Complexitatea Algoritmilor

Drd. Horea Mureşan

14 Ianuarie 2023

1 Introducere

Analiza complexităţii unui algorim are ca scop estimarea volumului de resurse de calcul necesare execuţiei acestuia în funcţie de dimensiunea datelor de intrare. Resursele sunt:

- spaţiul de memorie - pentru stocarea datelor prelucrate de algoritm
- timpul de execuţie - necesar pentru execuția operaţiilor

1.1 Considerente teoretice

Complexitatea spaţiu depinde de tipurile şi structurile de date folosite, iar complexitatea timp depinde de numărul de operaţii pe care le face algoritmul. Valorile concrete are complexităţilor depind de sistemul care rulează algoritmul. Pentru a putea evalua comparativ performanţa algoritmilor este necesar un model teoretic de calculator, independent de componentele fizice ale unei maşini de calcul, de particularităţile limbajelor de programare şi de stiluri de implementare ale programatorilor.

Acest model teoretic se numește model RAM (Random Access Machine). Acest model este unul simplificat, dar este adecvat pentru aproximarea performanţei algoritmilor şi are următoarele caracteristici:

- prelucrările se efectuează în mod secvenţial
- operaţiile elementare sunt efectuate în timp constant (o unitate de timp) indiferent de valoarea operanzilor
- timpul de accesare a informaţiilor nu depinde de poziţia acestora (primul element al unui şir se va accesa în acelaşi timp ca oricare alt element al şirului)

Astfel, complexitatea timp a unui algoritm depinde de numărul de operaţii elementare efectuate de acesta. Operații elementare:

- operaţii de atribuire
- operaţii aritmetice, de comparaţie şi logice
- operaţii de intrare/ieşire

1.2 Tipuri de analiză

- Cazul favorabil
  - Scenariul în care algoritmul rulează cel mai rapid (execută numărul minim de operaţii)
- Cazul defavorabil
  - Scenariul în care algoritmul rulează cel mai lent (execută numărul maxim de operaţii)
- Cazul mediu
  - Presupune că datele de intrare au dimensiuni aleatoare
  - Estimează performaţa unui algoritm în cazul general

2 Notaţii

2.1 Notaţia O (Big-O Notation)

Notația Big-O furnizează o limită superioară pentru ordinul timpului de execuţie, astfel se poate descrie complexitatea timp a unui algoritm în cazul defavorabil.

Notăm cu $T(n)$ timpul de execuţie al unui algoritm, unde n reprezintă dimensiunea datelor de intrare. Spunem că $T(n) \in O(f(n))$ sau $T(n) = O(f(n))$ dacă există constantele $c > 0$ şi n_0, independente de n, astfel încât

$$0 \leq T(n) \leq c \cdot f(n), \forall n \geq n_0$$

Altfel spus:

$$\lim_{n\to\infty} \frac{T(n)}{f(n)} = k, k \geq 0$$

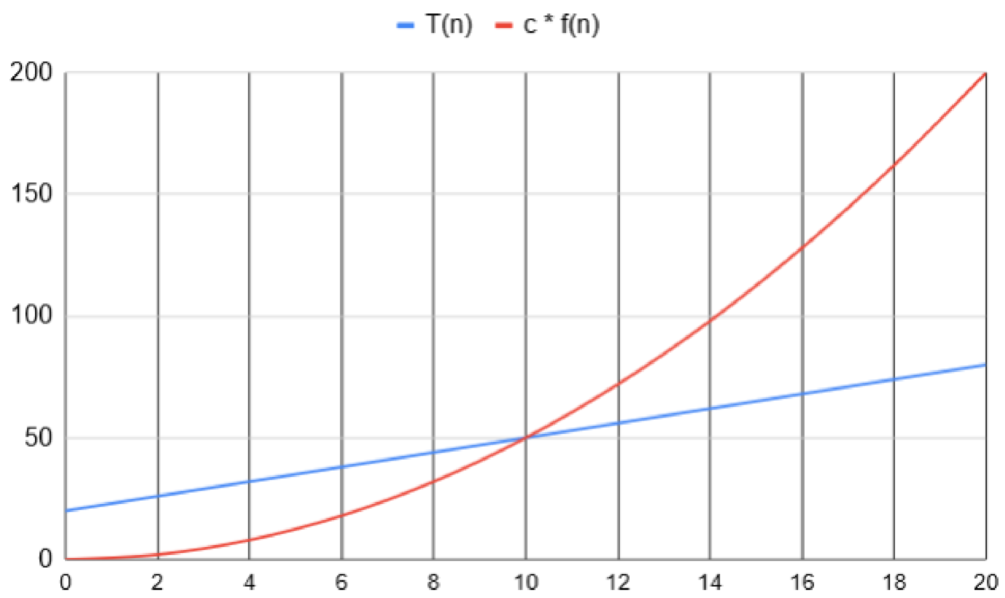

Fig. 1: Exemplu notaţie Big-O. Pentru $n_0 = 10$, $T(n)$ este mărginită superior de $c \cdot f(n), \forall n \geq n_0$.

2.2 Notaţia Ω

Notaţia Ω furnizează o limită inferioară pentru ordinul timpului de execuţie, astfel se poate descrie complexitatea timp a unui algoritm în cazul favorabil.

Notăm cu $T(n)$ timpul de execuţie al unui algoritm, unde n reprezintă dimensiunea datelor de intrare. Spunem că $T(n) \in \Omega(f(n))$ sau $T(n) = \Omega(f(n))$ dacă există constantele $c > 0$ şi n_0, independente de n, astfel încât

$$0 \leq c \cdot f(n) \leq T(n), \forall n \geq n_0$$

Similar cu notaţia Big-O, sub formă de limită:

$$\lim_{n\to\infty} \frac{f(n)}{T(n)} = k, k \geq 0$$

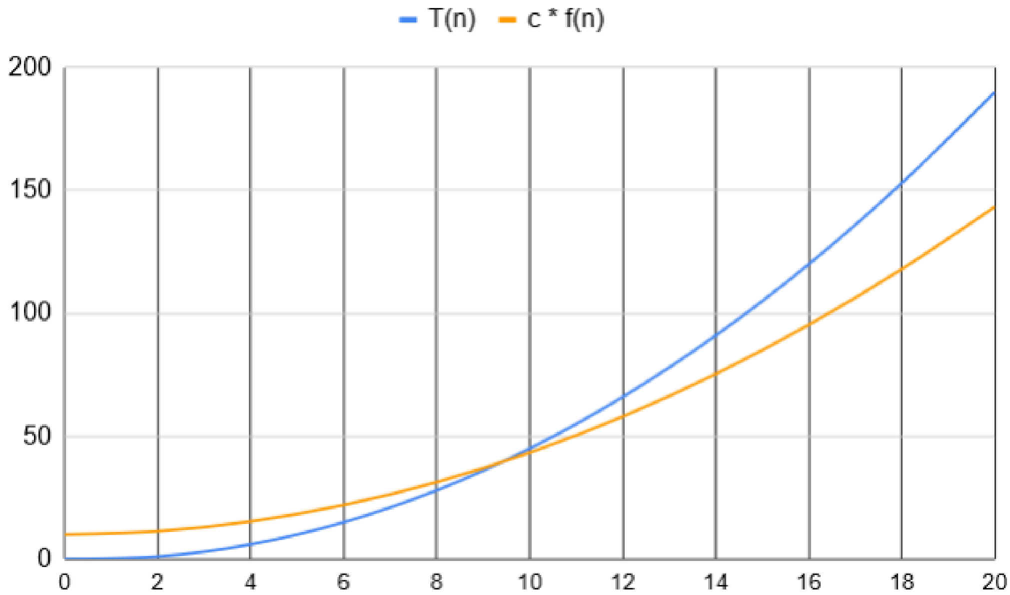

Fig. 2: Exemplu notaţie Ω. Pentru $n_0 = 10$, $T(n)$ este mărginită inferior de $c \cdot f(n), \forall n \geq n_0$.

2.3 Notaţia Θ

Notaţia Θ furnizază atât o margine superioară cât şi o margine inferioară pentru ordinul timpului de execuţie a unui algoritm.

Notăm cu $T(n)$ timpul de execuţie al unui algoritm, unde n reprezintă dimensiunea datelor de intrare. Spunem că $T(n) \in \Theta(f(n))$ sau $T(n) = \Theta(f(n))$ dacă există constantele $c1, c2 > 0$ şi n_0, independente de n, astfel încât

$$0 \leq c1 \cdot f(n) \leq T(n) \leq c2 \cdot f(n), \forall n \geq n_0$$

Aceeaşi relaţi, exprimată sub formă de limită:

$$\lim_{n \to \infty} \frac{T(n)}{f(n)} = k, k \in (0, \infty)$$

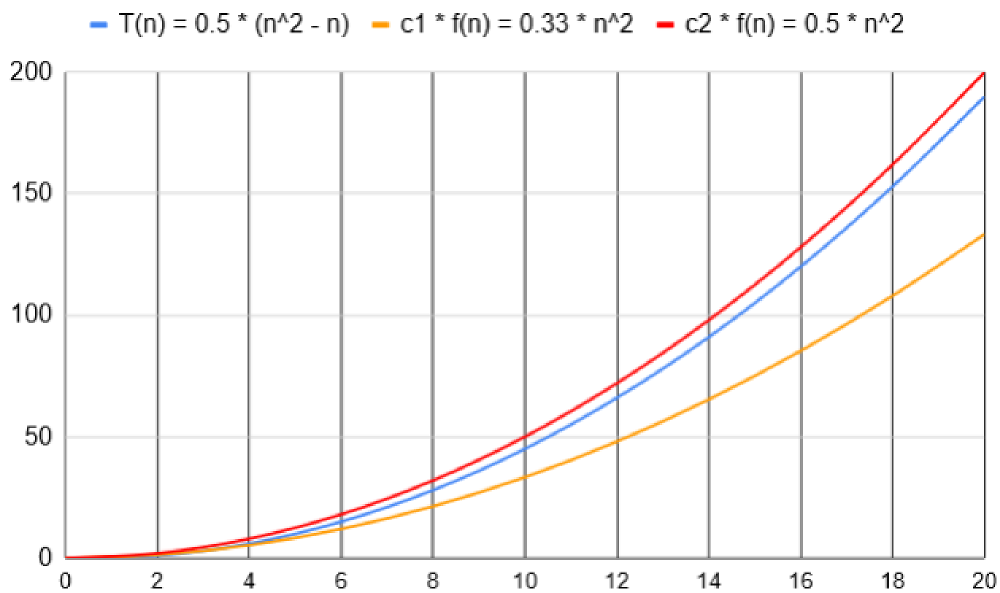

Fig. 3: Exemplu notaţie Θ. Pentru $n_0 = 0$, $T(n)$ este mărginită inferior de $c1 \cdot f(n), \forall n \geq n_0$ şi este mărginită superior de $c2 \cdot f(n), \forall n \geq n_0$.

2.4 Proprietăţi

1. Dacă $T(n) = \sum_{i=0}^{k} a_i \cdot n^i$, unde $a_k > 0$, atunci $T(n) \in O(n^p), \forall p \geq k$
   - $n^2 \in O(n^3)$, $n^2 \in O(n^4)$, etc.
   - $2^n \in O(n!)$
2. Reflexivitate: $f(n) \in O(f(n))$, $f(n) \in \Omega(f(n))$, $f(n) \in \Theta(f(n))$
   - $n^2 \in O(n^2)$
   - $n^2 \in \Omega(n^2)$
   - $n^2 \in \Theta(n^2)$
3. Tranzitivitate: dacă $f(n) \in O(g(n))$ şi $g(n) \in O(h(n))$ atunci $f(n) \in O(h(n))$, similar pentru Ω, Θ
4. Simetrie:
   - $f(n) \in \Theta(g(n))$ dacă şi numai dacă $g(n) \in \Theta(f(n))$
   - $f(n) \in O(g(n))$ dacă şi numai dacă $g(n) \in \Omega(f(n))$
5. $O(f(n) + g(n)) = O(max(f(n), g(n)))$, clasa de complexitate a unei funcţii este clasa de complexitate a termenului dominant
   - $n^4 + 10 \cdot n^2 + 2 \in O(n^4)$
   - $n + \log_2(n^2) \in O(n)$

Exemplu

Problema 1 Să se calculeze suma primelor n numere naturale nenule.

1: $S \leftarrow 0$
2: $i \leftarrow 0$
3: **while** $i < n$ **do**
4: $\quad i \leftarrow i + 1$
5: $\quad S \leftarrow S + i$
6: **end while**

Operaţie	Cost	Număr repetiţii
1	1	1
2	1	1
3	1	n + 1
4	2	n
5	2	n

Tabel 1: Costurile fiecărei operaţii elementare şi numărul de repetiţii al fiecăreia pentru Problema 1

În Problema 1, fiecare liniile 1, 2 şi 3 conţin cate o operaţie elementară, deci timpul de execuţie pentru fiecare din ele este 1 unitate de timp. Liniile 4 şi 5 conţin câte 2 operaţii elementare, deci timpul de execuţie pentru fiecare este 2 unităţi de timp.Însă liniile 3, 4, 5 sunt repetate de mai multe ori. Liniile 4, 5 se repetă de n ori, iar linia 3 se repetă de $n + 1$ ori (este necesar să se execute şi verificarea $n < n$ pentru a putea decide că bucla se încheie). Complexitatea timp totală se obţine prin însumarea complexităţilor individuale:

$$T(n) = 1 + 1 + (n + 1) + 2n + 2n = 5n + 3 \in \Theta(n)$$

3 Analiza cazurilor extreme

Numărul de operaţii sau repetiţii ale unei operaţii depinde uneori şi de alţi factori în afara dimensiunii datelor de intrare. Astfel, pentru mai multe cazuri ale aceleiaşi probleme se pot executa un număr diferit de operaţii. În aceste situaţii se urmăreşte determinarea unor margini ale numărului de operaţii:

- Cazul favorabil - în care se execută cel mai mic număr de operaţii
- Cazul nefavorabil - în care se execută cel mai mare număr de operaţii

Problema 2 Să se verifice dacă şirul de numere întregi S conţine un număr întreg X. Fie n lungimea şirului S.

```
1: i ← 1
2: while i ≤ n do
3:     if S[i] = X then
4:         return True
5:     end if
6:     i ← i + 1
7: end while
8: return False
```

Operaţie	Cost	Număr repetiţii
1	1	1
2	1	$1 \leq f1(n) \leq n+1$
3	1	$1 \leq f2(n) \leq n$
4	1	$0 \leq f3(n) \leq 1$
6	2	$0 \leq f4(n) \leq n$
8	1	$0 \leq f5(n) \leq 1$

Tabel 2: Costurile fiecărei operaţii elementare şi numărul de repetiţii al fiecăreia pentru Problema 2

$$f1(n) = \begin{cases} k & \text{X se află pe poziţia k} \\ n+1 & \text{X nu se află în şir} \end{cases}$$

$$f2(n) = \begin{cases} k & \text{X se află pe poziţia k} \\ n & \text{X nu se află în şir} \end{cases}$$

$$f3(n) = \begin{cases} 1 & \text{X se află pe poziţia k} \\ 0 & \text{X nu se află în şir} \end{cases}$$

$$f4(n) = \begin{cases} 2k-2 & \text{X se află pe poziţia k} \\ 2n & \text{X nu se află în şir} \end{cases}$$

$$f5(n) = \begin{cases} 0 & \text{X se află pe poziţia k} \\ 1 & \text{X nu se află în şir} \end{cases}$$

$$T(n) = \begin{cases} 4k-1 & \text{X se află pe poziţia k} \\ 4n+2 & \text{X nu se află în şir} \end{cases}$$

4 Analiza cazului mediu

Pentru unele probleme, atât cazul favorabil cât şi cel nefavorabil apar cu o frecvenţă mică. În această situaţie se poate estima complexitatea timp medie de execuţie. Pentru aceasta se foloseşte o medie ponderată unde valorile sunt complexităţile cazurilor posibile şi ponderile sunt probabilităţile aferente acestora. Fie C numărul total de cazuri (situaţii în care algoritmul efectuează acelaşi număr de operatţii). Definim $T_k(n)$, complexitatea timp şi $P(k)$, probabilitatea de apariţie a cazului k, unde $1 \leq k \leq C$. Complexitatea timp medie, notată $T_m(n)$ se calculează:

$$T_m(n) = \sum_{k=1}^{C} P(k) \cdot T_k(n)$$

În particular, dacă fiecare caz are aceeaşi probabilitate, atunci formula devine media aritmetică între complexităţile timp ale tuturor cazurilor.

Revenind la Problema 2, fie P probabilitatea ca elementul X să fie în şir. Atunci probabilitatea ca acesta să fie pe una din cele n poziţii ale şirului este $\frac{P}{n}$. Iar cazul în care X nu se află în şir are probabilitatea $1-P$. Timpul mediu $T_m(n)$ se va calcula astfel:

$$T_m(n) = \frac{P}{n}\sum_{k=1}^{n}(4k-1) + (1-P)\cdot(4n+2) = \frac{4P}{n}\cdot\frac{n^2+n}{2} - P + n(4-4P) + 2 - 2P =$$

$$T_m(n) = 2n(2-P) + 2 + P$$

Alternativ, putem presupune că probabilitatea ca elementul X să nu fie în şir este egală cu probabilitatea ca acesta să fie pe una dintre poziţiile din şir. În acest caz, probabilitatea fiecărui eveniment este de $\frac{1}{n+1}$. Atunci timpul mediu de execuţie este:

$$T_m(n) = \frac{1}{n+1}(\sum_{k=1}^{n}(4k-1) + 4n + 2) = \frac{2n^2+5n+2}{n+1}$$

5 Teorema Master (Divide and Conquer Master Theorem)

Fie $T(n)$ complexitatea unui algoritm recursiv de tipul Divide and Conquer. Complexitatea unui astfel de algoritm poate fi scrisă sub forma:

$$T(n) = a * T(n/b) + \Theta(n^k * log^p(n))$$

Clasa de complexitate a algoritmului poate fi determinată astfel:

1. $a > b^k \implies T(n) = \Theta(n^{log_b(a)})$
2. $a = b^k$
   (a) $p > -1 \implies T(n) = \Theta(n^{log_b(a)} \cdot log^{p+1}(n))$
   (b) $p = -1 \implies T(n) = \Theta(n^{log_b(a)} \cdot log(log(n)))$
   (c) $p < -1 \implies T(n) = \Theta(n^{log_b(a)})$
3. $a < b^k$
   (a) $p \geq 0 \implies T(n) = \Theta(n^k \cdot log^p(n))$
   (b) $p < 0 \implies T(n) = O(n^k)$

6 Probleme

Problema 3 Precizaţi complexitatea timp pentru următorul algoritm.

```
procedure F(n)
    for i ← 1, n do
        for j ← 1, n do
            print(i + j)
        end for
    end for
end procedure
```

Problema 4 Care din următorii algoritmi calculează corect suma numerelor de pe diagonala principală şi suma numerelor de pe diagonala secundară a unei matrici M cu n linii şi n coloane. Specificaţi complexitatea fiecărui algoritm.

A

```
procedure P(M, n)
    S1 ← 0
    S2 ← 0
    for i ← 1, n do
        for j ← 1, n do
            if i = j then
                S1 ← S1 + M[i][j]
            end if
            if i + j = n + 1 then
                S2 ← S2 + M[i][j]
            end if
        end for
    end for
end procedure
```

B

```
procedure P(M, n)
    S1 ← 0
    S2 ← 0
    for i ← 1, n do
        S1 ← S1 + M[i][i]
        S2 ← S2 + M[i][n − i + 1]
    end for
end procedure
```

Problema 5 Precizaţi complexitatea timp pentru următorul algoritm.

```
procedure F(n)
    s ← 0
    for i ← 1, n do
        j ← 1
        while j < n do
            j ← j * 2
        end while
        s ← s + j
    end for
end procedure
```

Problema 6 Precizaţi complexitatea timp pentru următorul algoritm.

```
procedure F(n)
    if n ≤ 1 then
        print(1)
    else
        m ← [n/2]                                  ▷ Partea întreagă a lui n/2
        F(m)
        F(m)
    end if
end procedure
```

Problema 7 Care din următorii algoritmi pot fi implementaţi astfel încât să aibă complexitatea timp O(n)?

1. Algoritmul de căutare secvenţială a unui element într-un vector cu ***n*** numere.
2. Algoritmul de sortare prin inserţie a unui tablou unidimensional cu ***n*** numere.
3. Algoritmul de căutare a numărului maxim într-un vector nesortat cu ***n*** numere.
4. Algoritmul de calcul a sumei elementelor de pe diagonala principală a unei matrici pătratice cu ***n*** linii şi ***n*** coloane.

Problema 8 Fie următorul algoritm având ca parametru un număr natural n.

```
function F(n)
    j ← n
    while j > 1 do
        i ← 1
        while i ≤ n do
            i ← 2 * i
        end while
        j ← [j/3]
    end while
    return j
end function
```

Din care din următoarele clase de complexitate face parte algoritmul descris la Problema 8?

A $O(\log_2(n))$

B $O(\log_2^2(n))$

C $O(\log_3^2(n))$

D $O(\log_2(\log_3(n)))$

Problema 9 Care din următorii algoritmi calculează corect $E(A, n)$ în complexitatea de timp specificată. Se presupune că x^k se calculează în $O(log(k))$, iar toate operaţiile se realizează pe tipuri de date pe 32 de biţi.

$$E(A, n) = (\sum_{i=1}^{n} A^i) \mod 2022, 1 < A < 2022, 1 < n < 2022123$$

A - Complexitate timp $O(log(n))$

```
function E(A, n)
    return (A · [(A^n − 1)/(A − 1)]) mod 2022
end function
```

B - Complexitate timp $O(log(n))$

```
function E(A, n)
    return [((A · (A^n − 1)) mod 2022)/((A − 1) mod 2022)]
end function
```

C - Complexitate timp $O(nlog(n))$

```
function E(A, n)
    raspuns ← A
    for i ← 2, n do
        raspuns ← raspuns + A^i
    end for
    return raspuns mod 2022
end function
```

D - Complexitate timp $O(log(n))$

```
function E(A, n)
    (aux1, aux2) = E1(A, n)
    return aux2
end function

function E1(A, n)
    if n = 1 then
        return (A, A)                          ▷ Returnează o pereche de numere
    end if
    if n mod 2 = 1 then
        (t1, t2) = E1(A, n − 1)
        p = (t1 ∗ A) mod 2022
        return (p, (p + t2) mod 2022)
    else
        (t1, t2) = E1(A, [n/2])
        p = (t1 ∗ t1) mod 2022
        return (p, ((1 + t1) · t2) mod 2022)
    end if
end function
```

Soluţii

Problema 3

Operaţie	Cost	Număr repetiţii
2	2	n
3	2	n^2
4	2	n^2

Tabel 3: Complexitate timp: $4n^2 + 2n \in \Theta(n^2)$

Problema 4

Ambii algoritmi calculează corect sumele de pe diagonala principală şi diagonala secundară.

Operaţie	Cost	Număr repetiţii
2	1	1
3	1	1
4	2	n
5	2	n^2
6	1	n^2
7	2	n
9	3	n^2
10	2	n

Tabel 4: Complexitate timp soluţie A: $6n^2 + 6n + 2 \in \Theta(n^2)$

Operaţie	Cost	Număr repetiţii
2	1	1
3	1	1
4	2	n
5	2	n
6	2	n

Tabel 5: Complexitate timp soluţie B: $6n + 2 \in \Theta(n)$

Problema 5

Numărul de repetiţii al instrucţiunii $while$ depinde de condiţia $j < n$ unde j se dublează la fiecare iteraţie.

În mod practic se caută cel mai mic j pentru care are loc $j \geq n$ cu $j = 2^k$, unde k este numărul de repetiţii al buclei $while$. Echivalent, căutăm cel mai mic k pentru care are loc $2^k \geq n$, cu $k, n > 0$. Logaritmând ambii membri, obţinem $k \geq log_2(n)$. Astfel putem aproxima numărul de repetiţii al buclei while cu $log_2(n)$.

Operaţie	Cost	Număr repetiţii
2	1	1
3	2	n
4	1	n
5	1	$n \log_2(n)$
6	2	$n \log_2(n)$
8	2	n

Tabel 6: Complexitate timp: $3n \log_2(n) + 5n + 1 \in \Theta(n \log_2(n))$

Problema 6

Varianta 1

Avem 2 cazuri pentru calculul complexităţii:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2 + 2T(n/2) & n > 1 \end{cases}$$

Calculăm complexitatea apelului recursiv:

$$T(n) = 2(1 + T(n/2) = 2(1 + 2(1 + T(n/4)) = ... = 2(1 + 2(1 + ...2(1 + T(X)))), \text{unde } X \leq 1$$

La fiecare apel recursiv, valoarea lui n se înjumătăţeşte. Dacă am avea un singur apel recursiv, atunci complexitatea funcţiei ar fi similară cu complexitatea buclei de la Problema 5. Însă în acest caz, la fiecare execuţie, numărul de apeluri se dublează. Deci complexitatea algoritmului va fi:

$$T(n) = 2^{\log_2(n)} = n \in \Theta(n)$$

Varianta 2

Aplicând Teorema Master:

$$T(n) = a * T(n/b) + \Theta(n^k * log^p(n))$$

Identificăm constantele a, b, k şi p pe baza formulei recursive:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2 + 2T(n/2) & n > 1 \end{cases}$$

$a = 2$, $b = 2$, $k = p = 0$ deoarece termenul $\Theta(n^k * log^p(n)) = 2 = \Theta(1)$
Aplicăm cazul 1 al teoremei: $a > b^k \iff 2 > 2^0$
Astfel $T(n) = \Theta(n^{log_b(a)}) = \Theta(n^{log_2(2)}) = \Theta(n)$

Problema 7

1. Adevărat
2. Fals
3. Adevărat
4. Adevărat

Problema 8

Similar cu soluţia problemei 5 putem determina că prima buclă $while$ se repetă de $log_3(n)$ ori, iar a doua de $log_2 n$.

Operaţie	Cost	Număr repetiţii
2	1	1
3	1	$\log_3(n)$
4	1	$\log_3(n)$
5	1	$\log_2(n) \cdot \log_3(n)$
6	2	$\log_2(n) \cdot \log_3(n)$
8	2	$\log_3(n)$

Tabel 7: Complexitate timp: $3\log_2(n) \cdot \log_3(n) + 4\log_3(n) + 1$

Termenul dominant al funcţiei care descrie complexitatea algoritmului este $\log_2(n) \cdot \log_3(n)$. Vom folosi proprietatea de schimbare a bazei a funcţiei logaritm:

$$\log_a(X) = \log_a b \cdot \log_b(X)$$

Astfel, obţinem:

$$\log_2(n) \cdot \log_3(n) = \log_2(3) \cdot \log_3^2(n) \in O(\log_3^2(n))$$

Similar:

$$\log_2(n) \cdot \log_3(n) = \log_3(2) \cdot \log_2^2(n) \in O(\log_2^2(n))$$

Pentru $n > 3$ are loc:

$$\log_2(n) \cdot \log_3(n) > \log_2(n), \text{deci } T(n) \notin O(log_2(n))$$

Folosind inegalitatea $\log_a(n) < n$ avem:

$$\log_2(\log_3(n)) < \log_3(n) < \log_2(n) \cdot \log_3(n), \forall n > 2, \text{deci } T(n) \notin O(\log_2(\log_3(n)))$$

Problema 9

A **Fals** $A \cdot \frac{A^n-1}{A-1} = \sum_{i=1}^{n} A^i$, deci funcţia calculează corect din punct de vedere matematic suma cerută. Însă spaţiul de reprezentare pe 32 de biţi nu va permite calculul corect al sumei pentru orice n din intervalul specificat.

B **Fals** Spaţiul de reprezentare nu permite calculul puterilor pentru orice valori ale lui n.

C **Fals** Similar cu varianta A, matematic este corect, însă spaţiul de reprezentare nu permite calculul puterilor pentru orice valori ale lui n.

D **Adevărat** La fiecare 2 apeluri recursive consecutive ale funcţiei E1, cel puţin unul dintre apeluri va înjumătăţi pe n. Deci complexitatea algoritmului este mărginită de $2log_2(n)$ şi aparţine lui $O(\log(n))$. La fiecare apel, funcţia E1 returnează valori $\mod 2022$, deci spaţiul de reprezentare nu este depăşit. Din punct de vedere al corectitudinii matematice, algoritmul descompune suma astfel:

$$\sum_{i=1}^{n} A^i = (A^k + 1)(\sum_{i=1}^{k} A^i), \text{dacă } n = 2k$$

$$\sum_{i=1}^{n} A^i = (A^n) + \sum_{i=1}^{n-1} A^i, \text{dacă } n = 2k + 1$$

Cum funcţia $\mod$ este distributivă la adunare şi înmulţire, putem să o aplicăm pe fiecare termen al descompunerii.