + 2 + ... + n-1 + n-1 = n*(n+1)/2 – 1 comparaţii.

v3 realizează acelaşi lucru indiferent de modul în care se prezintă datele de intrare (de fiecare dată va calcula suma celor `n−1` valori ale mulţimii `A`) pe când ceilalţi algoritmi îşi termină execuţia de îndată ce au identificat valoarea lipsă. Diferenţa este doar de `n` operaţii suplimentare. Dintre toţi algoritmii, `ValoareLipsă_v1` prezintă cea mai slabă comportare în cazurile CDF şi CM.

Tabelul 1.1 Timpul de execuţie al algoritmilor pentru problema elementului lipsă

Algoritm	CF	CDF	CM
`ValoareLipsă-v1`	4n+3	$2.5n^2+8n$	n^2+3n+1
`ValoareLipsă-v2`	10n+3	13n+2	11.5n
`ValoareLipsă-v3`	5n+3	5n+3	5n+3

1.2 Notaţii asimptotice

Pentru estimarea timpului de execuţie sunt folosite notaţii asimptotice, dintre care în continuare vom prezenta notaţiile O, Ω şi Θ.

Definiţie

Notaţia Θ

Dată fiind o funcţie $g(n): N \to N$, fie mulţimea de funcţii:

$$\Theta(g(n)) = \{f(n) / \exists\, c_1, c_2 > 0, n_0 > 0 \; a.i.\; 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \; \forall n \ge n_0\}$$

Scriem $f(n) = \Theta(g(n))$ cu semnificaţia g(n) reprezintă o margine asimptotic tare pentru f(n).

Cu alte cuvinte, funcţia g(n) mărgineşte f(n) asimptotic strâns, atât inferior cât şi superior.

Exemplu

Funcţia $f(n) = 5n^2 - 7n + 20$ este $\Theta(n^2)$ întrucât există $g(n) = n^2$ şi $c_1 = 4$, $c_2 = 6$ astfel încât $c_1 g(n) \le f(n) \le c_2 g(n)$ pentru orice $n \ge 3$.

Funcţia $f(n) = n^4 + 10$ este $\Theta(n^4)$ întrucât există $g(n) = n^4$, $c_1 = 1, c_2 = 2$ astfel încât $c_1 g(n) \le f(n) \le c_2 g(n)$ pentru orice $n \ge 2$.

Funcţia $f(n) = n\log(n) + 3n + 10$ este $\Theta(n\log(n))$ întrucât există funcţia $g(n) = n\log(n)$ şi constantele $c_1 = 1, c_2 = 2$ astfel încât $c_1 g(n) \le f(n) \le c_2 g(n)$ pentru orice $n \ge 20$.

Definiţie

Notaţia O

Dată fiind o funcţie $g(n): N \to N$, fie mulţimea de funcţii:

$$O(g(n)) = \{ f(n) / \exists\, c > 0\ si\ n_0 > 0\ a.i.\ 0 \le f(n) \le cg(n)\ \forall n \ge n_0 \}$$

Scriem $f(n) = O(g(n))$ cu semnificaţia g(n) reprezintă o margine asimptotic superioară pentru f(n).

Definiţie

Notaţia Ω

Dată fiind o funcţie $g(n): N \to N$, fie mulţimea de funcţii:

$$\Omega(g(n)) = \{ f(n) / \exists\, c > 0\ si\ n_0 > 0\ a.i.\ 0 \le cg(n) \le f(n)\ \forall n \ge n_0 \}$$

Scriem $f(n) = \Omega(g(n))$ cu semnificaţia g(n) reprezintă o margine asimptotic inferioară pentru f(n).

Notaţia O poate fi folosită pentru a delimita timpul de execuţie al unui algoritm în cazul cel mai defavorabil şi, implicit, delimitează timpul de execuţie pentru orice date de intrare. Notaţia Ω furnizează o limită inferioară pentru timpul de execuţie pentru orice date de intrare. Cu alte cuvinte, suntem siguri că nu putem obţine un timp mai bun pentru algoritmul respectiv. Notaţia Θ delimitează timpul de execuţie atât inferior cât şi superior, însă nu pentru orice date de intrare. Drept urmare, trebuie calculată în funcţie de cazul cel mai defavorabil, cazul cel mai favorabil, etc.

Tabelurile 1.2 şi 1.3 prezintă numărul de operaţii respectiv timpul de execuţie estimat necesar rulării unor algoritmi de diverse complexităţi pentru un sistem de calcul ce poate executa 10^9 operaţii pe secundă.

Tabelul 1.2 Numărul de operaţii necesare unor algoritmi de diverse complexităţi pentru diferite valori ale dimensiunii datelor de intrare `n`.

Complexitate	**n**	**10^2**	**10^3**	**10^4**	**10^5**
Logaritmică	**O(log(n))**	7	10	13	17
Liniară	**O(n)**	10^2	10^3	10^4	10^5
Liniar logaritmică	**O(nlog(n))**	7×10^2	10^4	$1,3\times10^5$	$1,7\times10^6$
Pătratică	**O(n^2)**	10^4	10^6	10^8	10^{10}
Polinomială de grad > 2	**O(n^p) (ex: p=4)**	10^8	10^{12}	10^{16}	10^{20}
Exponenţială	**O(a^n) (ex: a=10)**	10^{100}	$10^{1,000}$	$10^{10,000}$	$10^{100,000}$

Tabelul 1.3 Timpul de execuţie pentru diferite valori ale dimensiunii datelor de intrare n pentru un sistem de calcul care poate efectua 10^9 operaţii pe secundă.

Complexitate / **n**		**10^2**	**10^3**	**10^4**	**10^5**
Logaritmică	**O(log(n))**	0	0	0	0
Liniară	**O(n)**	0	0	0.01 ms	0.1 ms
Liniar logaritmică	**O(nlog(n))**	0	0.01 ms	0.13 ms	1,7 ms
Pătratică	**O(n^2)**	0.01 ms	1 ms	0.1 s	10 s
Polinomială de grad > 2	**O(n^p)** (ex. p=4)	0.1 s	16,6 min	115 zile	3170 ani
Exponenţială	**O(a^n)** (ex. a=10)	10^{92}ani	...	...	...

Putem realiza o ordonare a complexităţilor din tabelurile de mai sus în funcţie de numărul de operaţii necesare terminării calculului, astfel:

$$O(1) < O(\log(n)) < O(n) < O(n \times \log(n)) < O(n^2) < O(n^p) < O(a^n)$$

`O(1)` reprezintă complexitatea constantă. Un algoritm cu complexitatea `O(1)` va executa un număr constant de operaţii indiferent de dimensiunea datelor de intrare. În practică vom fi interesaţi de găsirea de algoritmi de complexitate logaritmică, liniară, liniar-logaritmică şi vom accepta complexităţi pătratice sau polinomiale în funcție de dimensiunea prognozată pentru datele de intrare.

1.3 Analiza timpului de execuţie: determinarea elementului maxim dintr-o mulțime de numere întregi

Problemă exemplu

Determinarea valorii maxime

Fie o mulţime de numere întregi de dimensiune `n`. Să se găsească elementul de valoare maximă.

Datele vor fi citite dintr-un fişier ce conţine pe prima linie numărul de elemente `n` iar pe a doua linie elementele mulţimii separate printr-un spaţiu. Fişierul de ieşire va avea o singură linie reprezentând elementul de valoare maximă.

Intrare	**Ieşire**
7 3 1 8 2 7 3 5	8

Pseudocod

```
  int DeterminaMaxim(int[] a, int n)
1     max ← a[0]
2     pentru i ← 1, n-1 execută
3         dacă max < a[i] atunci
4             max ← a[i]
5         ■
6     ■
7     întoarce max
8 sf.procedură
```

Pentru analiza complexităţii temporale a algoritmului vom număra operaţiile elementare efectuate pentru execuţia fiecărei linii, iar timpul total va fi obţinut prin însumarea timpilor de execuţie pe linie, cu discuţii în funcţie de cazul cel mai favorabil, cel mai defavorabil sau cazul mediu.

Linia 1 realizează o indexare în șirul `a` și o operaţie de atribuire. Liniile 2-6 reprezintă un ciclu care va fi executat de `n−1` ori (variabila `i` ia valori succesiv de la `1` la `n−1`). Iniţial, variabila `i` este iniţializată cu valoarea `1`, ceea ce reprezintă o operaţie elementară. La fiecare iteraţie, următorii paşi vor fi executaţi:

- este efectuat testul `i≤n−1` în cadrul instrucţiunii `pentru`
- este efectuat testul `max<a[i]`
- dacă expresia `max<a[i]` este evaluată la adevărat atunci va fi realizată atribuirea `max←a[i]`
- variabila `i` este incrementată

La terminarea ciclului, variabila `i` va avea valoarea `n` şi vom executa o nouă comparaţie `i≤n−1` care va termina ciclul repetitiv. Linia 7 întoarce valoarea maximă din şir pe care o considerăm o operaţie elementară.

Cazul cel mai defavorabil

Cazul cel mai defavorabil apare atunci când şirul de numere este sortat crescător şi atribuirea `max←a[i]` va fi realizată de `n−1` ori. Avem astfel timpul de execuţie[3]:

$$T\text{-}CDF(n)=2+(1+n+n-1+2(n-1)+2(n-1))+1=6n-1$$

Pentru a exprima superior timpul de execuţie folosind notaţia O trebuie să găsim o funcţie `g(n)`, o constantă `c` şi o valoare n_0 astfel încât pentru orice $n \geq n_0$, T−

[3] (1+n+n-1+2(n-1)+2(n-1)) înseamnă: o atribuire i ← 0, n comparaţii i ≤ n-1, n-1 comparaţii max<a[i], n-1 atribuiri max←a[i] şi n-1 incrementări ale variabilei i pentru care am numărat 2 operaţii: indexarea vectorului a și atribuirea efectivă.

$CDF(n)$ să fie marginit superior de funcţia $g(n)$. Putem observa că alegând $g(n)=n$, $c=6$ şi $n_0=1$ vom avea $T\text{-}CDF(n)\leq c\times g(n)$. Drept urmare, putem scrie:

$$T\text{-}CDF(n)=O(g(n))=O(n)$$

deci timpul de execuţie al algoritmului de determinare a elementului maxim în cazul cel mai defavorabil este liniar în funcţie de dimensiunea mulțimii. De asemenea, alegând $g(n)=n$, $c=3$ şi $n_0=1$ avem $c\times g(n)\leq T\text{-}CDF(n)$ pentru $n\geq n_0$. Putem scrie deci $T\text{-}CDF(n)=\Omega(n)$ şi, combinând cele două rezultate, $T\text{-}CDF(n)=\Theta(n)$.

Cazul cel mai favorabil apare atunci când şirul este sortat descrescător iar atribuirea $max\leftarrow a[i]$ nu are loc niciodată. În acest caz vom avea timpul de execuţie:

$$T\text{-}CF(n)=1+(1+n+n-1+2(n-1))+1=4n$$

şi, urmând o analiză similară celei de mai sus, obţinem şi în cazul cel mai favorabil un timp mărginit superior $O(n)$, inferior $\Omega(n)$ precum şi o margine strânsă $\Theta(n)$.

În **cazul mediu** ne vom aştepta ca atribuirea $max\leftarrow a[i]$ să se realizeze de aproximativ $n/2$ ori, drept urmare timpul de execuţie devine:

$$T\text{-}CM(n)=1+(1+n+n-1+2(n-1)+2n/2)+1=5n$$

având de asemenea un timp mărginit superior $O(n)$ respectiv inferior $\Omega(n)$.

1.4 Analiza timpului de execuţie: sortarea prin inserţie

Problemă exemplu

Sortarea prin inserţie

Fie o mulțime de numere întregi de dimensiune n. Să se sorteze elementele şirului în ordine crescătoare folosind algoritmul sortării prin inserţie.

Datele vor fi citite dintr-un fişier care conţine pe prima linie numărul de elemente n iar pe a doua elementele mulțimii separate printr-un spaţiu. Fişierul de ieşire va conţine o singură linie cu elementele mulțimii ordonate crescător.

Intrare	Ieşire
7 3 1 8 2 7 3 5	1 2 3 3 5 7 8

Algoritmul sortării prin inserţie funcţionează conform următorului principiu: presupunând elementele subşirului a[0]..a[j-1] deja sortate crescător, următorul element a[j] va fi inserat la poziţia corectă în a[0]..a[j-1] astfel încât, în final, elementele subşirului a[0]..a[j] să fie de asemenea sortate crescător. Inserarea elementului a[j] se realizează căutând locul său în subşirul sortat şi deplasând la dreapta cu o poziţie toate elementele mai mari decât el. Plecând de la cazul iniţial (subşirul alcătuit dintr-un singur element a[0] care este implicit sortat) şi parcurgând cu variabila j fiecare din elementele rămase pe poziţiile 1..n-1 vom obţine în final în şirul a elementele sortate crescător.

Pseudocod

```
   procedura SORTEAZĂ-PRIN-INSERŢIE(int[] a, int n)
1      *) fiecare element a[j] va fi inserat la
2      *) locul său în subşirul sortat a[0]..a[j-1]
3     pentru j ← 1, n-1 execută
4         cheie ← a[j]
5         *) inserează a[j] în a[0]..a[j-1]
6         i ← j - 1
7         cât timp i >= 0 şi a[i] > cheie execută
8             a[i+1] ← a[i]
9             i ← i - 1
10        ■
11        a[i+1] ← cheie
12    ■
13  sf.procedură
```

Algoritmul foloseşte două cicluri repetitive: odată variabila j parcurge elementele şirului între poziţiile 1 şi n-1, iar pentru fiecare poziţie j elementul a[j] va fi adus la poziţia sa corectă în subşirul deja sortat a[0]..a[j-1]. Linia 3 conţine drept operaţii: iniţializarea variabilei j cu valoarea 1, testul j≤n-1 executat de n ori și incrementarea variabilei j executată de n-1 ori. Vom avea pentru linia 3 un număr total de 1+n+n-1=2n operaţii. Linia 4 va fi executată de n-1 ori pentru fiecare valoare a variabilei j rezultând un număr de 2(n-1) operații elementare (indexări și atribuiri). Similar, linia 6 se execută pentru fiecare j, rezultând un număr de 2(n-1) operații elementare (scăderi și atribuiri). Liniile 7-10 descriu ciclul care va insera fiecare element a[j] la locul său în subşirul sortat a[0]..a[j-1]. Operaţiile elementare sunt cele două teste i>=0, a[i]>cheie şi cele două atribuiri care vor fi repetate de un număr de R_j ori (acest număr depinde de valoarea elementului a[j]). Deci, timpul total necesar execuţiei ciclului 7-10 pentru o valoare a variabilei j va fi $9 \times R_j$ (o comparație i>=0; o indexare și comparație a[i]>cheie; o adunare, două indexări și o atribuire pentru a[i+1]←a[i]; o scădere și o atribuire pentru i←i-1). Cum însă variabila j parcurge vectorul a între 1 şi n-1, timpul total pentru liniile 7-10 va fi $\sum_{j=1}^{n-1} 9R_j$.

Linia 11 este executată pentru fiecare a[j] deci de un număr de n-1 ori, rezultând 3(n-1) operații elementare. Însumând timpii calculaţi mai sus obţinem timpul total de execuţie necesar algoritmului de sortare prin inserţie:

$$T(n) = 2n + 4(n-1) + \sum_{j=1}^{n-1} 9R_j + 3(n-1) = 9n - 7 + \sum_{j=1}^{n-1} 9R_j$$

Cazul cel mai defavorabil apare atunci când şirul de numere este sortat descrescător întrucât ciclul din liniile 7-10 va executa un număr maxim de operaţii: fiecare element a[j] va fi adus până în poziţia 0, restul elementelor fiind deplasate cu o poziţie la dreapta. Drept urmare, valorea factorului de repetiţie pentru elementul a[j] va fi R_j=j. Timpul de execuţie devine:

$$T - CDF(n) = 9n - 7 + \sum_{j=1}^{n-1} 9j = 9n - 7 + 9 \cdot \frac{n(n-1)}{2} = 4.5n^2 + 4.5n - 7$$

Vom avea prin urmare: T-CDF(n)=O(n^2), T-CDF(n)=Ω(n^2), respectiv T-CDF(n)=Θ(n^2).

Cazul cel mai favorabil apare atunci când şirul de numere este sortat deja crescător ceea ce face ca liniile 8-9 din cadrul ciclului 7-10 să nu se execute nici o dată și să obținem de fiecare dată fals la testarea condiţiei a[i]>cheie întrucât elementul a[j] se află deja pe locul său. Timpul de execuţie devine T-CF(n)=9n-7+3(n-1)=12n-8. Vom avea deci pentru cazul cel mai favorabil T-CF(n)=O(n),T-CF(n)=Ω(n),T-CF(n)=Θ(n).

În **cazul mediu** ciclul 7-10 va fi executat de aproximativ j/2 ori ceea ce ne conduce la R_j=j/2 şi un timp de calcul:

$$T - CM(n) = 9n - 7 + \sum_{j=1}^{n-1} 9\frac{j}{2} = 9n - 7 + 4.5 \cdot \frac{n(n-1)}{2} = 2.25n^2 + 6.75n - 7$$

Vom avea deci şi pentru cazul mediu un timp de execuţie pătratic: T-CM(n)=O(n^2),T-CM(n)=Ω(n^2), respectiv T-CM(n)=Θ(n^2).

O implementare C# este prezentată în continuare împreună cu rezultate ale timpului de execuție pentru diverse valori ale dimensiunii datelor de intrare `n` (Figura 1.1).

```
class SortareaPrinInsertie
{
    /// <summary>
    /// Sorteaza vectorul a folosind metoda
    /// sortarii prin insertie.
    /// Complexitate O(n^2).
    /// </summary>
    /// <param name="a"></param>
    public static void Sorteaza(int[] a)
    {
        for (int j = 1; j < a.Length; j++)
        {
            // cheia va fi inserata la locul potrivit
            // in subsirul a[0]..a[j-1] deja sortat
            int cheie = a[j];
            int i = j - 1;
            while (i >= 0 && a[i] > cheie)
            {
                a[i + 1] = a[i];
                i--;
            }
            a[i + 1] = cheie;
        }
    }
}
```

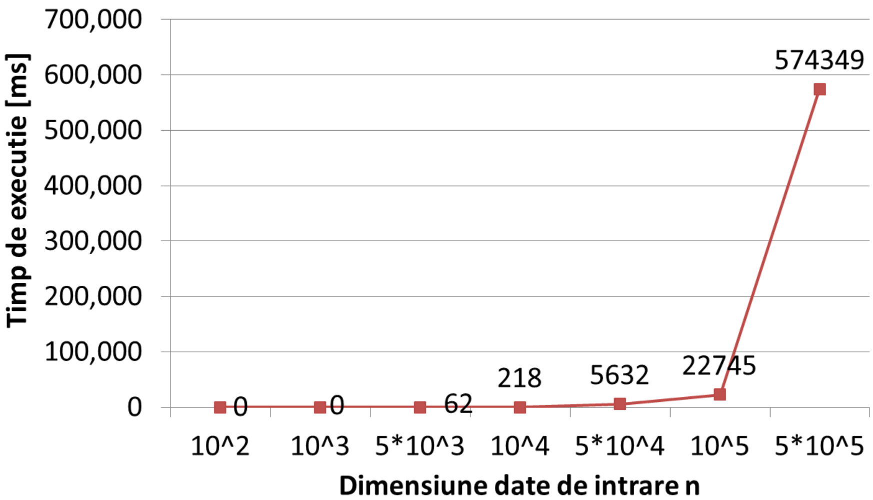

Figura 1.1. Timpul de execuție în milisecunde obținut pentru rularea algoritmului de sortare prin inserție pentru diverse dimensiuni ale datelor de intrare. Notă: timpi măsurați pe un PC Intel Core2 Quad CPU 2.40GHz.

1.5 Analiza timpului de execuţie: generarea permutărilor de ordin n

Problemă exemplu

Generarea permutărilor

Să se genereze toate permutările unei mulţimi de n elemente.

Valoarea n va fi citită dintr-un fişier de intrare. Fişierul de ieşire va conţine permutările mulţimii {1,2,...n}, fiecare pe câte o linie.

Intrare	Ieşire
4	1,2,3,4 1,2,4,3 1,3,2,4 1,3,4,2 1,4,2,3 1,4,3,2 2,1,3,4 2,1,4,3 2,3,1,4 2,3,4,1 2,4,1,3 2,4,3,1 ...

Pentru genererea permutărilor vom folosi o stivă cu ajutorul căreia, conform metodei backtracking, vom explora spaţiul soluţiilor problemei. O soluţie este validă dacă numărul de elemente aflate în stivă este n respectiv toate elementele sunt distincte. Procedura GENERARE-PERMUTĂRI(...) descrie modul de obținere a soluțiilor. Variabilele as respectiv ev fac referire la nivelul curent al stivei codificând dacă avem un succesor pentru nivelul curent (as), respectiv dacă acel succesor este valid (ev). Pentru fiecare nivel k=0,n−1 al stivei vom încerca pe rând fiecare din cele n numere din mulţimea {1,2,...n} testând condiţia de validitate. Când stiva are dimensiunea n am ajuns la o nouă soluţie (o nouă permutare) pe care o vom afișa.

Întrucât fiecare nivel al stivei ia pe rând toate cele n valori şi trebuie să avem o stivă completă de dimensiune n pentru o soluţie, rezultă că vom avea un timp de execuţie de ordinul $O(n^n)$. Implementări în pseudocod și în limbajul C# sunt prezentate în continuare.

Pseudocod

```
procedura GENERARE-PERMUTĂRI(int n)
    *) k reprezintă nivelul curent al stivei
    k ← 0
    stiva[k] ← 0
    cât timp k >= 0 execută
        repetă
            *) generează un nou element
            as ← ADEVĂRAT
            dacă stiva[k] < n atunci
                stiva[k] ← stiva[k] + 1
                *) toate elementele distincte?
                ev ← ADEVĂRAT
                pentru i ← 0, k-1 execută
                    dacă stiva[i]==stiva[k] atunci
                        ev ← FALS


             altfel
                as ← FALS

        cât timp as şi !ev
        dacă as atunci
             dacă k == n-1 atunci
                 *) scrie valorile din stiva
                 pentru i ← 0, n-1 execută
                      scrie stiva[i]

              altfel
                 k ← k + 1
                 stiva[k] ← 0

         altfel
             k ← k - 1

sf.procedură
```

```
class GenerarePermutari
{
    /// <summary>
    /// Genereaza permutarile de ordin n.
    </summary>
    public static void GenereazaPermutari(int n)
    {
        int[] stiva = new int[n];
        int k = 0;  // nivelul stivei
        stiva[k] = 0;
        while (k >= 0)
        {
            bool amSuccesor = false, esteValid = false;
            do
            {
                if (stiva[k] < n)
                {
                    amSuccesor = true;
                    stiva[k]++;
                    esteValid = true;
                    for(int i = 0; i < k; i++)
                        if (stiva[i] == stiva[k])
                        {
                            esteValid = false;
                            break;
                        }
                }
                else amSuccesor = false;
            } while (amSuccesor && !esteValid);
            if (amSuccesor)
            {
                if (k == n - 1)
                {
                    for (int i = 0; i < n; i++)
                        Console.Write("{0}{1}",
                            stiva[i], i == n -  1 ? "" : ",");
                    Console.WriteLine();
                }
                else { k++; stiva[k] = 0; }
            }
            else k--;
        }
    }
```

1.6 Probleme propuse

Problema #1

Maxim, inserție, permutări

Implementaţi algoritmii prezentaţi anterior:

- Determinarea valorii maxime a unei mulţimi cu n elemente.
- Sortarea prin inserţie a unei mulțimi de dimensiune n.
- Generarea permutărilor de ordin n.

Rulaţi fiecare algoritm pentru diferite valori ale lui n şi măsuraţi timpul de execuţie. Folosiți următoarele valori pentru n:

- Determinarea maximului: n=100; 1,000; 10,000; 100,000; 1,000,000; 100,000,000. Mulțimea de numere va fi generată aleator.
- Sortarea prin inserţie: n=10; 100; 1,000; 2,000; 5,000; 10,000. Mulțimea de numere va fi generată aleator.
- Generarea permutărilor: n=3; 5; 7; 10; 15.

Reprezentaţi grafic timpul de execuţie în funcţie de dimensiunea datelor pentru fiecare algoritm. Comparaţi forma graficului obţinut cu complexitatea așteptată.

Care este complexitatea temporală a următorilor algoritmi?

Problema #2

Înmulţirea matricilor

Cunoscând elementele matricii A de dimensiune $n \times m$ şi ale matricii B de dimensiune $m \times p$, să se calculeze produsul $A \times B$.

Cele două matrici vor fi citite dintr-un fişier cu următorul format: prima linie conţine dimensiunile matricei A (n și m) separate printr-un spaţiu iar următoarele n linii conţin elementele matricei, $A_{i,j}$. Matricea B urmează în aceeaşi reprezentare. Fişierul de ieşire va conţine elementele produsului $A \times B$.

Intrare	Ieşire
2 2 1 2 3 4 2 3 1 0 1 2 0 3	5 0 7 11 0 15

Pseudocod

```
procedura ÎnmulţireMatrici(
    int[,] A, int n, int m,
    int[,] B, int m, int p,
    out int[,] C)
    pentru i ← 0, n-1 execută
        pentru j ← 0, p-1 execută
            C[i,j] ← 0
            pentru k ← 0, m-1 execută
                C[i,j] ← C[i,j]+A[i,k]*B[k,j]



sf.procedură
```

Problema #3

Ridicarea la putere

Să se determine valoarea a^n unde a reprezintă un număr real iar n un număr întreg.

Fişierul de intrare conţine pe fiecare linie o pereche de valori a şi n separate printr-un spaţiu. Rezultatul fiecărei ridicări la putere se va regăsi în fişierul de ieşire pe câte o linie.

Intrare	Ieşire
10.2 3 10 4 8 20 2 4 1 1 27.45 0	1061.208 10000 1152921504606846976 16 1 1

Pseudocod

```
float Putere-1(float a, int n)
    p ← 1
    pentru i ← 1, n execută
        p ← p * a

    întoarce p
sf.procedură
```

Pseudocod

```
float Putere-2(float a, int n)
     dacă n == 0 atunci întoarce 1
     dacă n == 1 atunci întoarce a
     dacă n % 2 == 0 atunci
          întoarce Putere-2(a*a, n/2)
      altfel
          întoarce a * Putere-2(a*a, (n-1)/2)

sf.procedură
```

Pseudocod

```
float Putere-3(float a, int n)
     p ← 1
     i ← n
     cât timp i > 0 execută
          dacă i % 2 == 1 atunci
               p ← p * a

          a ← a * a
          i ← i / 2

     întoarce p
sf.procedură
```

Problema #4

Şir de puteri

Fie un şir de numere reale de dimensiune n. Să se calculeze suma $\sum_{i=0}^{n-1} s_i^{i+1}$. Fişierul de intrare conţine pe prima linie numărul n de elemente din şirul s urmate de elementele şirului pe câte o linie. Rezultatul sumei calculate se va regăsi în fişierul de ieşire.

Intrare	Ieşire
4 1 0.5 0.25 0.125	1.27

Pseudocod

```
float Sir-Putere(float[] sir, int n)
    r ← 0
    pentru i ← 0, n-1 execută
        r ← r + Putere-3(sir[i], i+1)

    întoarce r
sf.procedură
```

Problema #5

Testarea primalităţii

Să se determine dacă un număr natural `n` este prim.

Fişierul de intrare va conţine un număr de linii, fiecare linie reprezentând un număr natural `n` care trebuie testat. Rezultatul codificat prin textul `PRIM` sau `COMPUS` va fi afişat pe câte o linie în fişierul de ieşire.

Intrare	Ieşire
15 2 4999 15551889	COMPUS PRIM PRIM COMPUS

Pseudocod

```
string PRIM(int n)
    dacă n == 2 atunci
        întoarce "PRIM"

    dacă n % 2 == 0 atunci
        întoarce "COMPUS"

    r ← sqrt(n)
    pentru i ← 3, r, i+=2 execută
        dacă n % i == 0 atunci
            întoarce "COMPUS"


    întoarce "PRIM"
sf.procedură
```

Problema #6

Sortarea prin metoda bulelor

Fie o mulțime de numere întregi de dimensiune `n`. Să se sorteze elementele şirului în ordine crescătoare folosind metoda bulelor.

Datele vor fi citite dintr-un fişier de intrare care conţine pe prima linie numărul de elemente `n` iar pe a doua linie elementele mulțimii separate prin spaţiu. Fişierul de ieşire va conţine o singură linie cu elementele şirului ordonate crescător.

Intrare	Ieşire
7 3 1 8 2 7 3 5	1 2 3 3 5 7 8

Pseudocod

```
procedura SORTEAZĂ-BULE(int[] a, int n)
    repetă
        sortat ← ADEVĂRAT
        pentru i ← 0, n-2 execută
            dacă a[i] > a[i+1] atunci
                *) interschimbă a[i] cu a[i+1]
                a[i] ←→ a[i+1]
                sortat ← FALS
            ■
        ■
    cât timp !sortat
sf.procedură
```

Problema #7

Sortarea prin selecţia minimului

Fie o mulțime de numere întregi de dimensiune `n`. Să se sorteze elementele mulțimii folosind metoda selecţiei valorii minime.

Datele vor fi citite dintr-un fişier de intrare care conţine pe prima linie numărul de elemente `n` iar pe a doua linie elementele mulțimii separate printr-un spaţiu. Fişierul de ieşire va conţine o singură linie cu elementele mulțimii ordonate crescător.

Intrare	Ieşire
7 3 1 8 2 7 3 5	1 2 3 3 5 7 8

Pseudocod

```
procedura CAUTĂ-MIN(int[] a, int left,int right)
    indexMin ← left
    pentru i ← left+1, right execută
        dacă a[i] < a[indexMin] atunci
            indexMin ← i
        ▀
    ▀
    întoarce indexMin
sf.procedură

procedura SORTEAZĂ-SELECŢIE-MIN(int[] a, int n)
    pentru i ← 0, n-2 execută
        indexMin ← CAUTĂ-MIN(a, i, n-1)
        dacă i != indexMin atunci
            *) interschimbă a[i] cu a[indexMin]
            a[i] ←→ a[indexMin]
        ▀
    ▀
sf.procedură
```

Problema #8

Produs cartezian

Fie n mulţimi A_i de numere întregi de dimensiuni m_i, $i=0,n-1$. Să se determine elementele produsului cartezian $A_0 \times A_1 \times ... \times A_{n-1}$.

Datele vor fi citite dintr-un fişier de intrare care conţine pe prima linie numărul de mulţimi n. Urmează fiecare mulţime A_i descrisă prin numărul de elemente m_i pe prima linie respectiv elementele mulţimii pe a doua linie, separate printr-un spaţiu. Fişierul de ieşire va conţine elementele produsului cartezian.

Intrare	Ieşire
3	2 1 5
2	2 3 5
2 4	2 8 5
3	4 1 5
1 3 8	4 3 5
1	4 8 5
5	

Explicaţie: 3 mulțimi, A_0={2,4}, A_1={1,3,8}, A_2={5}.

Pseudocod

```
procedura PRODUS-CARTEZIAN(int n, int[] m,
                           int[,] A)
    *) k reprezintă nivelul curent al stivei
    k ← 0
    stiva[k] ← -1
    cât timp k >= 0 execută
        dacă stiva[k] < m[k]-1 atunci
            stiva[k] ← stiva[k] + 1
            dacă k == n-1 atunci
                *) scrie valorile din stiva
             altfel
                k ← k + 1
                stiva[k] ← -1
            ■
         altfel
            k ← k - 1
        ■
    ■
sf.procedură
```

Care este expresia complexităţii temporale pentru procedura PRODUS-CARTEZIAN atunci când mulţimile A_i au acelaşi număr de elemente, m_i=m?