SDA – Seminar 2 – Complexități

- **Cuprins**:
  - Introducere
  - Notații asimptotice de complexitate
  - Exerciții

Introducere

Cum definim eficiența unui algoritm? Eficiența unui algoritm este determinată de cantitatea de resurse pe care le consumă, în termeni de *timp* și *memorie*.

Ca modalități de măsurare a complexității timp, avem:

- Analiza **empirică** (sau **experimentală**), care constă în măsurarea timpului efectiv de execuție (aceasta reprezintă un avantaj al metodei), pentru un subset al datelor de intrare posibile, fără a putea prezice performanța pentru toate datele de intrare posibile (aceasta reprezintă un dezavantaj al metodei). Timpul de execuție va fi exprimat numeric, ca număr efectiv de secunde necesare procesării.
- Analiza **asimptotică** (sau **matematică**) este analiza în care surprindem nu timpii exacți de execuție (aceasta ar putea reprezenta un dezavantaj al metodei), ci ordinul de creștere al timpului de execuție, pentru toate datele de intrare posibile (aceasta reprezintă un avantaj al metodei).

Complexitatea unui algoritm poate să depindă și de valorile datelor de intrare, nu doar de dimensiunea lor. Prin urmare, distingem următoarele tipuri de analiză de complexitate:

- Analiza complexității în cazul defavorabil, adică pentru date de intrare defavorabile (care implică număr maxim de operații). Această analiză este importantă deoarece oferă o garanție, aceea că algoritmul nu se va purta mai prost, indiferent de valorile datelor de intrare.

- Analiza complexității în cazul favorabil, adică pentru date de intrare favorabile (care implică număr minim de operații), potrivită, mai degrabă, pentru probleme în care datele de intrare tind să fie favorabile.

- Analiza complexității în cazul mediu, adică pentru date de intrare **aleatorii**. O astfel de analiză este în mod particular utilă, însă este, totodată, mai dificil de realizat. De ce? Deoarece necesită cunoașterea (altfel, presupunerea) unei distribuții statistice a valorilor datelor de intrare pentru a calcula media ponderată cu probabilitățile lor de apariție.

Notații asimptotice de complexitate

Pentru clase de complexitate avem următoarele notații asimptotice: O, Ω, Θ.

O

Definiții:

- **T(n)** $\in$ **O(f(n))** ⇔ ∃ constantele **c** $\in$ **R₊**, **c>0**, si **n₀** $\in$ **N** a.i. $0 \le T(n) \le c \times f(n)$, pentru orice $n \ge n_0$
- **T(n)** $\in$ **O(f(n))** ⇔ $\lim_{n\to\infty}\frac{T(n)}{f(n)} = 0$ sau constantă nenulă, dar nu ∞

Semnificație: pentru valori (suficient de) mari ale dimensiunii intrării, c × f(n) este o limita superioara pentru T(n), algoritmul purtându-se mereu mai bine decât această limită.

Ω

Definiții:

- **T(n)** $\in$ **Ω(f(n))** ⇔ ∃ constantele **c** $\in$ **R₊**, **c>0**, si **n₀** $\in$ **N** a.i. $0 \le c \times f(n) \le T(n)$, pentru orice $n \ge n_0$
- **T(n)** $\in$ **Ω(f(n))** ⇔ $\lim_{n\to\infty}\frac{T(n)}{f(n)} =$ constantă **nenulă** sau ∞

Semnificație: pentru valori (suficient de) mari ale dimensiunii intrării, c × f(n) este o limită inferioară pentru T(n), algoritmul purtându-se mereu mai prost decât această limită.

Θ

Definiții:

- **T(n)** $\in$ **Θ(f(n))** ⇔ ∃ constantele $c_1, c_2 \in R_+$, $c_1>0$, $c_2>0$, si $n_0 \in N$ a.i. $0 \le c_1 \times f(n) \le T(n) \le c_2 \times f(n)$, pentru orice $n \ge n_0$
- **T(n)** $\in$ **Θ(f(n))** ⇔ $\lim_{n\to\infty}\frac{T(n)}{f(n)} =$ constantă **nenulă, finită**

Semnificație: pentru valori (suficient de) mari ale dimensiunii intrării, $c_1 \times f(n)$ este o limită inferioară pentru T(n), iar $c_2 \times f(n)$ este o limită superioară

Observație:

Să ne amintim complexitățile algoritmilor cunoscuți de căutare și sortare...

Algoritm	Complexitate timp				Complexitate spațiu
	CF	CD	CM	Total	
Căutare secvențială	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$O(n)$	$\Theta(1)$
Căutare binară	$\Theta(1)$	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$	$O(\log_2 n)$	$\Theta(1)$
Sortare prin selecție	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$ – *in place**
Sortare prin inserție	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(n^2)$	$\Theta(1)$ – *in place*
Sortare prin metoda bulelor	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(n^2)$	$\Theta(1)$ – *in place*
Sortare rapidă (*QuickSort*)	$\Theta(n \log_2 n)$	$\Theta(n^2)$	$\Theta(n \log_2 n)$	$O(n^2)$	$\Theta(1)$ – *in place*
Sortare prin interclasare (*Merge Sort*)	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n)$ - *out of place*

*Algoritmii de sortare *in place* sortează șirul fără a folosi structuri de date suplimentare, ci doar spațiu de memorie adițional constant, pentru variabilele auxiliare. De exemplu, un algoritm de sortare care efectuează ordonarea doar prin interschimbări de elemente este *in place.* Un algoritm care nu este *in place* este *out of place.*

Exerciții

1. Adevărat sau fals?
   a. $n^2 \in O(n^3)$
   b. $n^3 \in O(n^2)$
   c. $2^{n+1} \in \theta(2^n)$
   d. $n^2 \in \theta(n^3)$
   e. $2^n \in O(n!)$
   f. $\log_{10} n \in \theta(\log_2 n)$
   g. $O(n) + \theta(n^2) = \theta(n^2)$
   h. $\Theta(n) + O(n^2) = O(n^2)$
   i. $O(n) + O(n^2) = O(n^2)$
   j. $O(f) + O(g) = O(\max\{f,g\})$
   k. $O(n) + \theta(n) = O(n)$

Soluții:

a. $n^2 \in O(n^3)$ - Adevărat
b. $n^3 \in O(n^2)$ – Fals
c. $2^{n+1} \in \theta(2^n)$ – Adevărat
d. $n^2 \in \theta(n^3)$ – Fals
e. $2^n \in O(n!)$ - Adevărat
f. $\log_{10} n \in \theta(\log_2 n)$ - Adevărat

g. O(n) + θ(n^2) = θ (n^2) - Adevărat
h. Θ(n) + O (n^2) = O(n^2) – Adevărat
i. O(n) + O(n^2) = O(n^2) – Adevărat
j. O(f) + O(g) = O (max {f,g}) – Adevărat
k. O(n) + θ(n) = O(n) – Adevărat, dar preferăm O(n) + θ(n) = θ(n), întrucât este mai exact

2. Construiți un algoritm cu timpul θ(n $\log_2$n)

Exemplu de soluție :

```
Pentru i = 1, n execută
      j ← n
      Câttimp j ≠ 0 execută
            j = [j/2]
      sf_câttimp
sf_pentru
```

3. Calculați complexitatea pentru următorii 2 algoritmi:

a)

```
gasit ← fals
Pentru i ← 1, n executa
      Dacă xi = a atunci
            gasit ← adevarat
      sf_daca
sf_pentru
```

Soluție:

A se observa că ciclul se execută independent de valoarea variabilei *găsit*.

$$\left.\begin{aligned} CF&: \theta(n) \\ CD&: \theta(n) \end{aligned}\right\} => \theta(n)$$

b)

```
gasit ← fals
i ← 1
câttimp gasit = fals și i < n execută
      Dacă xi = a atunci
            gasit ← adevarat
      sf_daca
      i ← i + 1
sf_cattimp
```

CF : θ(1)
CD: θ(n)
CM: sunt n + 1 cazuri posibile (elementul de găsește pe oricare dintre cele n poziții sau pe niciuna)

$$T(n) = \sum_{I \in D} P(I) * E(I) = \frac{1}{n+1} + \frac{2}{n+1} + \dots + \frac{n+1}{n+1} = \frac{(n+1)*(n+2)}{2*(n+1)} = \frac{n+2}{2} \in \theta(n)$$

Complexitatea algoritmului este, prin urmare, O(n) .

4. Fie X este un șir de *n* numere naturale, fiecare element fiind <= *n* și următorul algoritm.

```
k← 0
Pentru i = 1, n execută
      Pentru j = 1, xi execută
            * k ← k + xj
   sf_pentru
sf_pentru
```

Observând că operația * se efectuează de un număr de ori egal cu suma elementelor șirului, este corect să determinăm și să exprimăm complexitatea timp precum mai jos ?

$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{xi} 1 = \sum_{i=1}^{n} x_i = s\ (suma\ elementelor)$ ϵ θ(s)

Soluție:

Exemplificăm un șir care respectă proprietatea enunțată, fiind definit astfel:

Fie $x_i = \begin{cases} 1, dacă\ i\ este\ pătrat\ perfect \\ 0, altfel \end{cases}$

Deducem că suma elementelor șirului este s = [√n], deci, conform calculului de mai sus, complexitatea este θ(√n), deci este sub-liniară. Dar ciclul exterior efectuează n pași. Survine, astfel, o contradicție.

Prin urmare, exprimarea corectă a complexității este:

T(n) ϵ θ(max {n, s})

Observații:

- Dacă elementele șirului ar fi fost numere naturale <u>nenule</u>, exprimarea inițială a complexității ar fi fost corectă

- Altfel, dacă ne gândim la cazul favorabil, acesta apare atunci când, de pildă, toate elementele sunt nule, în acest caz efectuându-se n operații, deci complexitatea fiind T(n) ϵ θ(max {n, 0}) = θ(n) , iar cazul defavorabil corespunde cazului în care toate elementele șirului au valoarea n, în acest caz complexitatea fiind T(n) ϵ θ(max {n,n^2 }) = θ(n^2).

5. Cum putem determina dacă un șir arbitrar de numere $x_1 \ldots x_n$ conține cel puțin două elemente egale în $\theta(n \log_2 n)$?

Soluție:

Se va aplica MergeSort pe șir, verificarea rezumându-se ulterior la a parcurge șirului și a verifica existența a două elemente egale, pe poziții consecutive (aceasta efectuându-se în O(n)).

6. Calculați complexitatea:

```
Pentru i = 1,n execută
      @op elementară
sf_pentru
i ← 1
k ← adevarat
Câttimp i <= n – 1 și k execută
      j ← i
      k₁ ← adevărat
      Câttimp j <= n și k₁ execută
            @ op elementară (k₁ poate fi modificat)
            j ← j + 1
      sf_câttimp
      i ← i + 1
      @op elementara (k poate fi modificat)
sf_câttimp
```

Soluție:

CF: k, k_1 poate deveni *fals* după primul pas => θ (n)

CD: k, k_1 nu devin *fals* niciodată

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i}^{n} 1 = \sum_{i=1}^{n-1} n - i + 1 = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 =$$

$$n * (n-1) - \frac{n * (n-1)}{2} + n - 1 \; \epsilon \; \theta\ (n^2)$$

CM: pentru un i fixat, k_1 poate deveni *fals* după 1,2, ..., n-i+1 execuții.

Prob: $\frac{1}{n-i+1}$

$$\frac{1}{n-i+1} + \frac{2}{n-i+1} + \dots + \frac{n-i+1}{n-i+1} = \frac{(n-i+1)*(n-i+2)}{2\,(n-i+1)} = \frac{(n-i+2)}{2}$$

Pentru ciclul *Câttimp* exterior, k poate deveni *fals* după 1, 2, ..., n-1 iterații.

$$T(n) = \frac{1}{n-1}*\frac{n-1+2}{2} + \frac{2}{n-1}*\frac{n-2+2}{2} + \dots + \frac{n-1}{n-1}*\frac{n-(n-1)+2}{2} =$$

$$\frac{1}{2*(n-1)}*\sum_{i=1}^{n-1} i*(n-i+2) = \dots$$

$$= \frac{1}{2*(n-1)}*\left(\frac{n*(n-1)*n}{2} - \frac{(n-1)*n*(2n-1)}{6} + 2*\frac{(n-1)*n}{2}\right)$$

$$= \frac{1}{2}*\left(\frac{n^2}{2} - \frac{2*n^2-n}{6} + n\right) = \frac{1}{2}*\left(\frac{3n^2-2n^2+5n}{6}\right) \in \theta(n^2)$$

Complexitatea totală: $O(n^2)$

7. Calculați complexitatea:

```
Subalg p(x,s,d) este:
      Daca s < d atunci
            m ← [(s+d)/2]
            Pentru i = s, d-1, execută
                  @op. Elementare
            sf_pentru
            Pentru i = 1,2 execută
                  p(x, s, m)
            sf_pentru
      sf_daca
sf_subalg

apel: p(x, 1, n)
```

Soluție:

$$T(n) = \begin{cases} 2*T\left(\frac{n}{2}\right) + n, \text{daca } n > 1 \\ 0, \quad altfel \end{cases}$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n \qquad\qquad n = 2^k$$

$$T(2^k) = 2*T(2^{k-1}) + 2^k$$
$$2*T(2^{k-1}) = 2^2*T(2^{k-2}) + 2^k$$
$$2^2*T(2^{k-2}) = 2^3*T(2^{k-3}) + 2^k$$
$$\dots$$
$$2^{k-1}*T(2) = 2^k*T(1) + 2^k$$

$$T(2^k) = 2^k * T(1) + k * 2^k = k * 2^k = n * log_2 n \rightarrow T(n) \in \theta(n \, log_2 n)$$

8. Calculați complexitatea:

```
s ← 0
Pentru i = 1, n² execută
      j ← i
      Câttimp j ≠ 0 execută
            s ← s + j
            j ← j - 1
      sf_câttimp
sf_pentru
```

Soluție:

$$T(n) = \sum_{i=1}^{n^2} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n^2} i = \frac{n^2 * (n^2 + 1)}{2} \in \theta(n^4)$$

9. Calculați complexitatea:

```
s ← 0
pentru i = 1, n² execută
      j ← n
      Câttimp j ≠ 0 execută
            s ← s + j - 10 * [j/10]
            j ← [j/10]
      sf_câttimp
sf_pentru
```

Soluție:

- Ciclul *Câttimp* se execută de log_{10} n ori.
- T(n) ϵ θ(n^2 log_{10}n) => T(n) ϵ θ(n^2 log_2n)

10. Calculați complexitatea:

```
Subalg operație(n, i) este
      Daca n > 1 atunci
            i ← 2 * i
            m ← [n/2]
            operatie (m, i-2)
            operatie (m, i-1)
            operatie (m, i+2)
            operatie (m, i+1)
      altfel
            scrie i
      sf_daca
```

```
sf_subalg
```

$$T(n) = \begin{cases} 4 * T\left(\frac{n}{2}\right) + 2, n > 1 \\ 1,\ altfel \end{cases}$$

Soluție:

$$T(n) = 4 * T\left(\frac{n}{2}\right) + 2$$
$$T\left(2^k\right) = 4 * T\left(2^{k-1}\right) + 2$$
$$4 * T\left(2^{k-1}\right) = 4^2 * T\left(2^{k-2}\right) + 4 * 2$$
$$4^2 * T\left(2^{k-2}\right) = 4^3 * T\left(2^{k-3}\right) + 4^2 * 2$$
$$\dots$$
$$4^{k-1} * T(2) = 4^k * T(1) + 4^{k-1} * 2$$
$$T(n) = 4^k + 2 * \left(4 + 4^2 + \dots + 4^{k-1}\right) = 4^k + 2 * \frac{4^k - 2}{3} = n^2 + 2 * \frac{n^2 - 2}{3} \in \theta(n^2)$$