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 (acesta este un avantaj al metodei), pentru un subset al datelor de intrare posibile, fără a putea prezice performanța pentru toate datele de intrare posibile (acesta este 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 (acesta este un dezavantaj al metodei), ci ordinul de creștere al timpului de execuție, pentru toate datele de intrare posibile (acesta este 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.

Definiții:

- **T(n) ϵ O(f(n))** <=> ∃ constantele **c ϵ R_+, c>0** și **n_0 ϵ N** a.i. **0 ≤ T(n) ≤ c * f(n)** pentru orice **n ≥ n_0**
- **T(n) ϵ O(f(n))** dacă $\lim_{n \to \infty} \frac{T(n)}{f(n)} = 0$ sau o constantă nenulă

Semnificație: pentru valori (suficient de) mari ale dimensiunii intrării, c * f(n) este o limita superioara pentru T(n)

- **T(n) ϵ Ω(f(n))** <=> ∃ constantele **c ϵ R_+, c>0 și n_0 ϵ N** a.i. **0 ≤ c * f(n) ≤ T(n)** pentru orice **n ≥ n_0**
- **T(n) ϵ Ω(f(n))** dacă $\lim_{n \to \infty} \frac{T(n)}{f(n)} = \infty$ sau o constantă nenulă

Semnificație: pentru valori (suficient de) mari ale dimensiunii intrării, c * f(n) este o limita inferioară pentru T(n)

- **T(n) ϵ Θ(f(n))** <=> ∃ constantele **c ϵ R_+, c_1, c_2>0 și n_0 ϵ N** a.i. **0 ≤ c_1 * f(n) ≤ T(n) ≤ c_2 * f(n)** pentru orice **n ≥ n_0**
- **T(n) ϵ Θ(f(n))** dacă $\lim_{n \to \infty} \frac{T(n)}{f(n)}$ este o constantă nenulă

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

Complexitățile algoritmilor cunoscuți de căutare și sortare:

Algoritm	Complexitate Timp				Complexitate Spațiu (extra)
	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
Quick Sort	$\Theta(n \log_2 n)$	$\Theta(n^2)$	$\Theta(n \log_2 n)$	$O(n^2)$	$\Theta(1)$ – 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*.

1. Adevărat sau Fals?
    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. $O(n) + \Theta(n^2) = \Theta(n^2)$ - Adevărat
    h. $\Theta(n) + O(n^2) = O(n^2)$ – Adevărat
    i. $O(n) + O(n^2) = O(n^2)$ – Adevărat
    j. $O(f) + O(g) = O(\max\{f, g\})$ – Adevărat
    k. $O(n) + \Theta(n) = O(n)$ – Adevărat, dar $\Theta(n)$ este mai exact

2. Construiți un algoritm având complexitatea timp $\Theta(n \log_2 n)$.

<u>Exemplu de soluție:</u>

```
pentru i ← 1, n execută
     j ← n
     câttimp j ≠ 0 execută
          j ← [j/2]
     sf_câttimp
sf_pentru
```

3. Calculați complexitatea timp pentru următorii 2 algoritmi:

a)

```
gasit ← fals
pentru i ← 1, n execută
     dacă xi = a atunci
          gasit ← adevarat
     sf_dacă
sf_pentru
```

<u>Soluție:</u>

Se observă că ciclul se execută independent de valoarea variabilei *găsit*.

$$\left.\begin{array}{l} CF:\theta(n) \\ CD:\theta(n) \end{array}\right\} = \Theta(n)$$

b)

```
gasit ← fals
i ← 1
câttimp gasit = false și i ≤ n execută
     dacă xi = a atunci
          gasit ← adevarat
     sf_dacă
     i ← i + 1
sf_câttimp
```

CF: Θ (1)
CD: Θ (n)
CM: sunt n+1 cazuri posibile (elementul se gasește pe oricare dintre cele n poziții sau pe niciuna). Presupunem că toate aceste cazuri au probabilitate egală.

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

Complexitate totală: O(n)

4. Fie X este un șir de n numere naturale, fiecare element fiind <= n. Se dă 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 ?

$$T(n)=\sum_{i=1}^{n}\sum_{j=1}^{x_i}1=\sum_{i=1}^{n}x_i=s\,(suma\,elementelor)\,\epsilon\,\Theta(s)$$

Exemplificăm un șir care respectă proprietatea enunțată:

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}) = Θ (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 θ(n log 2 n)?

<u>Soluție:</u>
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 timp a urmatorului algoritm:

```
pentru i ← 1,n execută
     @ operatie elementara
sf_pentru
i ← 1
k ← adevarat
câttimp i <= n – 1 și k execută
     j ← i
     k₁ ← adevarat
     câttimp j <= n și k₁ execută
          @ operatie elementara (k₁ poate fi modificat)
          j ← j + 1
     sf_câttimp
     i ← i + 1
     @ operatie elementara (k poate fi modificat)
sf_câttimp
```

<u>Soluție:</u>

CF: k, k_1 pot deveni *fals* după primul pas => Θ (n) (din cauza primului “*pentru*”)

CD: k, k_1 nu devin *fals* niciodată

$$T(n)=n+\sum_{i=1}^{n-1}\sum_{j=i}^{n}1=n+\sum_{i=1}^{n-1}n-i+1=n+\sum_{i=1}^{n-1}n-\sum_{i=1}^{n-1}i+\sum_{i=1}^{n-1}1=n+n*(n-1)-\frac{n*(n-1)}{2}+n-1\,\epsilon\,\Theta(n^2)$$

CM:

Considerăm pentru început ciclul *câttimp* interior. Pentru un i fixat, k_1 poate deveni *fals* dupa 1,2, ..., n-i+1 iterații =>probabilitate: $\frac{1}{n-i+1}$

Astfel, numărul mediu de pași executați de ciclul *câttimp* interior este

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

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

probabilitate: $\frac{1}{n-1}$

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

$$\ldots=\frac{1}{2*(n-1)}*\left(\frac{n*(n-1)*n}{2}-\frac{(n-1)*n*(2n-1)}{6}+\frac{2*(n-1)*n}{2}\right)=\frac{1}{2}*\left(\frac{n^2}{2}-\frac{2*n^2-n}{6}+n\right)=\frac{n^2+7n}{12}\in\Theta(n^2)$$

Complexitate totală: $O(n^2)$

7. Calculați complexitatea următorului algoritm:

```
subalgoritm p(x,s,d) este:
   daca s < d atunci
         m ← [(s+d)/2]
         pentru i ← s, d-1, executa
               @operatie elementara
         sf_pentru
         pentru i ← 1,2 executa
               p(x, s, m)
         sf_pentru
   sf_daca
sf_subalgoritm

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

<u>Solutie:</u>

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

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

Presupunem: $n=2^k$

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

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

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

…

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

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

8. Calculați complexitatea timp a următorului algoritm:

```
s ← 0
pentru i ← 1, n² executa
     j ← i
     cattimp j ≠ 0 executa
          s ← s + j
          j ← j – 1
     sf_cattimp
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 timp a următorului algoritm:

```
s ← 0
pentru i ← 1, n² executa
     j ← n
     cattimp j ≠ 0 executa
          s ← s + j – 10 * [j/10]
          j ← [j/10]
     sf_cattimp
sf_pentru
```

Soluție:

- Ciclul Câttimp se execută de $\log_{10} n$ ori.
- $T(n) \in \Theta(n^2 \log_{10} n) \Rightarrow T(n) \in \Theta(n^2 \log_2 n)$

10. Calculați complexitatea timp a următorului algoritm:

```
subalgoritm 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
```

Solutie:

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

Presupunem: $n=2^k$

$T(2^k)=4*T(2^{k-1})+1$

$4*T(2^{k-1})=4^2*T(2^{k-2})+4$

$4^2*T(2^{k-2})=4^3*T(2^{k-3})+4^2$

...

$4^{k-1}*T(2)=4^k*T(1)+4^{k-1}$

$$T(2^k)=1+4+...+4^{k-1}+4^k*T(1)=\frac{4^{k+1}-1}{3}=\frac{4*n^2-1}{3}\in\Theta(n^2)$$