**Ministry of Education and Research of the Republic of Moldova**

**Technical University of Moldova**

**Faculty of Computers, Informatics and Microelectronics**

**REPORT**

Laboratory work no. 1

*to Analysis and Design of Algorithms*

Did:

St. gr. TI-21X Nume Prenume

Checked:

asist. univ. Ceban Andrei

Chişinău - 2022

**Laboratory work no. 1**

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

**The purpose of the work:**

1. Empirical analysis of algorithms

2. Theoretical analysis of algorithms

3. Determining the time and asymptotic complexity of algorithms

**Task**:

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.

**Short summary of the topic of the laboratory work:**

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: xn= xn-1 + xn-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 4 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:**

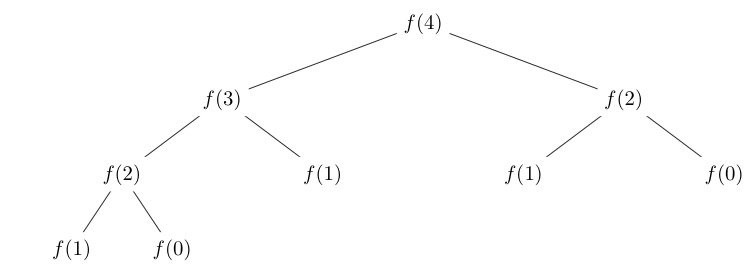
As input, each algorithm will receive two series of numbers that will contain the order of the Fibonacci terms being looked up. The first series will have a more limited scope, (5, 7, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 37, 40, 42, 45), to accommodate the recursive method, while the second series 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 seven algorithms will be implemented in their naïve form 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 o memory of the device used.

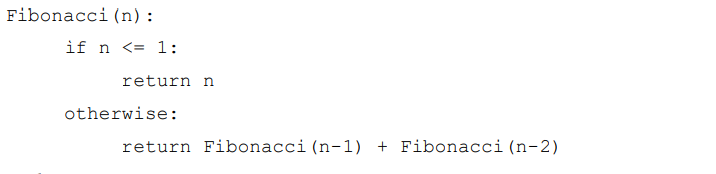
**Recursive Method:**

The recursive method, also considered the most inefficient method, follows a straightforward approach of computing the n-th term by computing it’s predecessors first, and then adding them. However, the method does it by calling upon itself a number of times and repeating the same operation, for the same term, at least twice, occupying additional memory and, in theory, doubling it’s execution time.

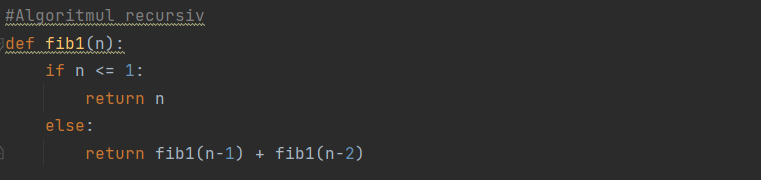


*Algorithm Description:*

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

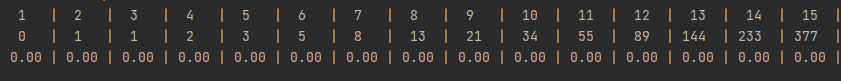


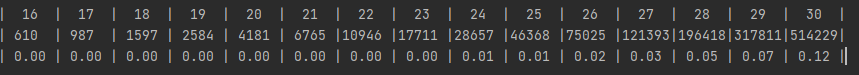
*Implementation:*



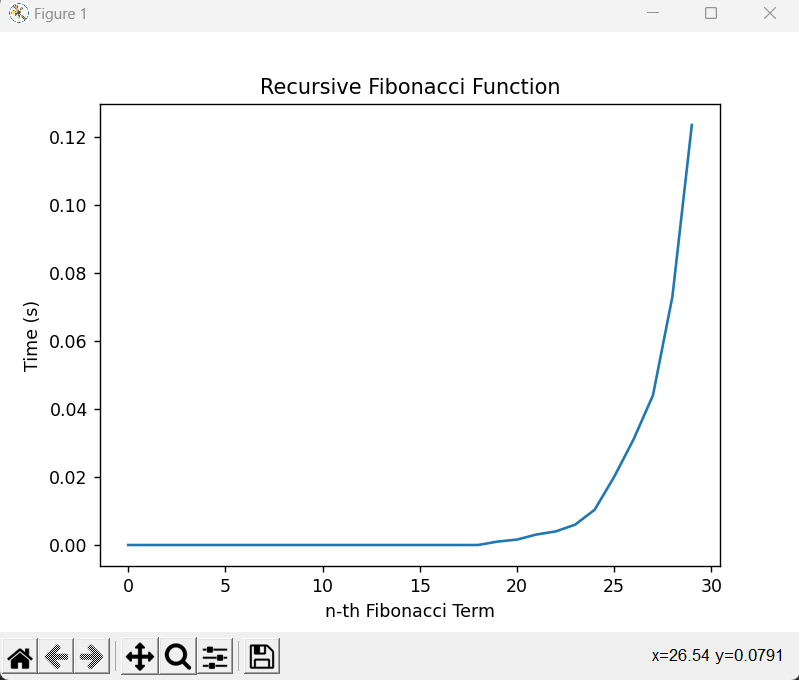
*Results:*

After running the function for each n Fibonacci term proposed in the list from the first Input Format and saving the time for each n, we obtained the following results:





Here is represented the table of results for the first set of inputs. The highest line(the name of the columns) denotes the Fibonacci n-th term for which the functions were run. Starting from the second row, we get the number of seconds that elapsed from when the function was run till when the function was executed. We may notice that the only function whose time was growing for this few n terms was the Recursive Method Fibonacci function.



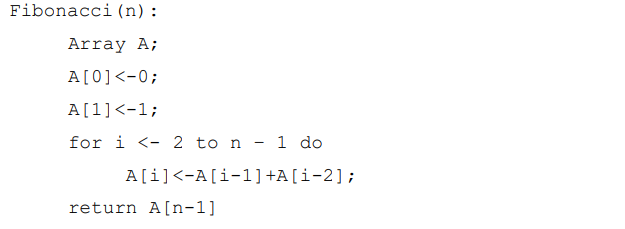
In the graph shows the growth of the time needed for the operations, we may easily see the spike in time complexity that happens after the 30nd term, leading us to deduce that the Time Complexity is exponential T().

**Dynamic Programming Method:**

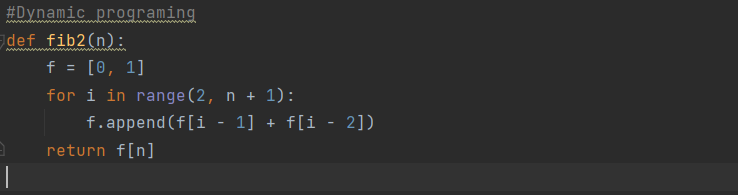
The Dynamic Programming method, similar to the recursive method, takes the straightforward approach of calculating the n-th term. However, instead of calling the function upon itself, from top down 7 it operates based on an array data structure that holds the previously computed terms, eliminating the need to recompute them.

*Algorithm Description:*

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

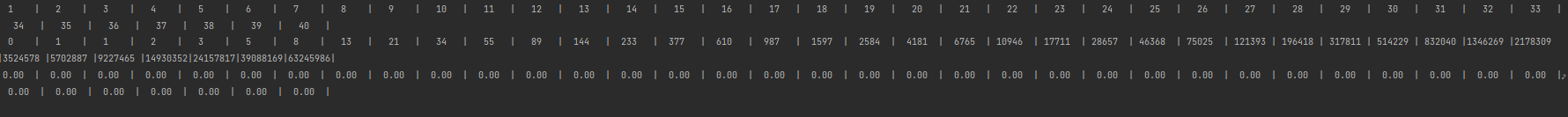


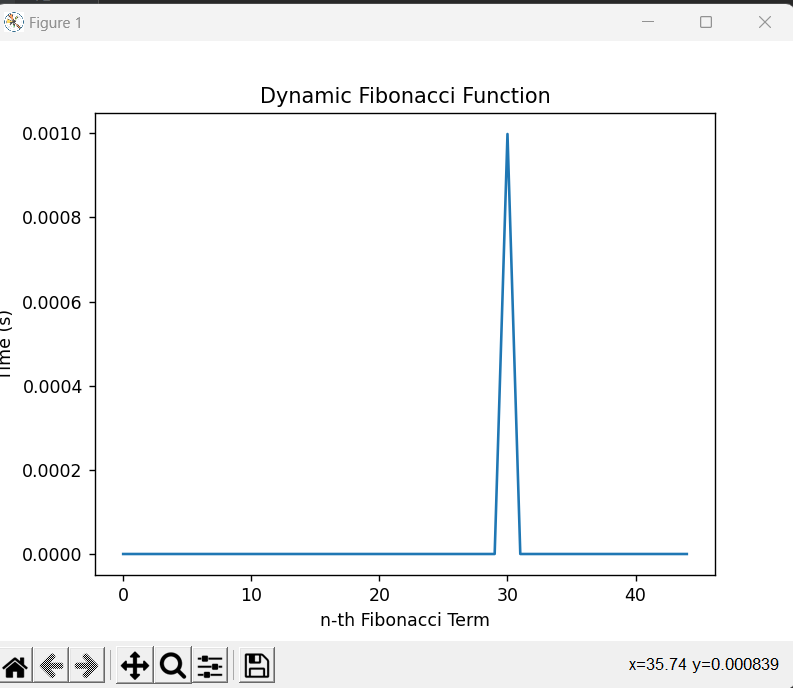
*Implementation:*



*Results:*

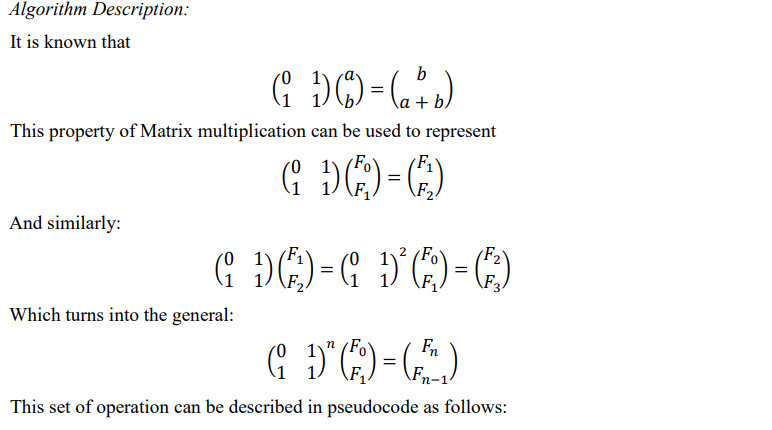
After the execution of the function for each n Fibonacci term mentioned in the second set of Input Format we obtain the following results:



With the Dynamic Programming Method (first row, row[0]) showing excellent results with a time complexity denoted in a corresponding graph of T(n)

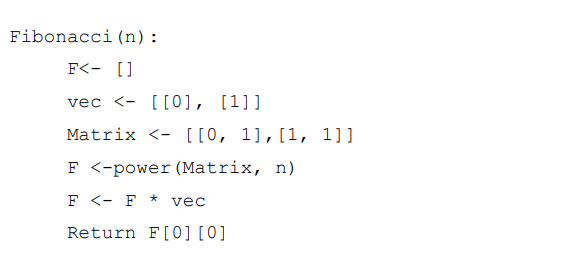
**Matrix Power Method:**

The Matrix Power method of determining the n-th Fibonacci number is based on, as expected, the multiple multiplication of a naïve Matrix



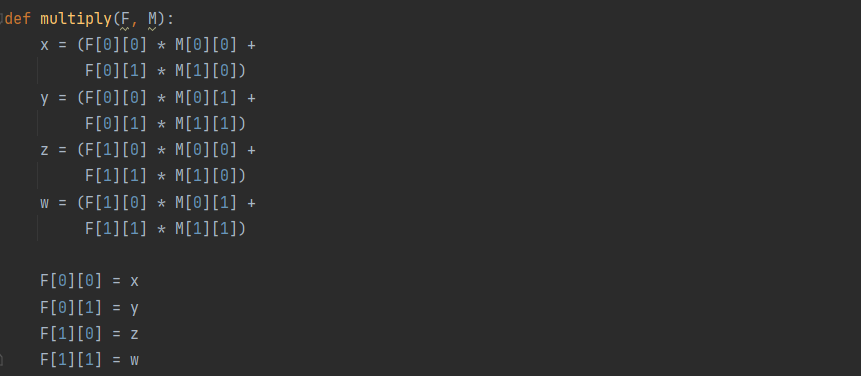
*Algorithm Description:*

The naïve MP algorithm for Fibonacci n-th term follows the pseudocode:



*Implementation:*

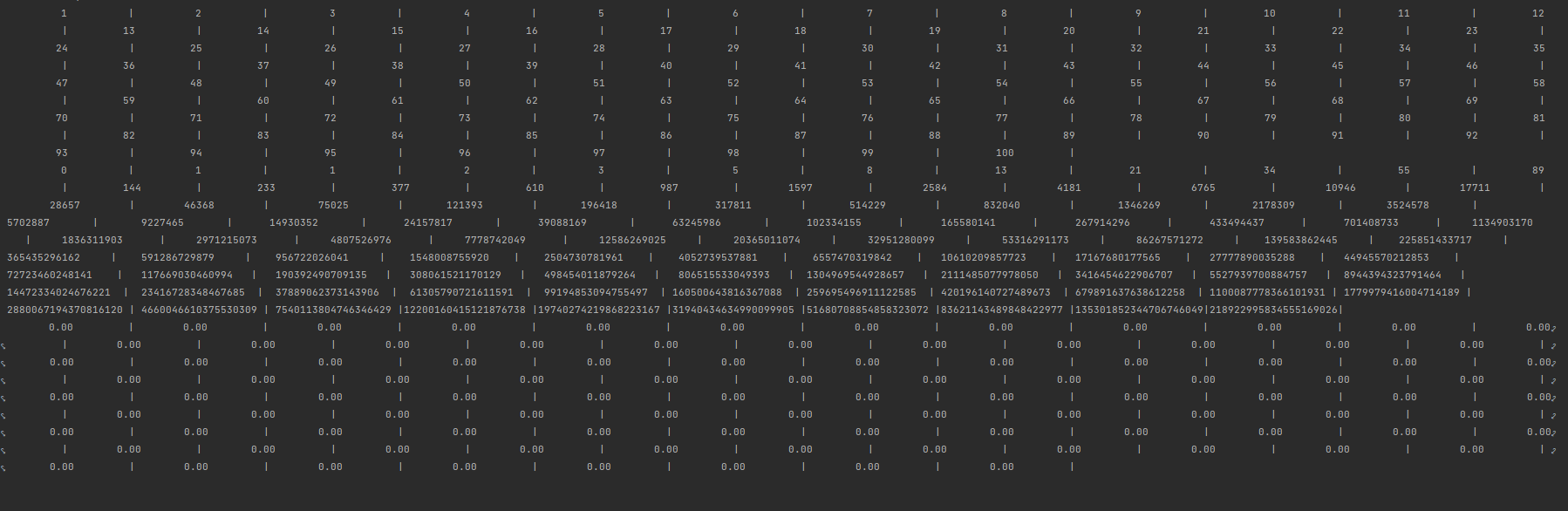


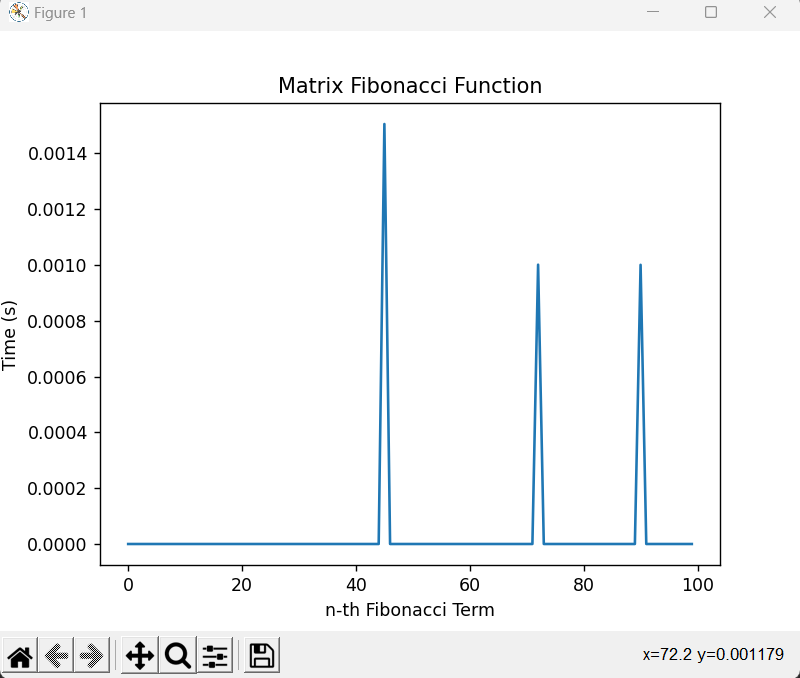




*Results:*

After the execution of the function for each n Fibonacci term mentioned in the second set of Input Format we obtain the following results:



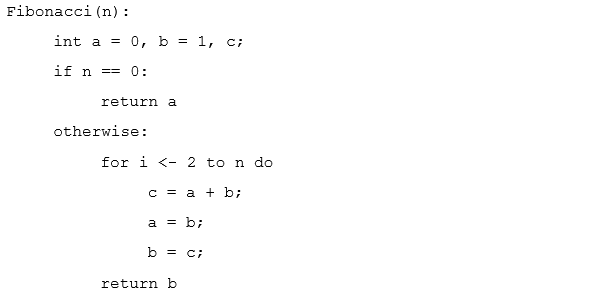
With the naïve Matrix method (indicated in last row, row[2]), although being slower than the Binet and Dynamic Programming one, still performing pretty well, with the form f the graph indicating a pretty solid T(n) time complexity 

**Space Optimized method:**

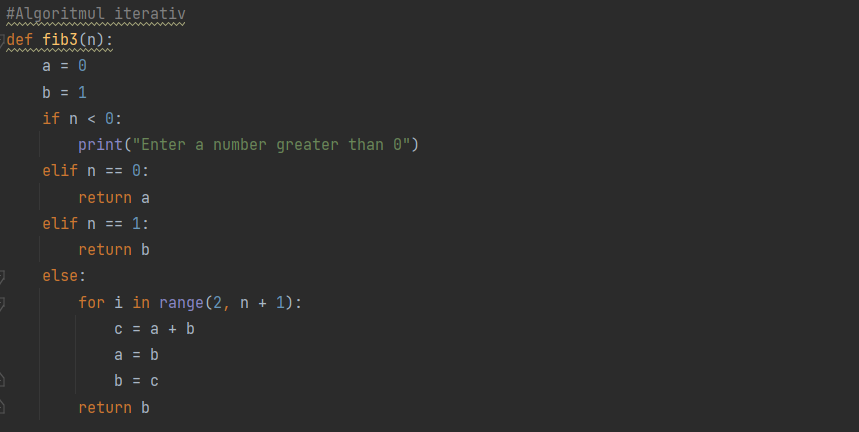
The given method is based on the dynamic programming method. Only that some optimizations are used in order not to load the computer with data, this fact can help to calculate larger numbers without overloading the computer. We can optimize the space used in dynamic programming method by storing the previous two numbers only because that is all we need to get the next Fibonacci number in series.

*Algorithm Description:*

The naïve Space Optimized method follows the algorithm as shown in the next pseudocode:

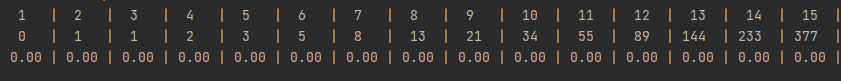


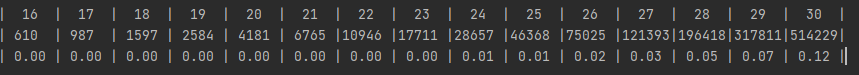
*Implementation:*

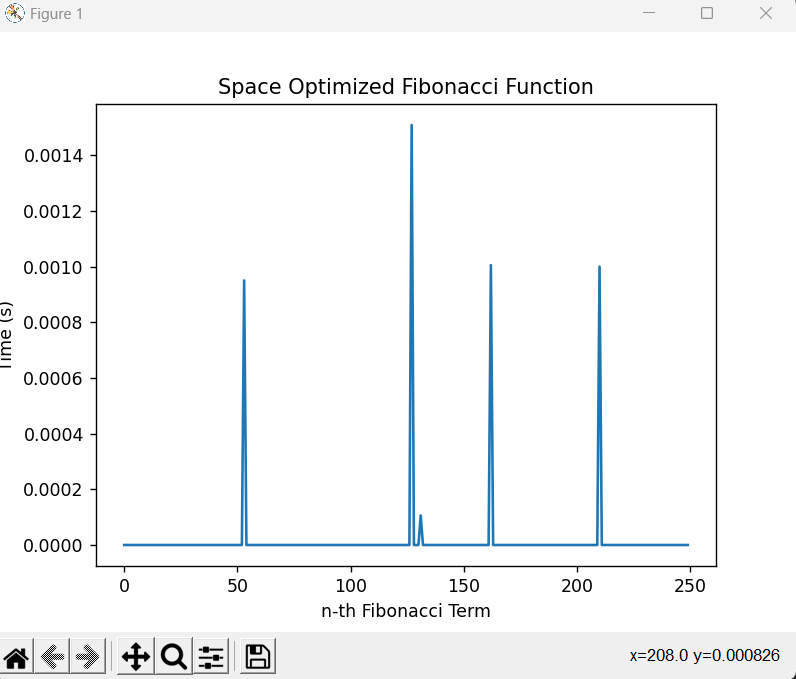


*Results:*

After running the function for each n Fibonacci term proposed in the list from the first Input Format and saving the time for each n, we obtained the following results:





. 

As we can see in the given diagram, the execution and calculation time is much lower than in the previous one. For small numbers, the execution time is almost constant, but when the order increases, the calculation time also increases, which we can observe in the 100th Fibonacci number. If we increase the order, the number of jumps in the diagram will also increase. From here we can deduce that the Time Compelexity is T(n).

**Recursive T(Logn) Time method:**

In the given method, the deduction of a formula from the matrix method is used and memoization is used for faster calculation of the terms. This method is one of the fastest methods of calculating Fibonacci numbers that also uses recursion.

*Algorithm Description:*

Below is one more interesting recurrence formula that can be used to find n’th Fibonacci Number in O(Log n) time.

If n is even then k = n/2: F(n) = [2\*F(k-1) + F(k)] \*F(k)

If n is odd then k = (n + 1)/2: F(n) = F(k)\*F(k) + F(k-1) \*F(k-1)

The formula can be derived from the above matrix equation:

Taking determinant on both sides, we get

Moreover, since for any square matrix A,

the following identities can be derived (they are obtained

from two different coefficients of the matrix product)

By putting n = n+1 in equation(1),

(2)

Putting m = n in equation(1).

Putting m = n in equation(2)

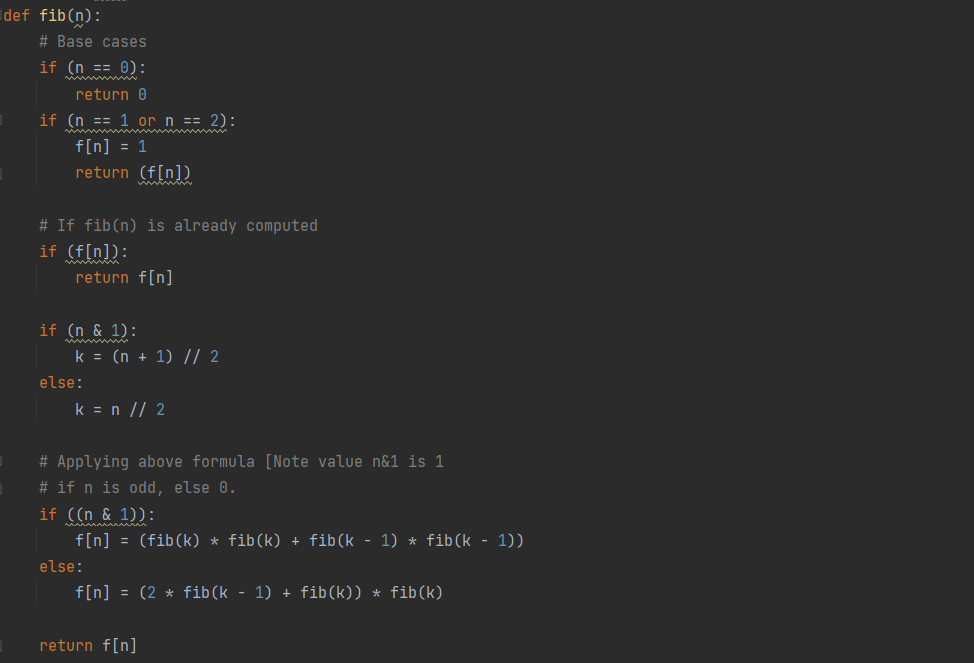
By putting

To get the formula to be proved, we simply need to do the following

If n is even, we can put k = n/2

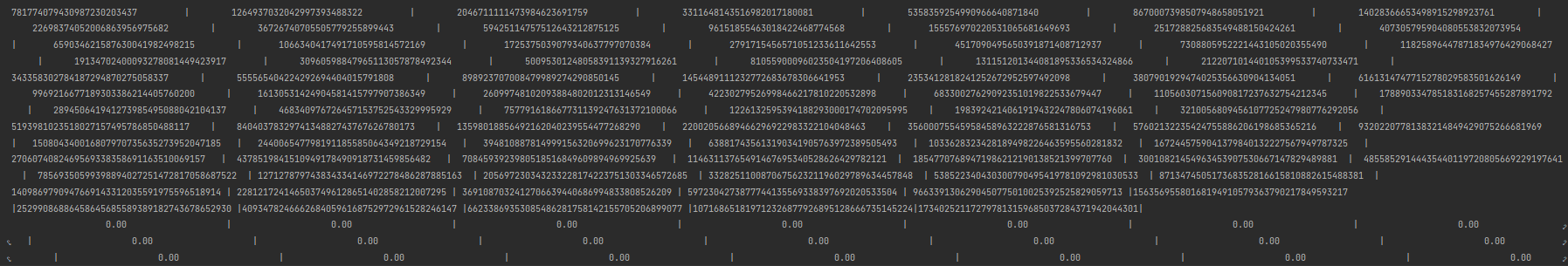
If n is odd, we can put k = (n+1)/2

*Implementation:*



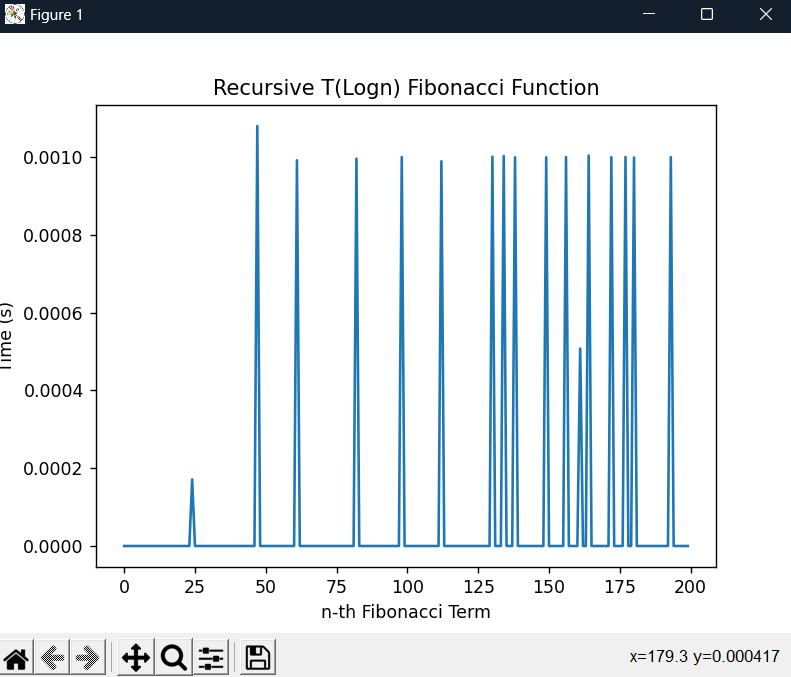
*Results:*

After running the function for each n Fibonacci term proposed in the list from the first Input Format and saving the time for each n, we obtained the following results:



.

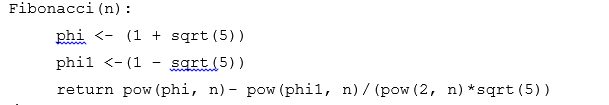
As we can see this method is much faster than all the methods presented above. From here we can deduce that the Time Compelexity is T(Logn).



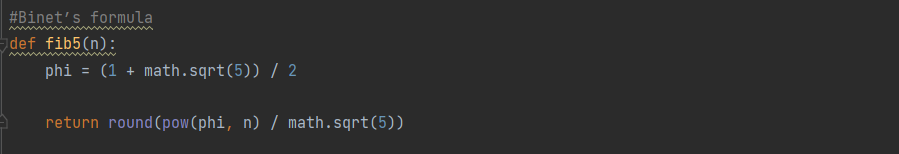
**Binet’s formula method:**

The explicit formula for the terms of the Fibonacci sequence, **Fn=(1+√52)n−(1−√52)n√5**. has been named in honor of the eighteenth century French mathematician Jacques Binet, although he was not the first to use it. However, because Python's nature necessitates the use of decimal numbers, eventually the rounding error that accumulates starts to have a substantial impact on the outcomes.

*Algorithm Description:*



*Implementation:*



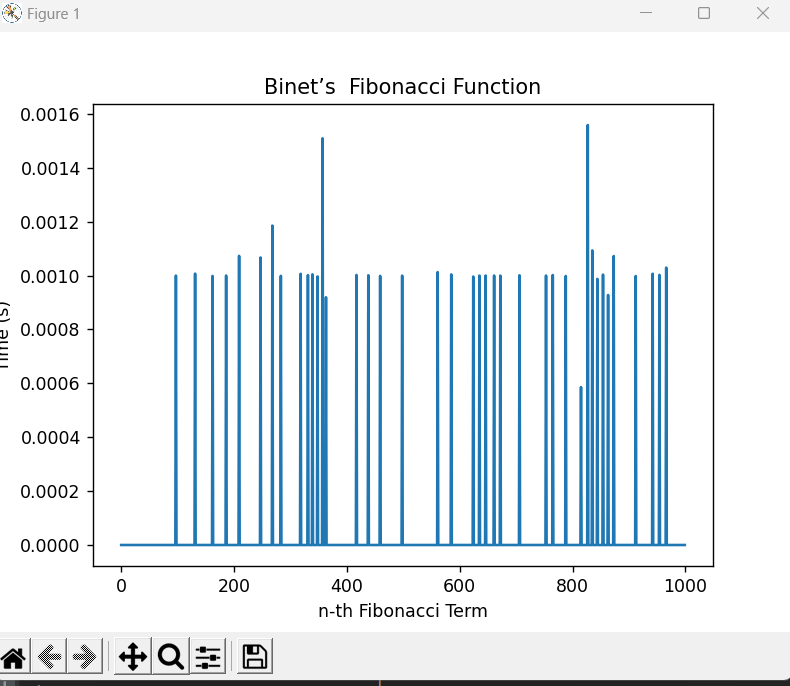
*Results:*

After running the function for each n Fibonacci term proposed in the list from the first Input Format and saving the time for each n, we obtained the following results:



.

This method has a very the low precision. The large Fibonacci numbers are very different from the number that is calculated by the given method. The Time Compelexity for this algorithm is T(n).



**CONCLUSION**

The purpose of the proposed work, which counts in the empirical and theoretical analysis of algorithms for determining the nth Fibonacci number, was successfully fulfilled. The comparison of the algorithms was carried out with the help of the table in which we collected the empirical data about the iterations received during the work and the graphs which show us the visible difference between the complexities of the algorithms. From here we notice that the recursive method is the most inefficient, because it recalculates the same values several times, while the formula method gave us much more favorable results. The iterative algorithm is also a sufficiently good one, having a linear complexity. Therefore, to determine the nth Fibonacci number, the iterative method and the formula method will be used. Through Empirical Analysis, within this paper, four classes of methods have been tested in their efficiency at both their providing of accurate results, as well as at the time complexity required for their execution, to delimit the scopes within which each could be used, as well as possible improvements that could be further done to make them more feasible.

The Recursive method, being the easiest to write, but also the most difficult to execute with an exponential time complexity, can be used for smaller order numbers, such as numbers of order up to 30 with no additional strain on the computing machine and no need for testing of patience.

The Binet method, the easiest to execute with an almost constant time complexity, could be used when computing numbers of order up to 80, after the recursive method becomes unfeasible. However, its results are recommended to be verified depending on the language used, as there could rounding errors due to its formula that uses the Golden Ratio.

The Dynamic Programming and Matrix Multiplication Methods can be used to compute Fibonacci numbers further then the ones specified above, both of them presenting exact results and showing a linear complexity in their naivety that could be, with additional tricks and optimisations, reduced to logarithmic.

Git Repo : *https://github.com/andeiceban0352/FLAB*