

**Ministerul Educației 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-211

Cunev Anastasia

Verified:

asist. univ.

Fiștic Cristofor

Chișinău - 2023

## TABLE OF CONTENTS

<b>ALGORITHM ANALYSIS.....</b>	<b>3</b>
Objective .....	3
Tasks.....	3
Theoretical Notes: .....	3
Introduction: .....	4
Comparison Metric:.....	4
Input Format: .....	4
<b>IMPLEMENTATION.....</b>	<b>5</b>
Recursive Method: .....	5
Iterative Method: .....	7
Dynamic Programming Method:.....	9
Matrix Multiplication Method: .....	10
Binet Formula Method: .....	13
Tail Recursion Method: .....	14
<b>CONCLUSION.....</b>	<b>17</b>

# 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.

## Theoretical Notes:

An alternative to mathematical analysis of complexity is empirical analysis. This may be useful for: obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer.

In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established.
2. Choose the efficiency metric to be used (number of executions of an operation (s) or time execution of all or part of the algorithm).
3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties).
4. The algorithm is implemented in a programming language.
5. Generating multiple sets of input data.
6. Run the program for each input data set.
7. The obtained data are analyzed.

The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behavior of the implementation of an algorithm then execution time is appropriate.

After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

## **Introduction:**

The Fibonacci sequence is the series of numbers where each number is the sum of the two preceding numbers. For example: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ... Mathematically we can describe this as:  $x_n = x_{n-1} + x_{n-2}$ .

Many sources claim this sequence was first discovered or "invented" by Leonardo Fibonacci. The Italian mathematician, who was born around A.D. 1170, was initially known as Leonardo of Pisa. In the 19th century, historians came up with the nickname Fibonacci (roughly meaning "son of the Bonacci clan") to distinguish the mathematician from another famous Leonardo of Pisa.

There are others who say he did not. Keith Devlin, the author of Finding Fibonacci: The Quest to Rediscover the Forgotten Mathematical Genius Who Changed the World, says there are ancient Sanskrit texts that use the Hindu-Arabic numeral system - predating Leonardo of Pisa by centuries.

But, in 1202 Leonardo of Pisa published a mathematical text, Liber Abaci. It was a "cookbook" written for tradespeople on how to do calculations. The text laid out the Hindu-Arabic arithmetic useful for tracking profits, losses, remaining loan balances, etc, introducing the Fibonacci sequence to the Western world.

Traditionally, the sequence was determined just by adding two predecessors to obtain a new number, however, with the evolution of computer science and algorithmics, several distinct methods for determination have been uncovered. The methods can be grouped in 4 categories, Recursive Methods, Dynamic Programming Methods, Matrix Power Methods, and Benet Formula Methods. All those can be implemented naively or with a certain degree of optimization that boosts their performance during analysis.

As mentioned previously, the performance of an algorithm can be analyzed mathematically (derived through mathematical reasoning) or empirically (based on experimental observations).

Within this laboratory, we will be analyzing the 6 naïve algorithms empirically.

## **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 that will contain the order of the Fibonacci terms being looked up. The first series will have a more limited scope, (5, 7, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 37, 40, 42, 45), to accommodate the recursive method, while the second series will have a bigger scope to be able to compare the other algorithms between themselves (501, 631, 794, 1000, 1259, 1585, 1995, 2512, 3162, 3981, 5012, 6310, 7943, 10000, 12589, 15849).

## IMPLEMENTATION

All six algorithms will be implemented in python and analyzed empirically based on the time required for their completion. Comparing all these algorithms we can define which method is most suitable for different needs or which one is the most efficient.

### Recursive Method:

The recursive algorithm is a simple and straightforward method for computing the nth Fibonacci number. The algorithm is based on the calculation of two previous numbers and their addition. This algorithm works well for small numbers, but is inefficient for large numbers. The problem is that by calling itself, it calculates the same values multiple times, which leads to a lot of redundant calculations and more memory usage.

#### *Algorithm Description:*

The next pseudocode represents the Recursive algorithm for calculating the n-th Fibonacci term:

```
fibonacci_recursive(n):  
  
    if n <= 0:  
        return 0  
  
    else if n == 1:  
        return 1  
  
    otherwise:  
  
        return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
```

#### *Implementation:*

```
def fibonacci_recursive(n):  
    if n <= 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
```

Figure 1. Fibonacci recursion in Python

## Results:

After running the Recursive algorithm for every term from the list of inputs, I got the following results:

Time(s) \ n-th number	5	7	10	12	15	17	20	22	25	27	30	32	35	37	40	42
Recursive	0.222	0.225	0.222	0.22	0.224	0.219	0.227	0.232	0.254	0.295	0.531	1.009	3.548	8.957	37.498	97.675
Iterative	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Matrix multiplication	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Dynamic programming	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Binet formula	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Tail recursion	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 2. Results for first set of inputs

In Figure 3 is represented the table of results for the first set of inputs. The first row denotes the Fibonacci n-th term for which the functions were run. Starting from the second line, we get the execution time. We can observe that the Recursive algorithm is the most inefficient compared to the other ones. While the execution time for the other algorithms strives to 0, the execution time for the recursive one becomes bigger as the n-th term becomes larger.

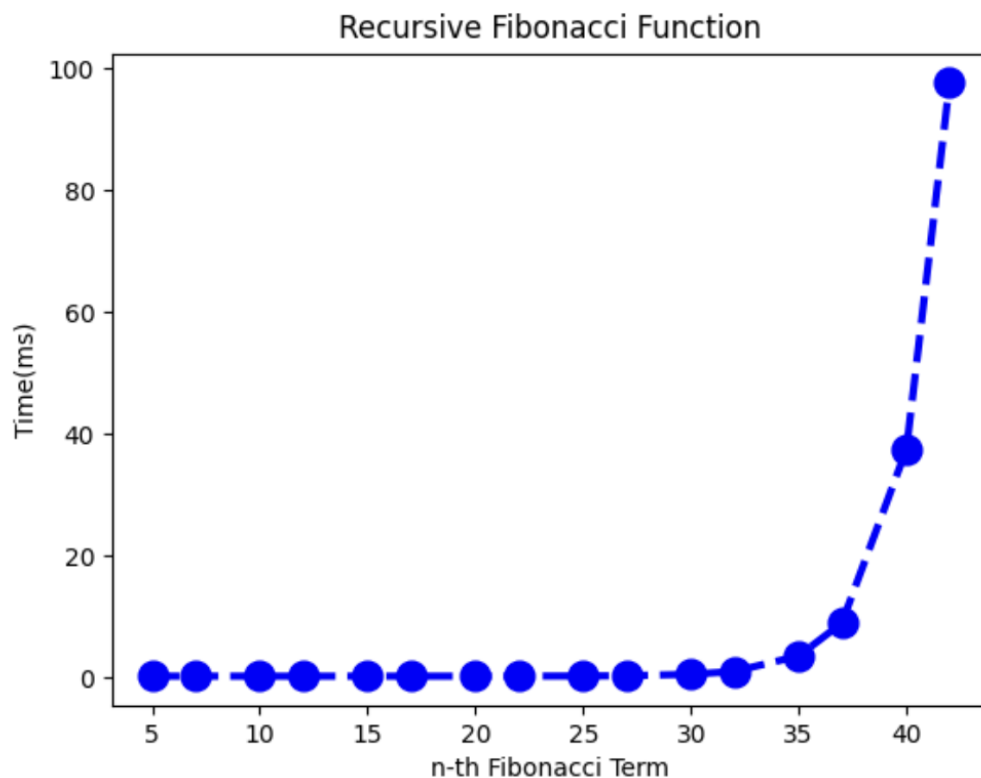


Figure 3. Graph of Recursive Fibonacci Function

On the graph in Figure 4 we can also see that after the 37-th term the function grows exponentially, which means that the Time Complexity is exponential ( $T(2^n)$ ).

### Iterative Method:

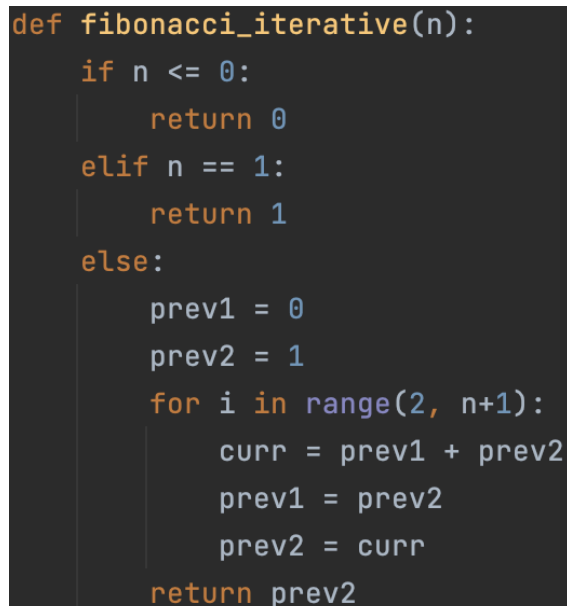
The iterative method is the algorithm to calculate the n-th Fibonacci term without use of recursion. The idea is that it computes the elements one by one in a loop and returns the n-th Fibonacci term. The iterative method is more efficient than the recursive one because it doesn't use recursion and doesn't produce redundant calculations.

#### *Algorithm Description:*

The next pseudocode represents the Iterative algorithm for calculating the n-th Fibonacci term:

```
fibonacci_iterative(n):  
    if n <= 0:  
        return 0  
    else if n == 1:  
        return 1  
    otherwise:  
        prev1 = 0  
        prev2 = 1  
        for i = 2 to n+1  
            curr = prev1 + prev2  
            prev1 = prev2  
            prev2 = curr  
        return prev2
```

#### *Implementation:*

A screenshot of a code editor with a dark background and light-colored text. The code is a Python function named 'fibonacci\_iterative' that takes an integer 'n' as input. It uses conditional statements to handle base cases (n <= 0 and n == 1) and a loop to calculate the Fibonacci sequence iteratively for n > 1. The function returns the nth Fibonacci number.

```
def fibonacci_iterative(n):  
    if n <= 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        prev1 = 0  
        prev2 = 1  
        for i in range(2, n+1):  
            curr = prev1 + prev2  
            prev1 = prev2  
            prev2 = curr  
        return prev2
```

Figure 4. Fibonacci iteration in Python

## Results:

After running the Iterative algorithm for every term from the second list of inputs, I got the following results:

Time(s) \ n-th number	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
Iterative	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001	0.0	0.001	0.001	0.002	0.003
Matrix multiplication	0.0	0.0	0.0	0.0	0.001	0.001	0.002	0.002	0.003	0.005	0.013	0.012	0.017	0.031	0.042	0.056
Dynamic programming	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001	0.001	0.002	0.003	0.004	0.007
Binet formula	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.002	0.002	0.005
Tail recursion	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001	0.0	0.001	0.0012	0.003	0.001	0.001	0.001	0.001

Figure 5. Results for the second set of inputs

In Figure 5 is represented the table of results for the second set of inputs. The first row denotes the Fibonacci n-th term for which the functions were run. Starting from the second line, we get the execution time. We can observe that the Iterative algorithm is doing pretty well compared to the other ones. Its execution time strives to 0, and starting only from the 5012-th term it starts to increase. Having the results from the table and the graph we can conclude that the time complexity for the Iterative algorithm is  $T(n)$ .

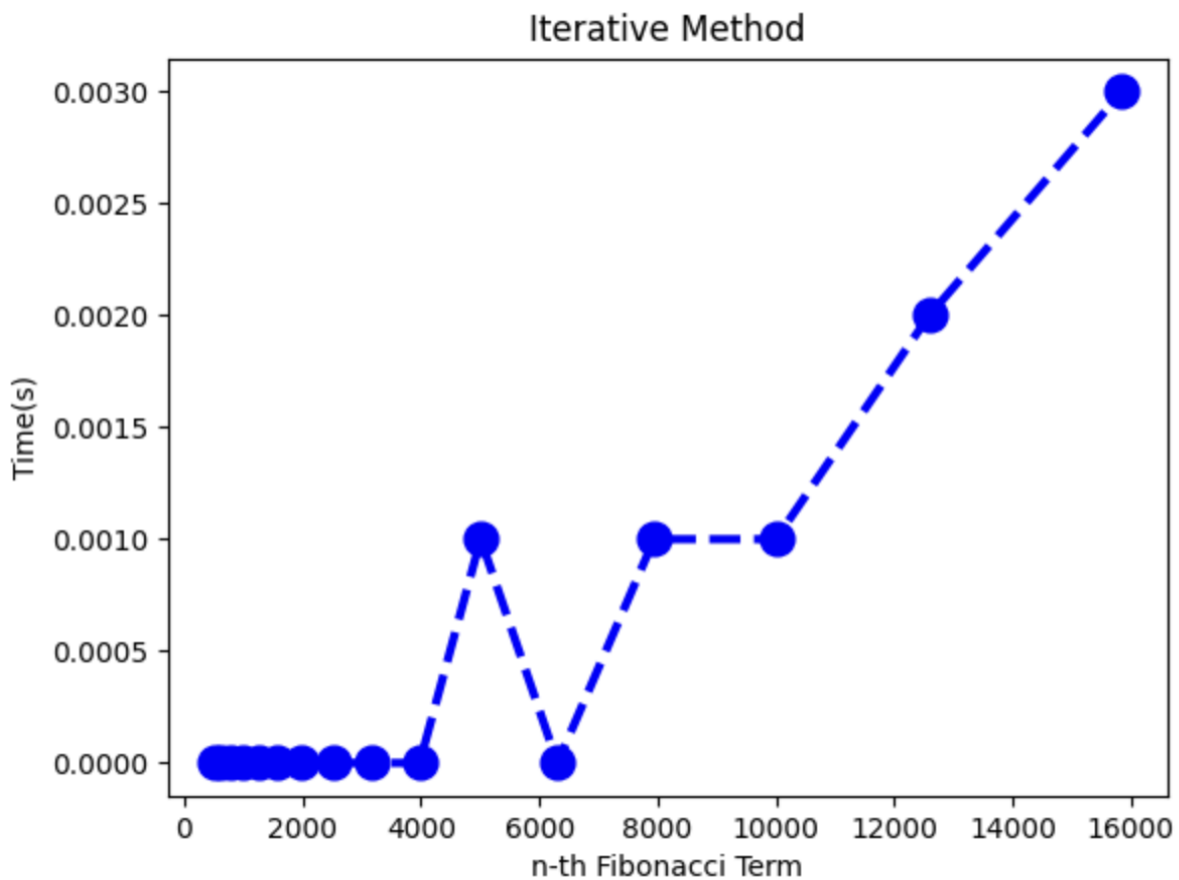


Figure 6. Graph of Iterative Fibonacci Function



## Dynamic Programming Method:

The dynamic programming algorithm is a simple and straightforward method for computing the  $n$ th Fibonacci number, but instead of calculating the elements recursively, it operates based on an array data structure that holds the previously computed terms, eliminating the need to recompute them.

### Algorithm Description:

The next pseudocode represents the Dynamic Programming algorithm for calculating the  $n$ -th Fibonacci term:

```
fibonacci_dp(n):  
    Array f;  
    f[0] = 0;  
    f[1] = 1;  
    for i = 2 to n + 1  
        f[i] <- f[i-1] + f[i-2];  
    return f[n]
```

### Implementation:

```
def fibonacci_dp(n):  
    f = [0, 1]  
  
    for i in range(2, n + 1):  
        f.append(f[i - 1] + f[i - 2])  
    return f[n]
```

Figure 7. Fibonacci dynamic programming in Python

### Results:

The results for the second list of inputs for the Dynamic Programming algorithm :

Time(s) \ n-th number	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
Iterative	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001	0.0	0.001	0.001	0.002	0.003
Matrix multiplication	0.0	0.0	0.0	0.0	0.001	0.001	0.002	0.002	0.003	0.005	0.013	0.012	0.017	0.031	0.042	0.056
Dynamic programming	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001	0.001	0.002	0.003	0.004	0.007
Binet formula	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.002	0.0002	0.0005
Tail recursion	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001	0.0	0.001	0.0012	0.003	0.001	0.001	0.001	0.001

Figure 8. Results for second set of inputs

We can observe that the Dynamic Programming algorithm is pretty efficient compared to the other ones. Its execution time strives to 0, and starting only from the 5012-th term it starts to increase. Having the results from the table and the graph we can conclude that the time complexity for the Dynamic Programming algorithm is  $T(n)$ .

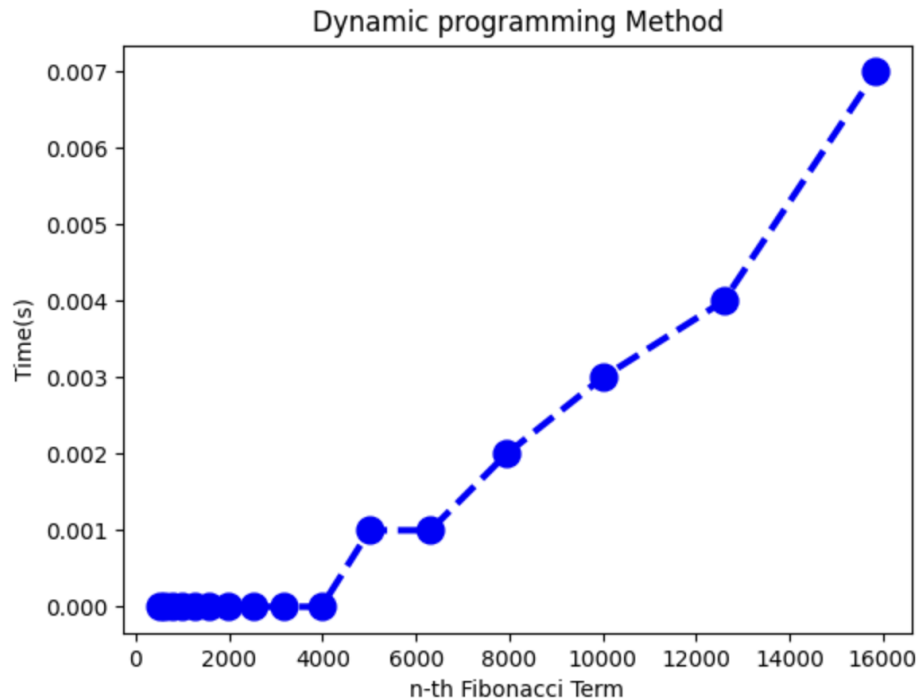


Figure 9. Graph of Dynamic Programming Fibonacci Function

### Matrix Multiplication Method:

The Matrix multiplication method is an efficient algorithm to calculate the n-th Fibonacci term. The algorithm starts by defining a 2x2 matrix

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

The matrix A can be raised to the power of n to give another 2x2 matrix, B, where the elements of B are the Fibonacci numbers:

$$A^n = \begin{bmatrix} \text{fib}(n+1) & \text{fib}(n) \\ \text{fib}(n) & \text{fib}(n-1) \end{bmatrix}$$

*Algorithm Description:*

1. Create a 2x2 matrix C with all values set to 0
2. Calculate the values of each cell in matrix C using matrix multiplication
3. Return matrix C
4. If n is less than or equal to 0, return 0

5. If  $n$  is equal to 1, return 1
6. Else, create a 2x2 matrix  $A$  with values  $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
7. Create a 2x2 matrix  $B$  with values  $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
8. For  $i$  from 2 to  $n$ , repeat the following steps:
9. Set  $B$  equal to the result of `matrix_mult(A, B)`
10. Return the value in the first cell of the first row of matrix  $B$ ,  $B[0][0]$

The next pseudocode represents the Matrix Multiplication algorithm for calculating the  $n$ -th Fibonacci term:

```

fibonacci_matrix(n):

    if n <= 0:

        return 0
    else if n == 1:
        return 1
    otherwise:
        A = [[1, 1], [1, 0]];
        B = [[1, 1], [1, 0]];
        for i = 2 to n:
            B = matrix_mult(A, B)
        return B[0][0]

```

*Implementation:*

```

def matrix_mult(A, B):
    C = [[0, 0], [0, 0]]
    C[0][0] = A[0][0] * B[0][0] + A[0][1] * B[1][0]
    C[0][1] = A[0][0] * B[0][1] + A[0][1] * B[1][1]
    C[1][0] = A[1][0] * B[0][0] + A[1][1] * B[1][0]
    C[1][1] = A[1][0] * B[0][1] + A[1][1] * B[1][1]
    return C

def fibonacci_matrix(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        A = [[1, 1], [1, 0]]
        B = [[1, 1], [1, 0]]
        for i in range(2, n):
            B = matrix_mult(A, B)
        return B[0][0]

```

Figure 10. Fibonacci matrix multiplication in Python

Results:

The results for the second list of inputs for the Matrix Multiplication algorithm :

Time(s) \ n-th number	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
Iterative	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001	0.0	0.001	0.001	0.002	0.003
Matrix multiplication	0.0	0.0	0.0	0.0	0.001	0.001	0.002	0.002	0.003	0.005	0.013	0.012	0.017	0.031	0.042	0.056
Dynamic programming	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001	0.001	0.002	0.003	0.004	0.007
Binet formula	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.002	0.0002	0.0005
Tail recursion	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001	0.0	0.001	0.0012	0.003	0.001	0.001	0.001	0.001

Figure 11. Results for second set of inputs

From the following set of results we can see that the matrix multiplication method is quite efficient and accurate, even if it's slower than the other ones. Its Time Complexity is  $T(n)$  in the worst case and  $T(\log(n))$  in the best case.

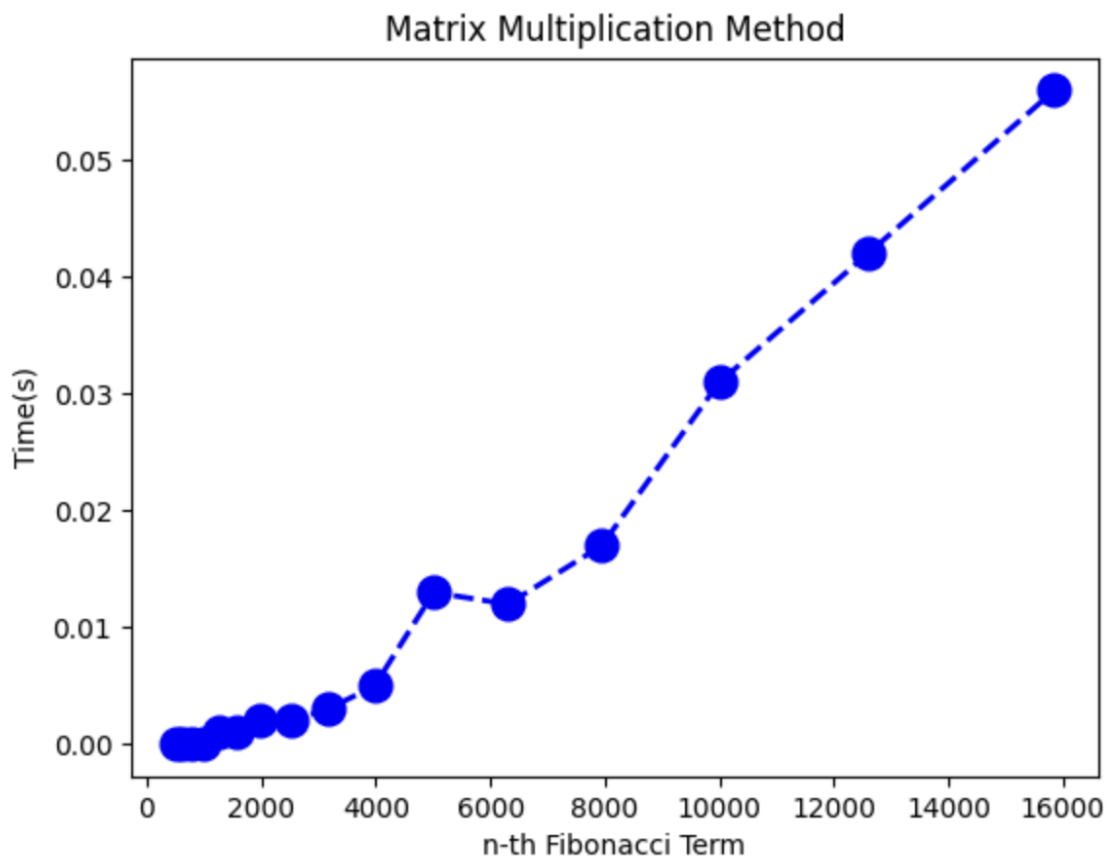


Figure 12. Graph of Matrix Multiplication Fibonacci Function

### Binet Formula Method:

Binet's Formula method is considered the most efficient algorithm as it uses the formula based on the Golden Ratio. The Binet formula expresses the nth Fibonacci number as a function of the golden ratio and its conjugate. The Binet formula is inaccurate for very large values of n, due to the fact that the golden ratio is an irrational number and its representation in floating-point arithmetic is not exact.

#### Algorithm Description:

The next pseudocode represents the Binet's formula algorithm for calculating the n-th Fibonacci term:

```
fibonacci_binet(n):  
  
    phi = (1 + sqrt(5)) / 2  
  
    psi = (1 - sqrt(5)) / 2;  
  
    return int((phi**n - psi**n) / sqrt(5))
```

#### Implementation:

```
def fibonacci_binet(n):  
    phi = (1 + math.sqrt(5)) / 2  
    psi = (1 - math.sqrt(5)) / 2  
    return int((phi**n - psi**n) / math.sqrt(5))
```

Figure 13. Fibonacci Binet formula in Python

#### Results:

After running the Binet's formula algorithm for every term from the second list of inputs, I got the following results:

Time(s) \ n-th number	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
Iterative	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001	0.0	0.001	0.001	0.002	0.003
Matrix multiplication	0.0	0.0	0.0	0.0	0.001	0.001	0.002	0.002	0.003	0.005	0.013	0.012	0.017	0.031	0.042	0.056
Dynamic programming	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001	0.001	0.002	0.003	0.004	0.007
Binet formula	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.002	0.002	0.005
Tail recursion	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001	0.0	0.001	0.0012	0.003	0.001	0.001	0.001	0.001

Figure 14. Results for second set of inputs

From the following set of results presented in the 4-th row of the table we can conclude that the Binet's formula method is the most efficient one from the proposed ones, because its execution time is equal to 2 milliseconds starting from the 10000-th term. Being the most efficient method, Binet's formula is not the most accurate one, as it gives only approximate results for large terms.

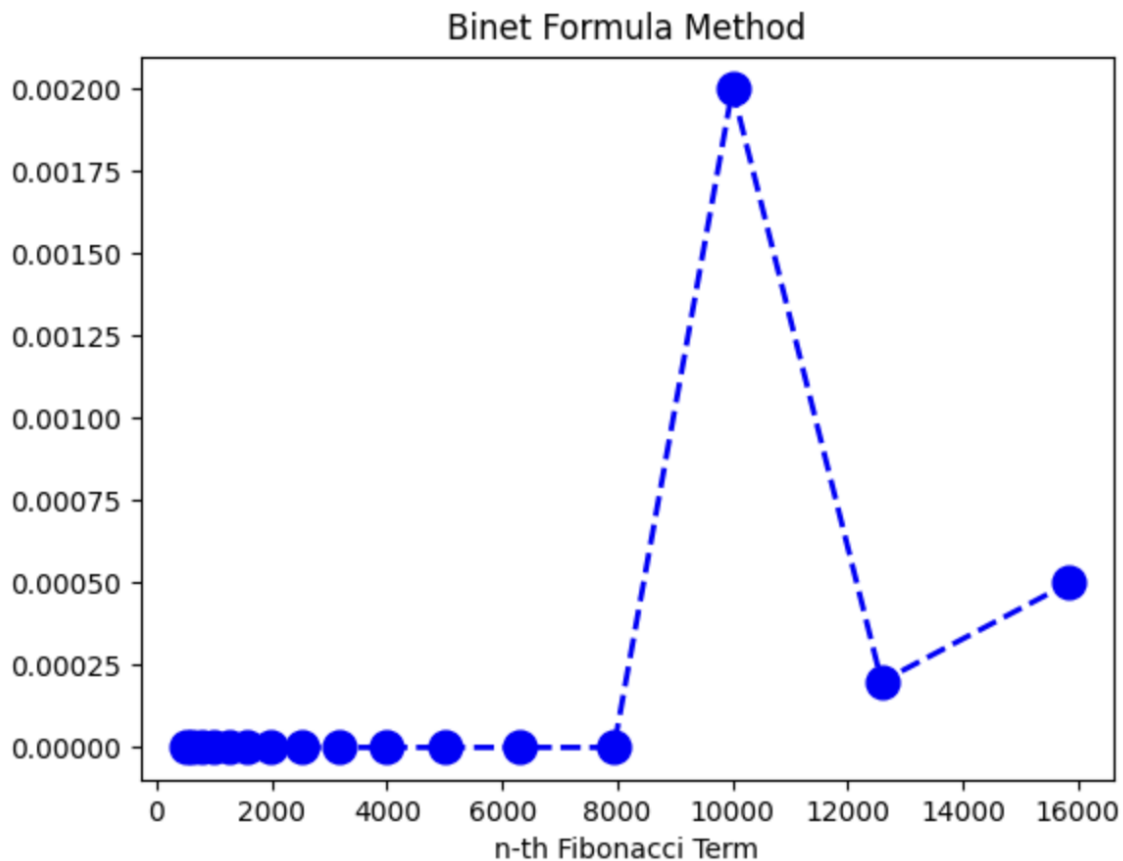


Figure 15. Graph of Binet Formula Fibonacci Function

### Tail Recursion Method:

A function is considered tail recursive when the recursive call is the final operation executed by the function. This method is the optimization of the inefficient Recursive algorithm. The difference is that it avoids recalculating the same values multiple times and performs the recursion in the last step and single step.

### Algorithm Description:

The next pseudocode represents the Tail Recursion algorithm for calculating the n-th Fibonacci term:

```
fibonacci_tail_recursion(n, a = 0, b = 1):  
    if n == 0:  
        return a  
    else if n == 1:  
        return b  
    otherwise:  
        return fibonacci_tail_recursion(n-1, b, a+b)
```

### Implementation:

```
def fibonacci_tail_recursion(n, a=0, b=1):  
    if n == 0:  
        return a  
    elif n == 1:  
        return b  
    else:  
        return fibonacci_tail_recursion(n-1, b, a+b)
```

Figure 16. Fibonacci tail recursion in Python

### Results:

After running the Tail Recursion algorithm for every term from the second list of inputs, I got the following results:

Time(s) \ n-th number	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
Iterative	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001	0.0	0.001	0.001	0.002	0.003
Matrix multiplication	0.0	0.0	0.0	0.0	0.001	0.001	0.002	0.002	0.003	0.005	0.013	0.012	0.017	0.031	0.042	0.056
Dynamic programming	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001	0.001	0.002	0.003	0.004	0.007
Binet formula	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.002	0.0002	0.0005
Tail recursion	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001	0.0	0.001	0.0012	0.003	0.001	0.001	0.001	0.001

Figure 17. Results for second set of inputs

From the following set of results presented in the 5-th row of the table we can conclude that the Tail Recursion method is one of the most efficient from the proposed ones. The Time Complexity of this algorithm is linear ( $T(n)$ ).

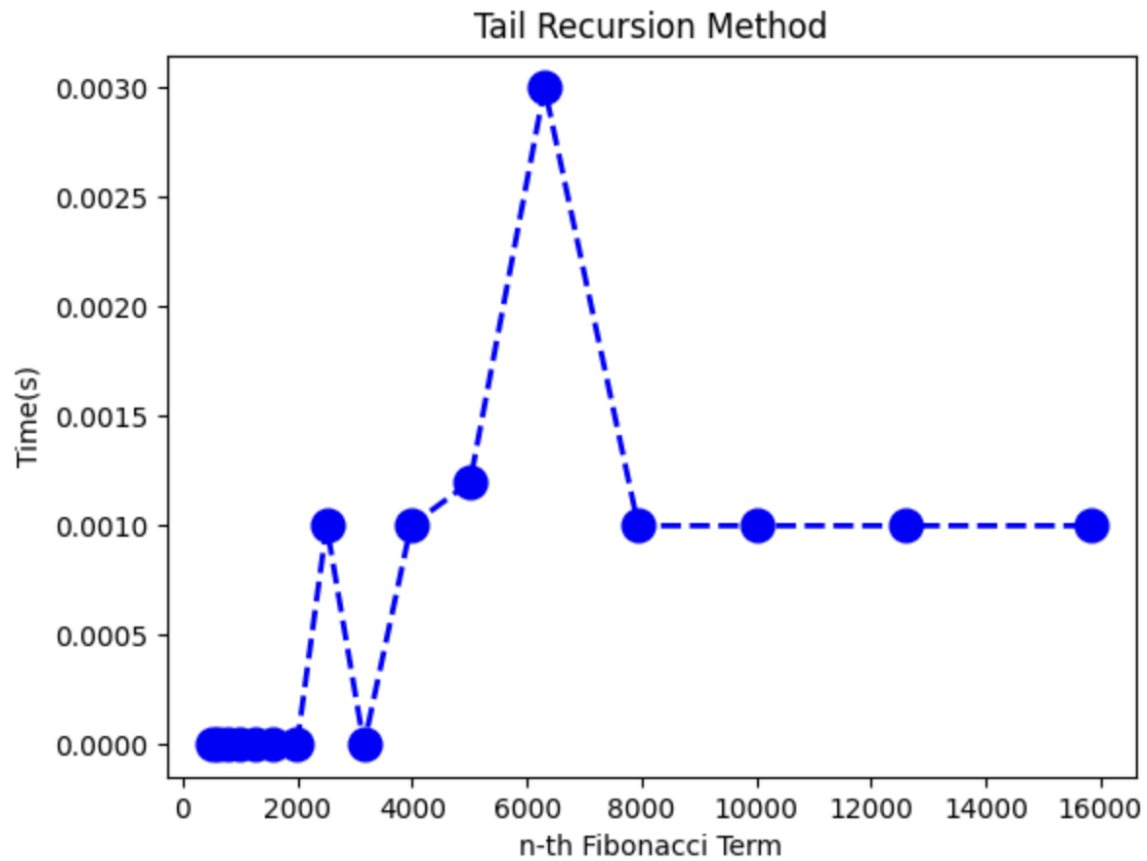


Figure 18. Graph of Tail Recursion Fibonacci Function



## CONCLUSION

In this laboratory work we had to make the empirical analysis of the algorithms to calculate the  $n$ -th Fibonacci term. After performing this work we can make conclusions which algorithm is better for different needs.

Talking about the Recursive method, which is the most straightforward one, we can observe that even if it's simple and works fine, we can't use it for large numbers because it has exponential time complexity. It can be used to calculate up to 40-th Fibonacci term.

Binet's formula method is the most efficient one, but the problem is that it is not accurate enough to be considered to calculate the large numbers. Even if the time complexity of this algorithm is  $T(1)$ , it produces rounding errors for large values of  $n$ . The Binet formula is inaccurate for very large values of  $n$ , due to the fact that it uses the Golden Ratio and produces rounding errors.

The Matrix Multiplication method is slower than the Tail recursion, Dynamic Programming and Iterative methods, but its Time Complexity can be reduced to the logarithmic ( $T(\log(n))$ ). This method is accurate and in its naïve form it has the linear Time Complexity ( $T(n)$ ).

Tail recursion, Dynamic Programming and Iterative methods showed the most efficient and accurate results compared to the other ones and they can be used to calculate the large values of  $n$ -th Fibonacci terms. All these algorithms have linear Time Complexity ( $T(n)$ ).

The link to the repository:

[https://github.com/NastiaCu/AA\\_LABS](https://github.com/NastiaCu/AA_LABS)