**ALGORITHM ANALYSIS**

**Objective**: Study and analyze different algorithms for determining Fibonacci n-th term.

**Tasks:**

1. Implement at least 3 algorithms for determining Fibonacci n-th term;

2. Decide properties of input format that will be used for algorithm analysis;

3. Decide the comparison metric for the algorithms;

4. Analyze empirically the algorithms;

5. Present the results of the obtained data;

6. Deduce conclusions of the laboratory.

**1. Recursive Fibonacci**

def fibonacci\_recursive(n):

if n <= 0:

return 0

elif n == 1:

return 1

return fibonacci\_recursive(n - 1) + fibonacci\_recursive(n - 2)

* **Time Complexity:** O(2n) (Exponential)
* **Space Complexity:** O(n) (Due to recursion depth)
* **Best for:** **Very small values** of n (e.g., n≤20n )
* **Worst for:** Large values of n, as it grows exponentially and quickly becomes impractical.

🔹 **Why?** It repeatedly recalculates the same Fibonacci numbers, leading to massive redundant computations.

**2. Memoization (Top-Down Dynamic Programming)**

def fibonacci\_memoization(n, memo={}):

if n in memo:

return memo[n]

if n <= 0:

return 0

elif n == 1:

return 1

memo[n] = fibonacci\_memoization(n - 1, memo) + fibonacci\_memoization(n - 2, memo)

return memo[n]

* **Time Complexity:** O(n) (Linear)
* **Space Complexity:** O(n) (For storing computed values in a dictionary)
* **Best for:** **Moderate to large values** of n (up to 105 depending on memory)
* **Worst for:** **Very large values** where space becomes an issue.

🔹 **Why?** It stores previously computed Fibonacci values in a dictionary, avoiding redundant recursive calls.

**3. Tabulation (Bottom-Up Dynamic Programming)**

def fibonacci\_tabulation(n):

if n <= 0:

return 0

elif n == 1:

return 1

dp = [0] \* (n + 1)

dp[1] = 1

for i in range(2, n + 1):

dp[i] = dp[i - 1] + dp[i - 2]

return dp[n]

* **Time Complexity:** O(n) (Linear)
* **Space Complexity:** O(n) (Array storage)
* **Best for:** **Moderate values** of n (up to 106, limited by memory)
* **Worst for:** **Very large values** where memory allocation becomes an issue.

🔹 **Why?** It fills an array iteratively, avoiding recursion overhead.

**4. Iterative Fibonacci**

def fibonacci\_iterative(n):

if n <= 0:

return 0

elif n == 1:

return 1

a, b = 0, 1

for \_ in range(2, n + 1):

a, b = b, a + b

return b

* **Time Complexity:** O(n) (Linear)
* **Space Complexity:** O(1) (Constant, only two variables used)
* **Best for:** **Large values** of n (handles up to 107 efficiently)
* **Worst for:** Small values (but still runs fast).

🔹 **Why?** It only keeps track of the last two Fibonacci numbers, making it the most memory-efficient.

**5. Matrix Exponentiation**

def fibonacci\_matrix(n):

def multiply\_matrices(F, M):

x = F[0][0] \* M[0][0] + F[0][1] \* M[1][0]

y = F[0][0] \* M[0][1] + F[0][1] \* M[1][1]

z = F[1][0] \* M[0][0] + F[1][1] \* M[1][0]

w = F[1][0] \* M[0][1] + F[1][1] \* M[1][1]

F[0][0], F[0][1], F[1][0], F[1][1] = x, y, z, w

def power(F, n):

if n <= 1:

return

M = [[1, 1], [1, 0]]

power(F, n // 2)

multiply\_matrices(F, F)

if n % 2 != 0:

multiply\_matrices(F, M)

if n <= 0:

return 0

F = [[1, 1], [1, 0]]

power(F, n - 1)

return F[0][0]

* **Time Complexity:** O(log n) (Logarithmic)
* **Space Complexity:** O(1) (Constant)
* **Best for:** **Very large values** of nnn (up to 1018)
* **Worst for:** Small values where simpler methods are faster.

🔹 **Why?** Uses matrix exponentiation to reduce computations to logarithmic time.

**6. Binet’s Formula (Closed-Form)**

def fibonacci\_binet(n):

getcontext().prec = 100 # Increase precision

sqrt5 = Decimal(5).sqrt()

phi = (Decimal(1) + sqrt5) / Decimal(2)

return round((phi\*\*n - (-1/phi)\*\*n) / sqrt5)

* **Time Complexity:** O(1) (Constant)
* **Space Complexity:** O(1) (Constant)
* **Best for:** **Medium-sized values** (up to 105)
* **Worst for:** **Very large values** where floating-point precision errors occur.

🔹 **Why?** It calculates Fibonacci numbers instantly but loses precision for extremely large nnn.

**Summary Table**

| **Method** | **Time Complexity** | **Space Complexity** | **Best for n** | **Worst for n** |
| --- | --- | --- | --- | --- |
| **Recursive** | O(2n) | O(n) | n≤20 | n>30n (too slow) |
| **Memoization** | O(n) | O(n) | n≤105 | n>106 (memory limit) |
| **Tabulation** | O(n) | O(n) | n≤106 | n>107 (memory limit) |
| **Iterative** | O(n) | O(1) | n≤107 | Very large n (too slow) |
| **Matrix Exponentiation** | O(log n) | O(1) | n≤1018 | - |
| **Binet’s Formula** | O(1) | O(1) | n≤105 | Very large n (precision issues) |

**Which One Should You Use?**

* **If n is very small (n≤20)** → Use **Recursive**
* **If n is moderate (n≤105)** → Use **Memoization or Tabulation**
* **If n is large (n≤107)** → Use **Iterative**
* **If n is very large (n≤1018)** → Use **Matrix Exponentiation**
* **If you need instant calculation for small n** → Use **Binet’s Formula (but beware of precision loss)**



