**Ministerul Educaţiei și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

**REPORT**

Laboratory work no.1

*Study and Empirical Analysis of Algorithms*

*for Determining Fibonacci N-th Term*

Elaborated:

st. gr. FAF-213 Botnari Ciprian

Verified:

asist. univ. Fiștic Cristofor

Chişinău – 2023

**TABLE OF CONTENTS**

[**ALGORITHM ANALYSIS**](#page3)[**3**](#page3)

[Objective](#page3) [3](#page3)

[Tasks](#page3) [3](#page3)

[Theoretical Notes:](#page3) [3](#page3)

[Introduction:](#page4) [4](#page4)

[Comparison Metric:](#page4) [4](#page4)

[Input Format:](#page4) [4](#page4)

[**IMPLEMENTATION**](#page5)[**5**](#page5)

[Recursive Method:](#page5) [5](#page5)

[Dynamic Programming Method:](#page6) [6](#page6)

[Matrix Power Method:](#page8) [8](#page8)

[Binet Formula Method:](#page11) [11](#page11)

[**CONCLUSION**](#page12)[**13**](#page12)

**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 approach to evaluating the complexity of an algorithm is through empirical analysis, which can provide insights on the complexity class of the algorithm, comparison of efficiency between algorithms solving the same problem, comparison of different implementations of the same algorithm, and performance on specific computers.

The process of empirical analysis typically involves:

1. Establishing the purpose of the analysis
2. Choosing the efficiency metric (number of operations or execution time)
3. Defining the properties of the input data
4. Implementing the algorithm in a programming language
5. Generating input data
6. Running the program with each set of data
7. Analyzing the results

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:**

Fibonacci numbers, commonly denoted Fn , form a sequence, the Fibonacci sequence, in which each number is the sum of the two preceding ones. The sequence commonly starts from 0 and 1, although some authors start the sequence from 1 and 1 or sometimes (as did Fibonacci) from 1 and 2. Starting from 0 and 1, the first few values in the sequence are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144.

The Fibonacci numbers were first described in Indian mathematics, as early as 200 BC in work by Pingala on enumerating possible patterns of Sanskrit poetry formed from syllables of two lengths. They are named after the Italian mathematician Leonardo of Pisa, later known as Fibonacci, who introduced the sequence to Western European mathematics in his 1202 book Liber Abaci.

Fibonacci numbers appear unexpectedly often in mathematics, so much so that there is an entire journal dedicated to their study, the Fibonacci Quarterly. Applications of Fibonacci numbers include computer algorithms such as the Fibonacci search technique and the Fibonacci heap data structure, and graphs called Fibonacci cubes used for interconnecting parallel and distributed systems. They also appear in biological settings, such as branching in trees, the arrangement of leaves on a stem, the fruit sprouts of a pineapple, the flowering of an artichoke, an uncurling fern, and the arrangement of a pine cone's bracts.

Fibonacci numbers are also strongly related to the golden ratio: Binet's formula expresses the nth Fibonacci number in terms of n and the golden ratio, and implies that the ratio of two consecutive Fibonacci numbers tends to the golden ratio as n increases. Fibonacci numbers are also closely related to Lucas numbers, which obey the same recurrence relation and with the Fibonacci numbers form a complementary pair of Lucas sequences.

Traditionally, the sequence was determined just by adding two predecessors to obtain a new number, however, with the evolution of computer science, several distinct algorithms for determination have been uncovered. These algorithms are:

1. **Recursion**
2. **Dynamic Programming**
3. **Power of the matrix + Optimized version**
4. **Binet’s formula**
5. **Space-Time Trade-Off**
6. **Modulo**

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 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:

**6, 7, 8, 10, 11, 14, 16, 18, 19, 20, 22, 24, 25, 28, 33, 34, 35, 37, 39, 40**

to accommodate the recursive method, while the second series will have a bigger scope to be able to compare the other algorithms between themselves

**2006, 2026, 2546, 3998, 5568, 8768, 9577, 12409, 14557, 14827, 23497, 24253, 27233, 38982, 39359, 40879, 47018, 56137, 64163, 88847**

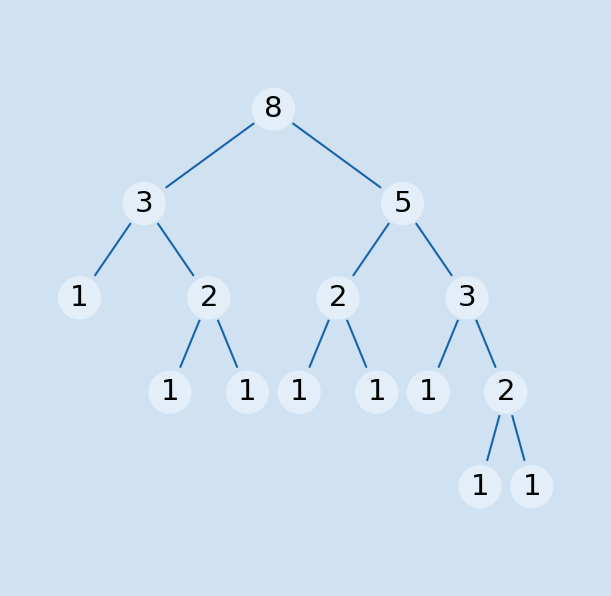
**IMPLEMENTATION**

All six algorithms will be implemented in Python an analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending on memory of the device used. The error margin determined will constitute some seconds as per experimental measurement.

**Recursive**

The recursive algorithm for determining the n-th Fibonacci number is one that uses a function that calls itself in order to calculate the desired result. The basic idea behind it is to use the definition of the Fibonacci sequence, which states that each number is the sum of the two preceding ones. This implementation uses a recursive function to calculate the nth Fibonacci number by breaking down the problem into smaller subproblems. If n is 0 or 1, the function returns n directly. Otherwise, it calculates the (n - 1) and (n - 2) Fibonacci numbers using two separate function calls, and then returns their sum as the result.

It's important to note that this implementation can be quite slow for large values of n, as it calculates many terms multiple times, leading to a large amount of redundant work. This can be addressed through the use of memoization, which caches intermediate results to avoid repeating calculations.



*Figure 1. Fibonacci Recursion*

***Algorithm Description:***

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

Fibonacci(n):

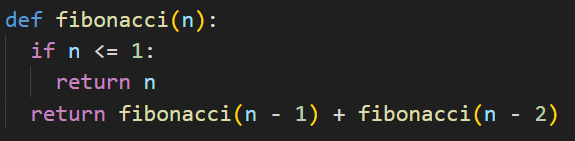
if n <= 1:

return n

otherwise:

return Fibonacci(n-1) + Fibonacci(n-2)

***Implementation:***

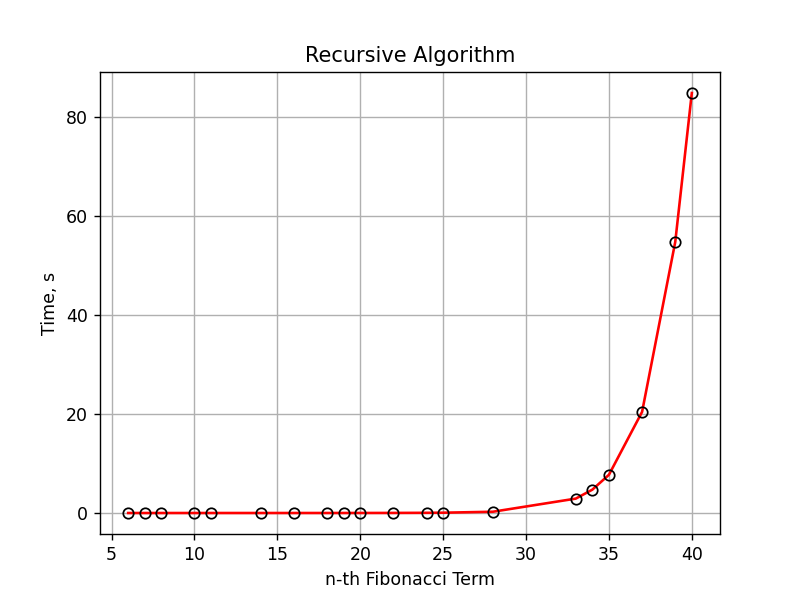


*Figure 2. Fibonacci Recursion in Python*

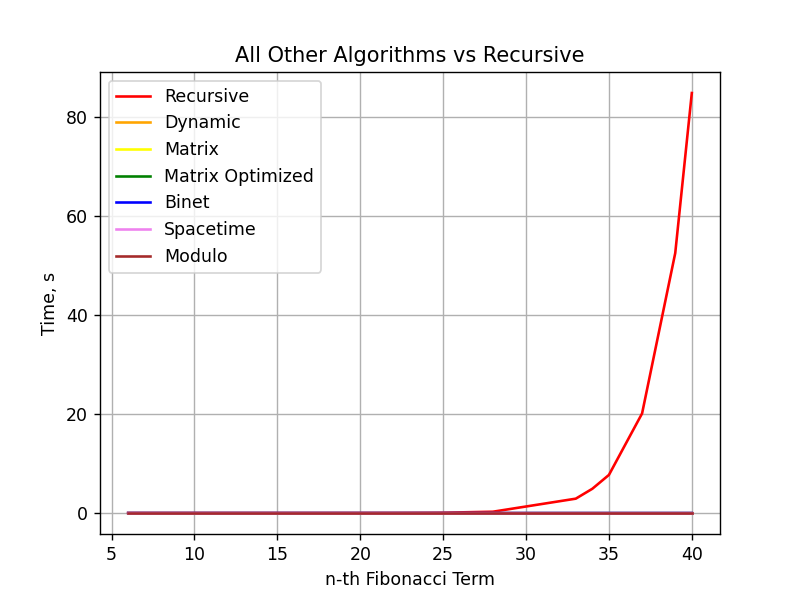
***Results:***

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **n-th term,**  **seconds** | **6** | **7** | **8** | **18** | **19** | **20** | **22** | **24** | **25** | **28** | **33** | **34** | **37** | **39** | **40** |
| ***Recursive*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.01*** | ***0.036*** | ***0.059*** | ***0.252*** | ***2.935*** | ***4.827*** | ***7.619*** | ***54.870*** | ***87.590*** |
| Dynamic | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Matrix | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Matrix optimized | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Binet | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Space | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Modulo | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |

The numbers in bold represent n-th Fibonacci term and the values for each Algorithm represent the time in seconds and I measure that using the function **perf\_counter** from **time** module. We could notice the spike in time around 37-39 terms which tell a lot about the behavior and setbacks of this algorithm. From the graphs below we could deduce that the time complexity is T(), which means it is exponential.



*Figure 3. Graph of Fibonacci Recursion*



*Figure 4. Graph of Fibonacci Recursion vs other algorithms*