**Ministerul Educaţiei și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

Laboratory work 1:

Study and Empirical Analysis of Algorithms for

Determining

Fibonacci N-th Term

Elaborated:

st. gr. FAF-231Vornicescu Ion

Verified:

asist. univ. Fiștic Cristofor

Chişinău - 2025

**TABLE OF CONTENTS**

[**ALGORITHM ANALYSIS** **3**](#_Toc11643)

[Objective 3](#_Toc11644)

[Tasks 3](#_Toc11645)

[Theoretical Notes: 3](#_Toc11646)

[Introduction: 4](#_Toc11647)

[Comparison Metric: 4](#_Toc11648)

[Input Format: 4](#_Toc11649)

[**IMPLEMENTATION** **5**](#_Toc11650)

[Recursive Method: 5](#_Toc11651)

[Dynamic Programming Method: 6](#_Toc11652)

[Matrix Power Method: 8](#_Toc11653)

[Binet Formula Method: 11](#_Toc11654)

[**CONCLUSION** **13**](#_Toc11655)

# ALGORITHM ANALYSIS

## Objective

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

## Tasks:

1. Implement 6 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.

## Theoretical Notes:

Empirical analysis of algorithm complexity is an alternative to mathematical analysis and can be useful in various scenarios.

It helps estimate an algorithm’s complexity class, compare the efficiency of different algorithms solving the same problem, evaluate multiple implementations of the same algorithm, and assess how well an algorithm performs on a specific computer.

This type of analysis follows a structured process. First, the objective of the analysis is defined. Next, an efficiency metric is selected, which can be the number of operations performed or the execution time of the algorithm or its components. The properties of the input data are then determined, considering aspects like data size or specific characteristics that may influence performance. After this, the algorithm is implemented in a programming language, and multiple sets of input data are generated. The program is then executed for each dataset, and the results are recorded. Finally, the collected data is analyzed, either by calculating statistical values like averages and standard deviations or by visualizing the results through graphs that relate problem size to the chosen efficiency measure.

The choice of efficiency metric depends on the goal of the analysis. If the purpose is to estimate the complexity class or verify a theoretical prediction, counting the number of operations is more appropriate. However, if the focus is on evaluating how an implementation behaves in a real-world scenario, measuring execution time is more relevant. Once the program has been executed with test data, the results are processed to extract meaningful insights, helping to understand the performance and efficiency of the algorithm.

**Introduction:**

The Fibonacci sequence is a series of numbers where each term is the sum of the two preceding ones, typically starting with 0 and 1. Mathematically, it is defined as:F(n) = F(n-1) + F(n-2), with initial values F(0) = 0 and F(1) = 1.

This sequence was introduced to the Western world by the Italian mathematician Leonardo of Pisa, known as Fibonacci, in his 1202 book *Liber Abaci*. However, similar sequences appeared in Indian mathematics centuries earlier. Fibonacci numbers have numerous applications in mathematics and beyond. They appear in nature, such as in the arrangement of leaves, the branching of trees, the spirals of shells, and the patterns of sunflower seeds.

The sequence is also closely related to the golden ratio (approximately 1.618), as the ratio of consecutive Fibonacci numbers approaches this value as the sequence progresses. In computer science, Fibonacci numbers are used in algorithms, data structures, and mathematical analysis. Fibonacci search techniques and Fibonacci heaps are examples of their applications. In financial markets, analysts use Fibonacci retracement levels to predict stock price movements. The sequence also has connections to art,

architecture, and music. Many classical compositions and architectural designs incorporate proportions related to the Fibonacci sequence and the golden ratio to create aesthetically pleasing structures.

Overall, Fibonacci numbers demonstrate a fascinating blend of mathematics, nature, and human creativity, making them one of the most intriguing sequences in number theory.

## Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n))

## Input Format:

As input, each algorithm will receive two series of numbers representing the order of Fibonacci terms to be computed.

The first series has a more limited scope: (5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49)  
This selection ensures that the recursive method, which has exponential time complexity O(2^n), remains feasible. Since the recursive approach without optimization quickly becomes impractical, we limit testing to relatively small values, allowing us to analyze its performance without excessive computation time.

The second series has a broader scope to compare more optimized algorithms, such as memoization and iterative approaches:(5, 10, 15, 20, 25, 30, 35, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 600, 700, 1000, 2000, 3000, 4000, 5000, 7500, 10000, 12500, 15000, 17500, 20000, 30000)  
 This selection allows us to observe the efficiency of different approaches across small, moderate,and large values. Memoization O(n) and matrix exponentiation/logarithmic approaches O(log(n)) can handle much larger numbers, making it essential to test their behavior for increasingly large inputs.

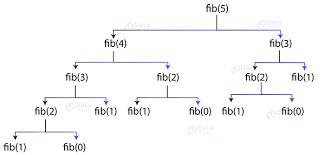
By structuring the test values this way, we can effectively compare the performance of naive recursion, dynamic programming, and optimized Fibonacci algorithms.

# IMPLEMENTATION

All six Fibonacci algorithms will be implemented in their basic form in Python and analyzed empirically based on execution time. The recursive approach has exponential complexity O(2n), making it infeasible for large inputs, while memoization, iteration, and dynamic programming have linear complexity O(n), offering significant improvements. The matrix exponentiation method runs in logarithmic time O(log(n)), and Binet’s formula achieves constant time O(1), though it is limited by numerical precision.By comparing these algorithms across different input sizes, we can evaluate their real-world performance and determine the most effective approach depending on computational constraints.

## Recursive Method:

The naive recursive Fibonacci function follows the direct mathematical definition, computing each term by recursively summing the two previous ones. However, this approach leads to exponential timecomplexity, making it highly inefficient for large values due to redundant calculations. It is only feasible for



*Figure 1 Fibonacci recursion*

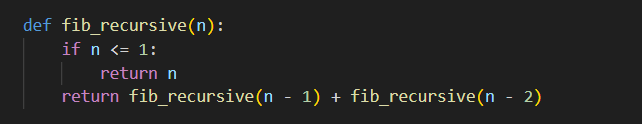
*Algorithm Description:*

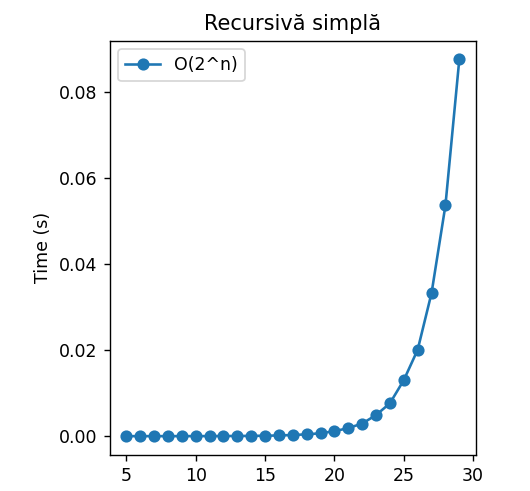
The naive recursive Fibonacci method follows the algorithm as shown in the next pseudocode:

|  |
| --- |
| FUNCTION fib\_recursive(n):  IF n <= 1:  RETURN n  RETURN fib\_recursive(n - 1) + fib\_recursive(n - 2) |

Directly follows the Fibonacci definition but recomputes values redundantly.

*Implementation:*

*Figure 2 Fibonacci recursion in Python*



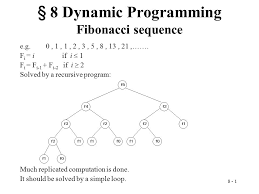
*Figure 3 Graph of Recursive Fibonacci Function*

The y-axis represents execution time in seconds, while the x-axis represents Fibonacci input values ranging from 5 to 30. The data points (blue circles) show measured times, and the solid line represents the trend of exponential growth, labeled O(2ⁿ), which is the time complexity of this approach.

As expected, the execution time remains very low for small values of n, but it increases rapidly as n grows, illustrating the inefficiency of the naive recursive approach.

## Dynamic Programming Method:

The dynamic programming method stores all previously computed Fibonacci numbers in an array to build up the sequence iteratively. It has the same linear time complexity as the iterative method but requires additional memory proportional to n. This makes it useful when multiple Fibonacci numbers need to be accessed quickly.



*Figure 4 Fibonacci DP algoritm*

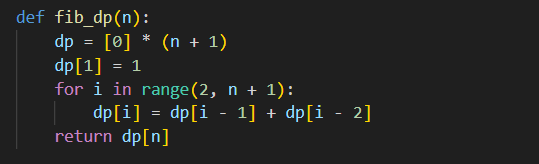
*Algorithm Description:*

The naive DP algorithm for Fibonacci n-th term follows the pseudocode:

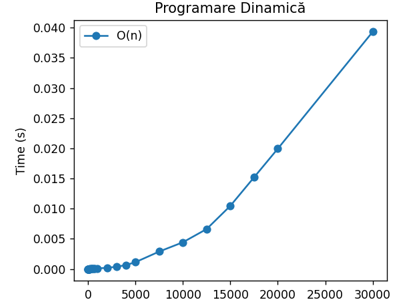
|  |
| --- |
| FUNCTION fib\_dp(n):  IF n <= 1:  RETURN n  dp ← ARRAY of size (n + 1)  dp[0] ← 0  dp[1] ← 1  FOR i FROM 2 TO n:  dp[i] ← dp[i - 1] + dp[i - 2]  RETURN dp[n] |

This method Stores all previous values in an array for easy access but uses more memory.

*Implementation:*



*Figure 5 Fibonacci DP in Python*



*Figure 6 Fibonacci DP*

This graph displays how execution time (y-axis) increases as the input size n grows (x-axis), demonstrating a clear non-linear growth pattern that becomes steeper after n=10,000. This implementation likely has O(n) time complexity, which is significantly more efficient than the naive recursive approach (which would be O(2n)), as dynamic programming eliminates redundant calculations by storing previously computed values.

## Matrix Power Method:

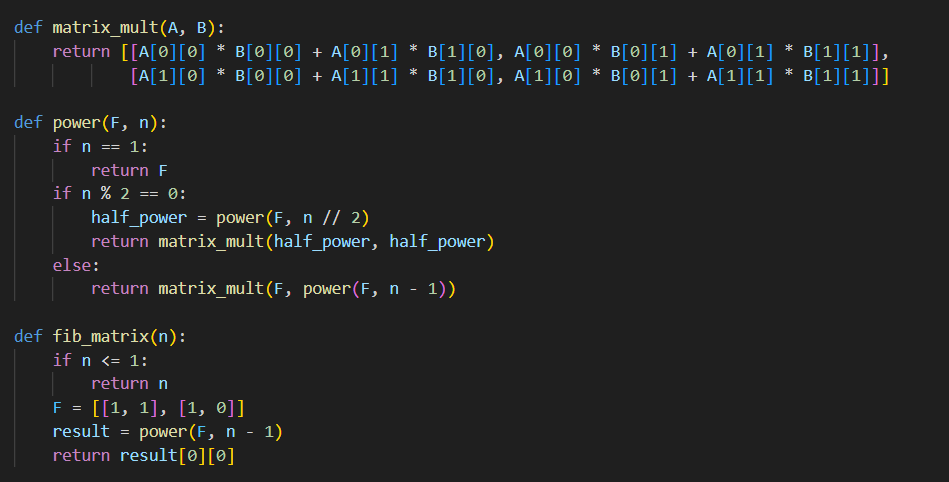
This method uses **matrix exponentiation** to compute Fibonacci numbers in **logarithmic time**. By raising a transformation matrix to the power of n−1n-1n−1, the Fibonacci number can be derived using fast exponentiation techniques. This method is highly efficient for large inputs but requires matrix multiplication operations.

This set of operation can be described in pseudocode as follows:

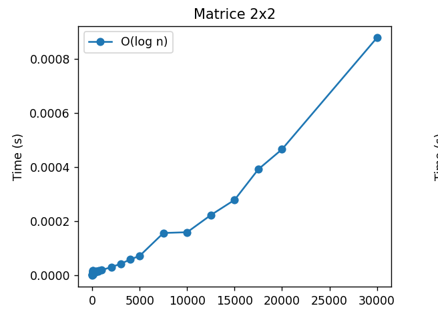
|  |
| --- |
| FUNCTION matrix\_mult(A, B):  RETURN [[A[0][0] \* B[0][0] + A[0][1] \* B[1][0], A[0][0] \* B[0][1] + A[0][1] \* B[1][1]],  [A[1][0] \* B[0][0] + A[1][1] \* B[1][0], A[1][0] \* B[0][1] + A[1][1] \* B[1][1]]]  FUNCTION power(F, n):  IF n == 1:  RETURN F  IF n MOD 2 == 0:  half\_power ← power(F, n / 2)  RETURN matrix\_mult(half\_power, half\_power)  ELSE:  RETURN matrix\_mult(F, power(F, n - 1))  FUNCTION fib\_matrix(n):  IF n <= 1:  RETURN n  F ← [[1, 1], [1, 0]]  result ← power(F, n - 1)  RETURN result[0][0] |

Uses fast exponentiation to compute Fibonacci numbers efficiently.

*Implementation:*



*Figure 7 Fibonacci Matrix Power Method in Python*



*Figure 8 Matrix Method Fibonacci graph*

This graph shows the time complexity of a matrix exponentiation method for calculating Fibonacci numbers, labeled as O(log n). It illustrates how execution time (y-axis) increases with input size n (x-axis), demonstrating a much more efficient algorithm than standard dynamic programming. This matrix method achieves logarithmic time complexity by using the property that Fibonacci numbers can be computed through 2×2 matrix exponentiation, making it incredibly efficient for calculating large Fibonacci values as it requires significantly fewer operations than linear approaches.

## Binet Formula Method:

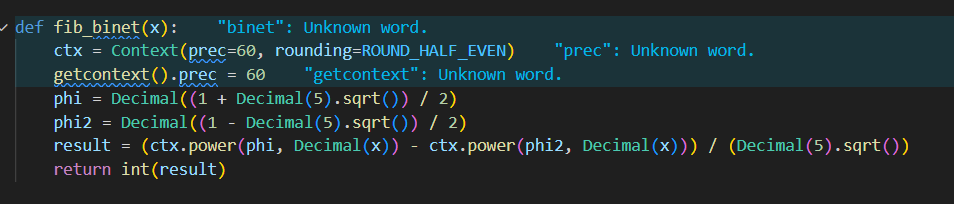
Binet’s formula uses the golden ratio (ϕ) and its conjugate to compute Fibonacci numbers directly using a closed-form expression. While it has constant time complexity, its accuracy is limited by floating-point precision, making it unreliable for very large values without arbitrary-precision arithmetic.

*Algorithm Description:*

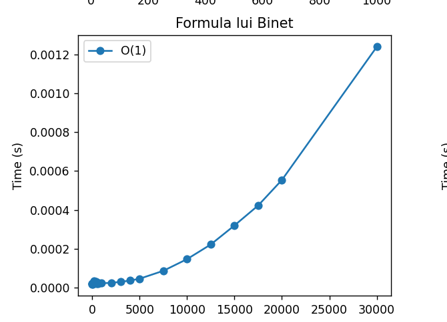
The set of operation for the Binet Formula Method can be described in pseudocode as follows:

|  |
| --- |
| FUNCTION fib\_binet(n):  phi ← (1 + sqrt(5)) / 2  psi ← (1 - sqrt(5)) / 2  result ← (phi^n - psi^n) / sqrt(5)  RETURN round(result) |

*Implementation:*



*Figure 9 Binet Method Fibonacci*

**

*Figure 10 Binet Method Fibonacci graph*

The Binet Formula becomes unreliable for calculating Fibonacci numbers beyond the 80th term when implemented in standard Python. This limitation stems from floating-point precision issues that accumulate when computing larger terms. While theoretically elegant, its practical application requires either specialized numerical libraries or alternative programming languages with extended precision capabilities to maintain accuracy for larger Fibonacci values.

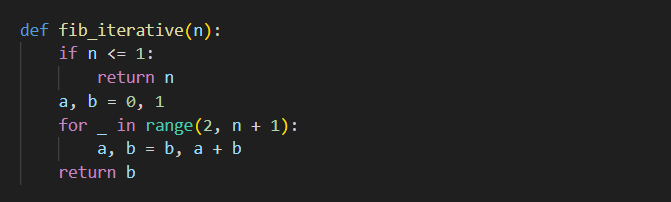
**Iterative Approach**

The iterative method computes Fibonacci numbers using a simple loop, maintaining only the last two values at each step. This approach is efficient in both time and space, as it runs in linear time while using constant extra memory. It is one of the most practical methods for moderate Fibonacci values.

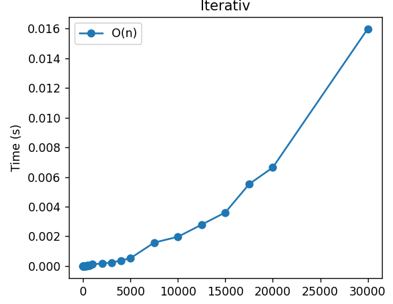
*Algorithm pseudecode:*

|  |
| --- |
| FUNCTION fib\_iterative(n):  IF n <= 1:  RETURN n  a ← 0  b ← 1  FOR i FROM 2 TO n:  temp ← a + b  a ← b  b ← temp  RETURN b |

*Implementation:*



*Figure 11 iterative Method Fibonacci*



*Figure 12 iterative Method Fibonacci graph*

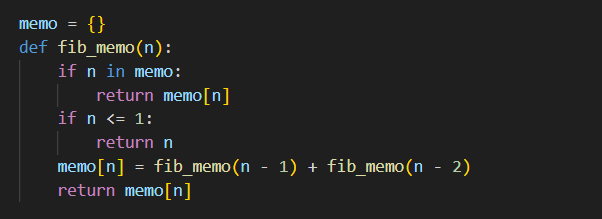
The graph displays the execution time of an iterative approach for calculating Fibonacci numbers, with O(n) time complexity. It shows a linear relationship between input size (x-axis) and computation time (y-axis), with execution time steadily increasing as n grows toward 30,000. This iterative implementation likely uses a simple loop to calculate each Fibonacci number sequentially, offering the same linear time complexity as dynamic programming but with potentially lower memory overhead since it doesn't require storing all previous values in a lookup table.

**Memoized Recursive Approach**

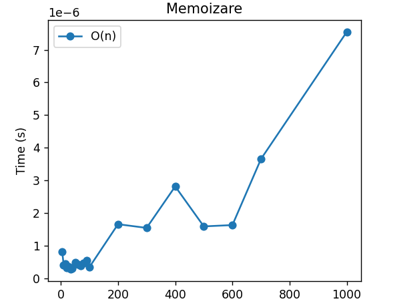
Memoization improves recursion by storing previously computed Fibonacci numbers in a dictionary to avoid redundant calculations. This reduces the time complexity from exponential to linear, making it significantly faster while maintaining a recursive structure. However, it still requires additional memory to store intermediate results.

*Algorithm pseudecode:*

|  |
| --- |
| FUNCTION fib\_memo(n, memo):  IF n IN memo:  RETURN memo[n]  IF n <= 1:  RETURN n  memo[n] = fib\_memo(n - 1, memo) + fib\_memo(n - 2, memo)  RETURN memo[n]  # Wrapper function to initialize memoization dictionary  FUNCTION fibonacci(n):  RETURN fib\_memo(n, {}) |



*Figure 13 meoisation Method Fibonacci*

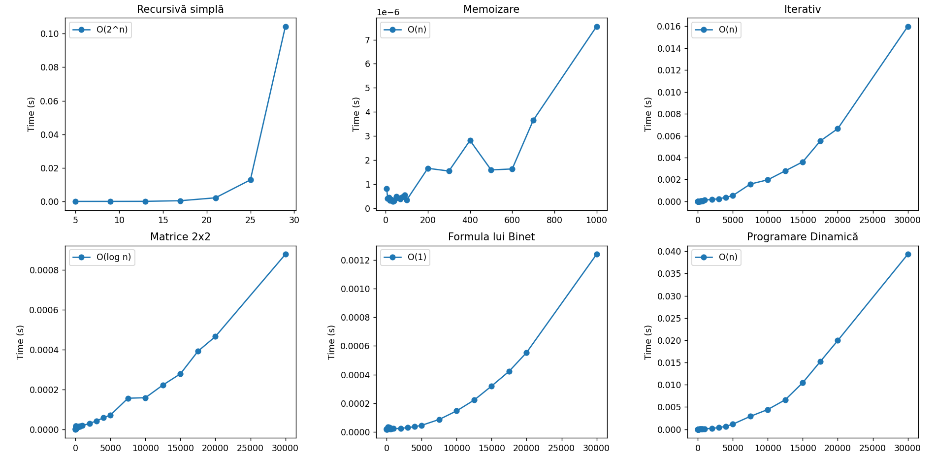


*Figure 14 meoisation Method Fibonacci*

The graph shows an O(n) memoization implementation for calculating Fibonacci numbers, with execution times measured in microseconds (10-6 seconds). Despite some fluctuations in the middle range, it demonstrates linear growth overall, becoming more pronounced as n approaches 1000. Unlike the previous graphs, this implementation operates on a much smaller input range (0-1000) and achieves dramatically faster execution times, suggesting highly optimized caching of previously calculated values to avoid redundant computation.

# CONCLUSION

This laboratory work demonstrates a comprehensive comparison of multiple approaches to computing Fibonacci numbers, revealing significant differences in performance characteristics and practical applications. Performance measurements across different input scales confirm the theoretical complexity classes while revealing the practical thresholds where each algorithm becomes preferable. For small values (n<20), differences are negligible. For moderate values (20<n<1000), linear algorithms provide a good balance of simplicity and performance. For very large values (n>1000), the logarithmic approach of matrix exponentiation becomes the clear winner. This empirical analysis demonstrates how algorithm selection must consider not only asymptotic complexity but also practical factors such as implementation language limitations, memory constraints, and specific input ranges for the target application.



*Figure 15 Fibonacci Algoritms*