**Ministerul Educației și Cercetării al Republicii Moldova Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică și Microelectronică**

**Raport**

**Lucrarea de laborator nr.1:**

**Analiza algoritmilor (Timpul de execuție al algoritmilor).**

**Elaborat: st. gr. TI-232, Galiț Ștefan**

**Verificat: asist. univ. Coșeru Catalin**

**Chișinău – 2024**

# Scopul lucrării:

# 1. Analiza empirică a algoritmilor.

# 2. Analiza teoretică a algoritmilor.

# 3. Determinarea complexității temporale și asimptotice a algoritmilor.

**SARCINA DE BAZĂ:**

1. Efectuați analiza empirică a algoritmilor Fibonacci propuși.

2. Determinați relația ce determină complexitatea temporală pentru acești algoritmi.

3. Determinați complexitatea asimptotică a algoritmilor.

4. Faceți o concluzie asupra lucrării efectuate.

**Termeni teoretici:**

**Etapele analizei complexității :**

În analiza complexității unui algoritm se parcurg următoarele etape:

1. Se stabilește dimensiunea problemei.

2. Se identifică operația de bază.

3. Se verifică dacă numărul de execuții ale operației de bază depinde doar de dimensiunea problemei. Dacă da, se determină acest număr. Dacă nu, se analizează cazul cel mai favorabil, cazul cel mai defavorabil și (dacă este posibil) cazul mediu.

4. Se stabilește clasa de complexitate căruia îi aparține algoritmul.

**Etapele analizei empirice:**

In analiza empirică a unui algoritm se parcurg de regulă următoarele etape:

1. Se stabilește scopul analizei.

2. Se alege metrica de eficientă ce va fi utilizată (număr de execuții ale unei/unor operații sau timp de execuție a întregului algoritm sau a unei porțiuni din algoritm.

3. Se stabilesc proprietățile datelor de intrare în raport cu care se face analiza (dimensiunea datelor sau proprietăți specifice).

4. Se implementează algoritmul într-un limbaj de programare.

5. Se generează mai multe seturi de date de intrare.

6. Se execută programul pentru fiecare set de date de intrare.

7. Se analizează datele obținute.

# Input:

# Ca input fiecare algoritm va poziția elementului din șirul lui Fibonacci, în (fig. 1) algoritmii vor căuta elementul al șaselea din șirul lui Fibonacci, unde variabila “n” de tip integer este parametru formal ce indică poziția elementului în șir pentru următorii algoritmi.



Fig.1 Input inițial.

**Analiza:**

**Algoritmul I:**

Metoda recursivă, considerată și cea mai ineficientă metodă, urmează o abordare simplă de a calcula al n-lea termen prin calcularea predecesorilor săi mai întâi, și apoi

adunându-i. Cu toate acestea, metoda face acest lucru apelându-se pe sine de mai multe ori și repetând aceeași operație, pentru același termen, cel puțin de două ori, ocupând memorie suplimentară și, teoretic, dublând timpul de execuție (fig. 3).

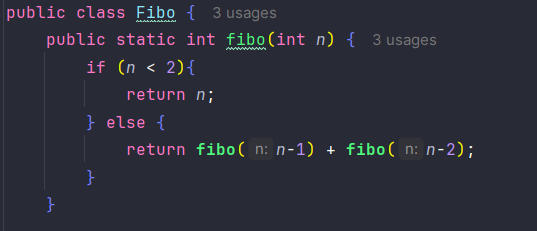


Fig. 2. Algoritmul Recursiv pentru Fibonacci (fibo).

**Output:**

Ca output se va afișa elementul calculat la fel si timpul de execuție pentru determinarea lui (fig. 2).



Fig. 3. Exemplu de output după execuția programului a algoritmului recursiv.

După execuția programului elementele mai mici, au fost găsite instant dar după întâmpină greutate, cea ce e reprezentat in tabelul 1:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Tabelul timpului de execuție fib0** | | | | | | | |
| Elementul | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
| Timpul | 0,000391600 | 0,000657 | 0,0004114 | 0,0007976 | 0,0016776 | 0,0089436 | 0,063937 |

**Tab. 1. Tabelul timpului de execuție pentru Fibonacci 1**

Structura recursivă este similară cu un **arbore binar**, unde fiecare nod are două sub noduri (două apeluri recursive), iar înălțimea acestui arbore este **n**. Acest lucru înseamnă că numărul total de noduri (adică numărul total de apeluri) este exponențial în funcție de **n**.

Numărul total de noduri într-un arbore binar complet de înălțime **n** este aproximativ 2n . Fiecare nod (apel recursiv) face o operație constantă (adică verificarea condiției **if** și adunarea), astfel că timpul total de execuție va fi proporțional cu numărul total de noduri:

**T(n) = 2n**

**Algoritmul II:**

Acest algoritm folosește o metodă **iterativă** pentru a calcula numărul Fibonacci. Mult mai eficient ca algoritmul recursiv, calculează secvența Fibonacci într-un singur ciclu for. Folosește două variabile, first\_number și second\_number, pentru a păstra ultimele două valori Fibonacci, și le actualizează la fiecare iterație (fig. 4) . Dar nu cel mai eficient pentru numere foarte mari

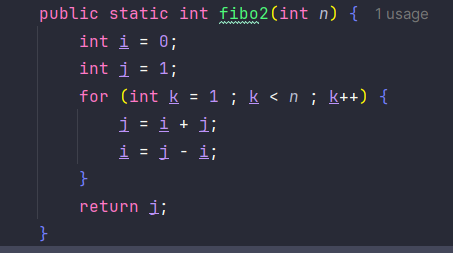


Fig. 4**. Algoritmul Iterativ pentru Fibonacci(fibo2).**

**Output:**

****

Fig. 5. Exemplu de output după execuția programului a algoritmului recursiv.

După execuția programului elementele mai mici și cele mari au fost găsite instant cea ce e reprezentat in tabelul 2:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Tabelul timpului de execuție fibo2** | | | | | | | |
| Elementul | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
| Timpul | 0,000657 | 0,0006588 | 0,0004467 | 0,0006648 | 0,0006714 | 0,0006387 | 0,0004648 |

**Tab. 2. Tabelul timpului de execuție pentru Fibonacci 2(fibo2)**

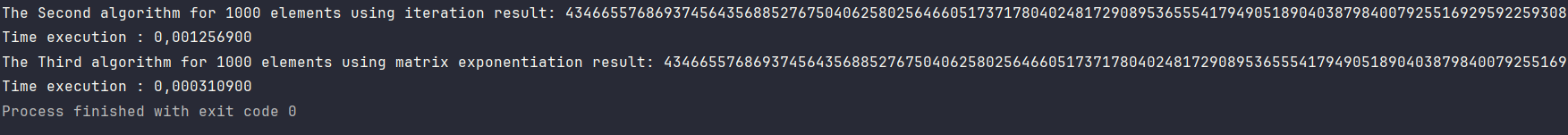
Acest algoritm conține o buclă for care se execută de **exact n ori**. La fiecare iterație, sunt efectuate câteva operații constante (**două adunări și două scăderi**). Numărul de operații efectuate într-o iterație este constant, indiferent de valoarea lui **n**, și putem nota timpul pentru fiecare iterație ca fiind **O(1)**.

Timpul total pentru **fibo2** este proporțional cu numărul de iterații ale buclei, care este exact **n**. Prin urmare, complexitatea este:

**T(n) = O(1) \* O(n) = O(n)**

**Algoritmul III:**

Acest algoritm calculează numărul Fibonacci folosind o metodă **matematică** avansată, cunoscută sub numele de **exponențiere rapidă a matricelor (fig. 6)** . Ideea principală este de a folosi proprietățile matricelor și a exponențialilor pentru a calcula valorile Fibonacci într-un mod mult mai eficient. Metoda data e cea mai eficienta dintre toate, de acest motiv am schimbat termenul cel mai mare să fie 1000 pentru a demonstra diferența dintre algoritmul dat cu cel iterativ in determinarea termenilor mult mai mari **(fig. 5)**.

**Output:** 

**Fig. 6. Rezultatul executării programului in cauza de a găsi elementul 1000.**

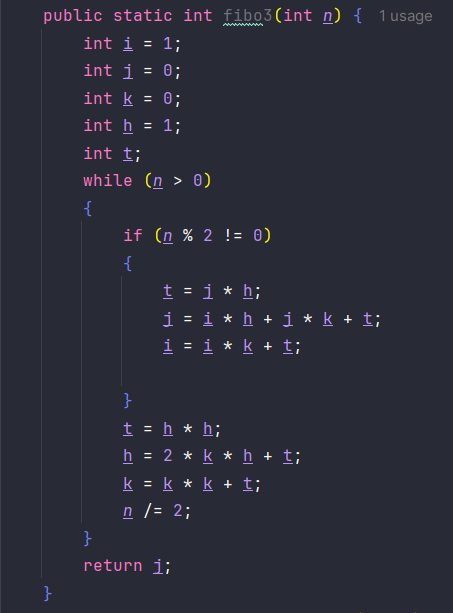


Fig. 6**. Algoritmul Optimizat pentru Fibonacci(fibo3).**

După execuția programului elementele mai mici și cele mari au fost găsite chiar mai rapid decât fib2 cea ce e reprezentat in tabelul (Tab.3):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Tabelul timpului de execuție fibo3** | | | | | | | |
| Elementul | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
| Timpul | 0,0000402 | 0,0000797 | 0,0000489 | 0,0000746 | 0,0000867 | 0,0000951 | 0,0000716 |

Tab.3. **Tabelul timpului de execuție pentru Fibonacci 3**

Acest algoritm reduce dimensiunea problemei prin împărțirea repetată a lui **number** la 2 (prin operația **number //= 2**). Prin urmare, numărul de pași necesari pentru a reduce **number** la 0 este logaritmic în funcție de n, adică **O(log n)**

La fiecare pas, algoritmul efectuează un număr constant de operații, inclusiv verificarea condiției **if**, înmulțiri, adunări și actualizări ale variabilelor. Fiecare pas durează un timp constant **O(1).**

Timpul total de execuție este proporțional cu numărul de pași (care este **O(log n)**) înmulțit cu timpul constant pe pas **O(1)**, astfel complexitatea este:

**T(n) = O(1) \* O(log n) = O(log n)**

# Concluzie

Cei trei algoritmi Fibonacci prezintă o diferență notabilă în eficiența lor temporală. Algoritmul recursiv fibo1, având o complexitate exponențială O(2^n), devine ineficient pentru valori mari ale lui n din cauza apelurilor recursive multiple și redundante. Algoritmul iterativ fibo2 aduce o îmbunătățire considerabilă prin complexitatea liniară O(n), permițând calculul într-o singură buclă, ceea ce îl face mult mai eficient pentru n mari. Totuși, cea mai performantă soluție este algoritmul cu exponențiere rapidă fibo3, având o complexitate logaritmică O(log n), care rezolvă problema eficient prin împărțirea continuă a problemei.

Astfel, fibo3 este preferat pentru calcule care implică valori foarte mari ale lui n, oferind cea mai bună performanță, în timp ce fibo1, cu complexitatea sa exponențială, devine impracticabil pe măsură ce n crește. În concluzie, alegerea algoritmului potrivit depinde de dimensiunea problemei, iar optimizarea timpului de execuție devine crucială în aplicațiile care necesită un număr mare de calcule Fibonacci.