Ministerul Educaţiei şi 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 Coreţchi Mihai

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2023

TABLE OF CONTENTS

ALGORITHM ANALYSIS **3**
Objective 3
Tasks 3
Theoretical Notes: 3
Introduction: 4
Comparison Metric: 4
Input Format: 4
IMPLEMENTATION **5**
Recursive Method: 5
Iterative Method: 6
Dynamic Programming Method: 7
Power Matrix Method: 9
Matrix Reccurence Method: 10
Binet Formula Method: 11
CONCLUSION **12**

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: 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, random number given by the user.

IMPLEMENTATION

All 6 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 o memory of the device used.

The error margin determined will constitute 1 seconds as per experimental measurement.

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.

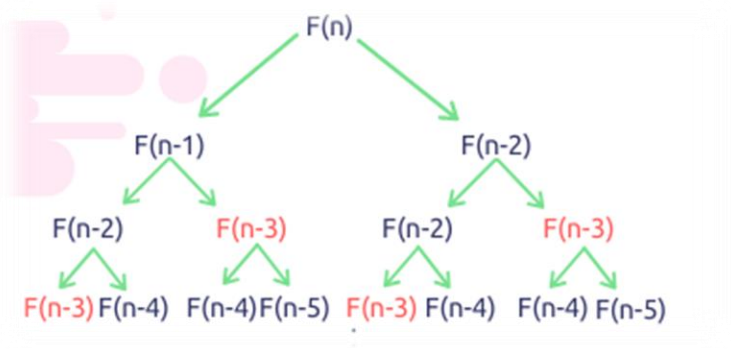

Figure 1 Fibonacci Recursion

Algorithm Desc

The naïve 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:

```
def Fib(n):
    if n <= 1:
        return n
    else:
        return Fib(n-1) + Fib(n-2)
```

Figure 2 Fibonacci recursion

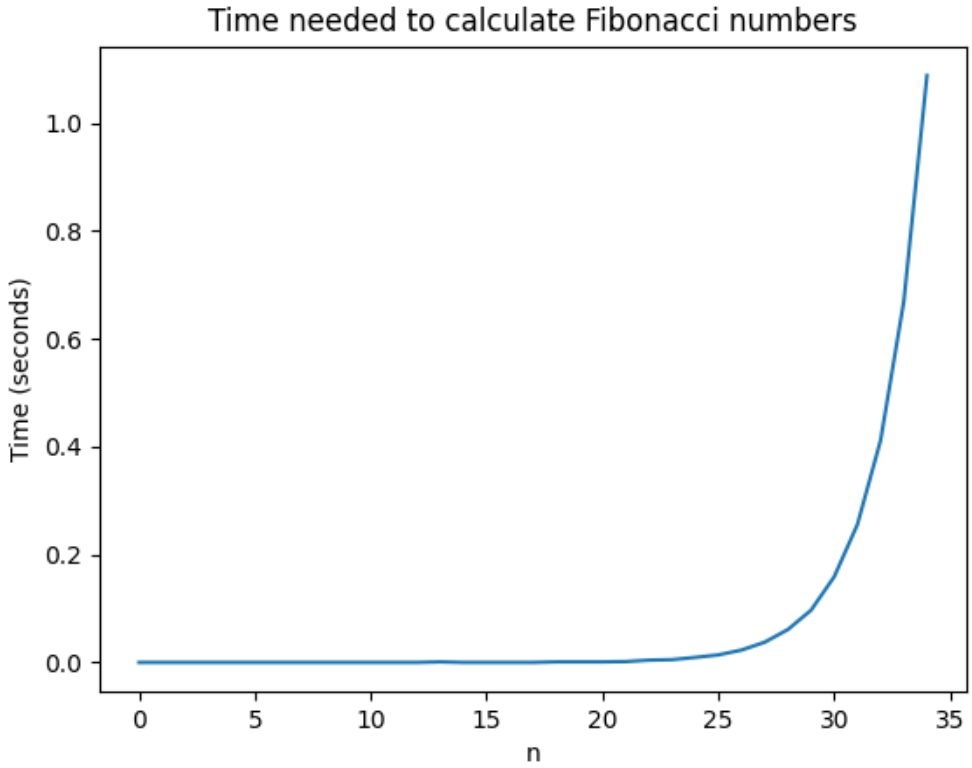

Figure 4 Graph of Recursive Fibonacci Function

Not only that, but also in the graph in Figure 4 that shows the growth of the time needed for the operations, we may easily see the spike in time complexity that happens after the 35th term, leading us to deduce that the Time Complexity is exponential. T(2^n).

Iterative Method:

This solution is one of the simplest and most efficient algorithms on this list. This function has a loop that runs through n iterations before returning, and each iteration does constant work, making the algorithm run in $O(n)$ time. This function doesn't use any extra data structures, and it isn't recursive, so we can say that it uses $O(1)$ space. It operates based on an array data structure that holds the previously computed terms, eliminating the need to recompute them.

Implementation:

```
def fibonacci_iterative(n):
    if n <= 1:
        return n
    else:
        a = 0
        b = 1
        for i in range(2, n + 1):
            c = a + b
            a = b
            b = c
        return b
```

Figure 5 Fibonacci iterative

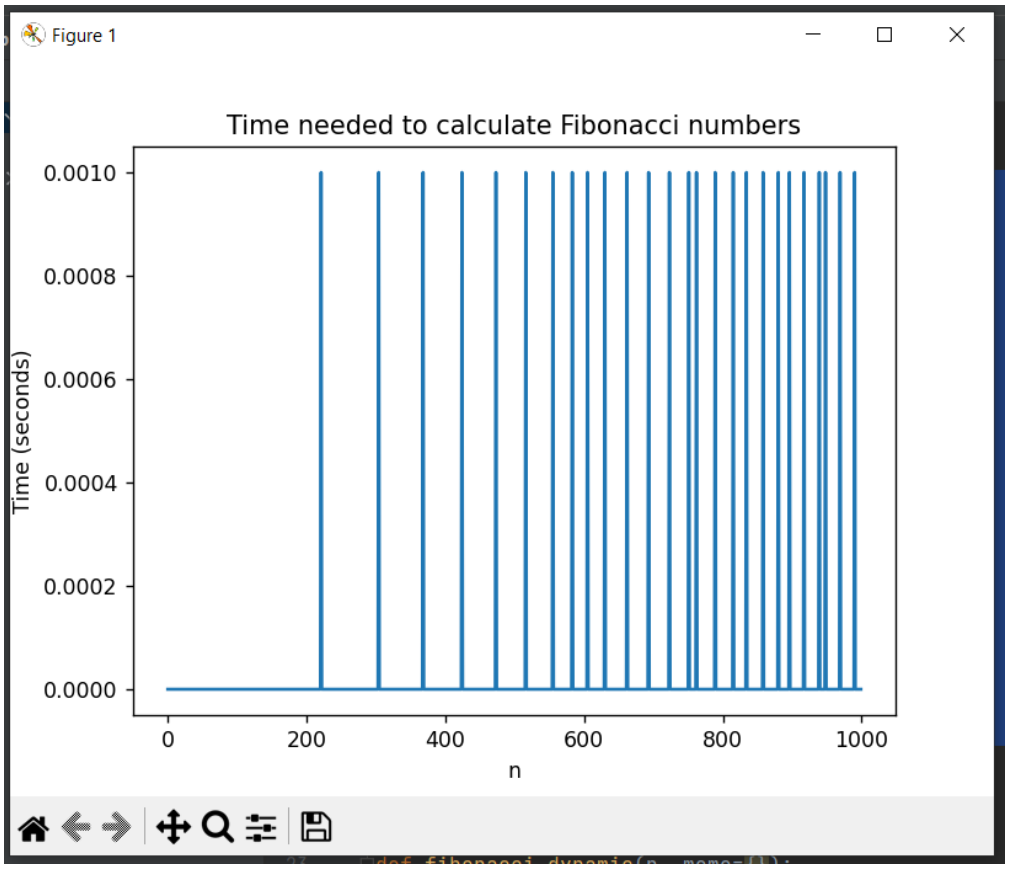

Figure 6 Graph of Iterative Fibonacci Function

Not only that, but also in the graph in Figure 6 that shows the growth of the time needed for the operations.

Dynamic Programming Method:

Dynamic Programming is mainly an optimization compared to simple recursion. The main idea is to decompose the original question into repeatable patterns and then store the results as many sub-answers. Therefore, we do not have to re-compute the prestep answers when needed later. In terms of big O, this optimization method generally reduces time complexities from exponential to polynomial.

```
def fibonacci_dynamic(n, memo={}):
    if n <= 1:
        return n
    elif n in memo:
        return memo[n]
    else:
        memo[n] = fibonacci_dynamic(n-1, memo) + fibonacci_dynamic(n-2, memo)
        return memo[n]
```

Figure 7 Dynamic Programming for Fibonacci Function

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

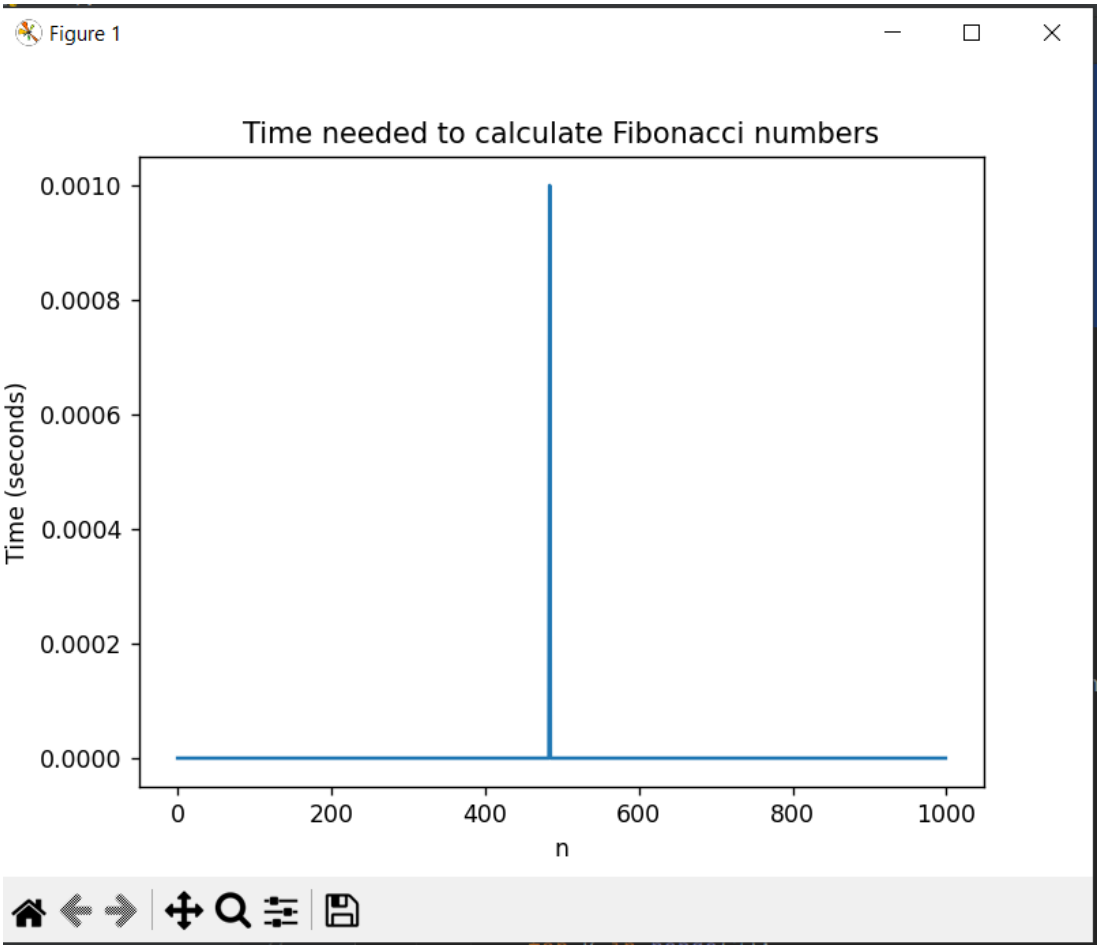

Figure 8 Graph of Dynamic Programming for Fibonacci Function

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 $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ with itself.

Algorithm Description:

It is known that

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} b \\ a+b \end{pmatrix}$$

This property of Matrix multiplication can be used to represent

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \end{pmatrix}$$

And similarly:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_2 \\ F_3 \end{pmatrix}$$

Which turns into the general:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

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

Implementation:

The implementation of the driving function is as follows:

```
def multiply(A, B):
    C = [[0, 0], [0, 0]]
    for i in range(2):
        for j in range(2):
            for k in range(2):
                C[i][j] += A[i][k] * B[k][j]
    return C

def power(A,n):
    if n <= 1:
        return A
    else:
        B = power(A, n//2)
        C = multiply(B, B)
        if n % 2 == 0:
            return C
        else:
            return multiply(C, A)

def fibonacci_matrix(n):
    if n <= 1:
        return n
    else:
        F = [[1, 1], [1, 0]]
        F_n = power(F, n-1)
        return F_n[0][0]
```

Figure 9 Power Function

Where the power function (Figure 10) handles the part of raising the Matrix to the power n and the multiplying function handles the matrix multiplication with itself.

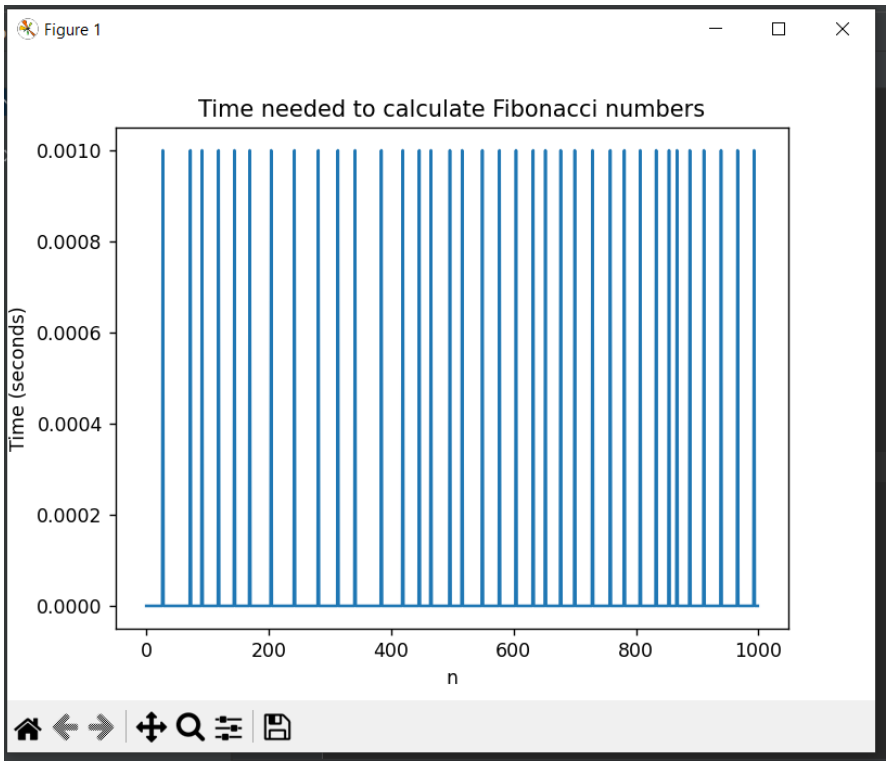

Figure 10 Matrix Method Fibonacci graph

Matrix Recurrence Method:

```
def fibonacci_recurrence_matrix(n):
    if n <= 1:
        return n
    else:
        F = [[1, 1], [1, 0]]
        F_n = [[1, 0], [0, 1]]
        for i in range(n):
            F_n = multiply_r(F_n, F)
        return F_n[0][0]
```

The recurrence matrix method of the Fibonacci sequence is an efficient way to calculate the n-th Fibonacci number, where n is a positive integer. This method is based on the observation that the Fibonacci sequence satisfies a linear recurrence relation.
The linear recurrence relation can be represented as a matrix equation: F_n = F_{n-1} + F_{n-2}, where F_n is the n-th Fibonacci number, F_{n-1} is the n-1-th Fibonacci number, and F_{n-2} is the n-2-th Fibonacci number.

Figure 11 Matrix Recurrence Method Fibonacci

So, to calculate the n-th Fibonacci number, we can multiply the matrix [[1, 1], [1, 0]] with itself n times and return the first element of the resulting matrix. The matrix multiplication can be done using the multiply function defined in the code.

This method is efficient because it has a time complexity of O(log n) for each matrix multiplication, so the total time complexity for calculating the n-th Fibonacci number is O(log n).

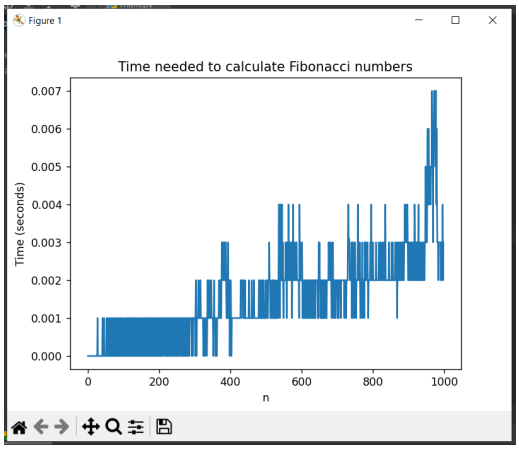

Figure 12 Matrix Recurrence Method Fibonacci graph

Binet Formula Method:

The Binet Formula Method is another unconventional way of calculating the n-th term of the Fibonacci series, as it operates using the Golden Ratio formula, or phi. However, due to its nature of requiring the usage of decimal numbers, at some point, the rounding error of python that accumulates, begins affecting the results significantly. The observation of error starting with around 70-th number making it unusable in practice, despite its speed. O(1)

```
def fibonacci_closed_form(n):
    if n <= 1:
        return n
    else:
        golden_ratio = (1 + math.sqrt(5)) / 2
        return int(round((golden_ratio**n - (1 - golden_ratio)**n) / math.sqrt(5)))
```

Figure 13 Binet formula implementation

And as shown in its performance graph,

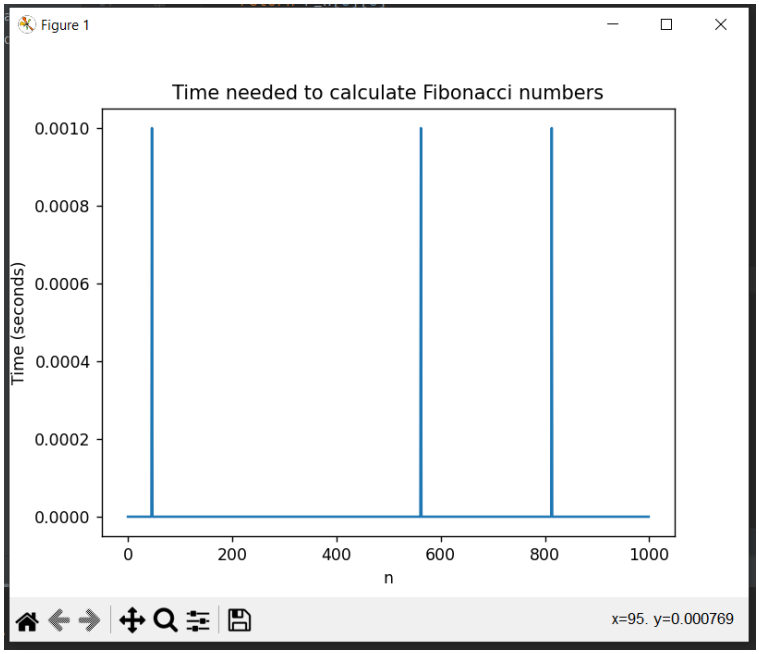

Figure 14 Fibonacci Binet formula graph

CONCLUSION

In conclusion, Fibonacci numbers are an important aspect of mathematics with numerous applications in various fields such as computer science, engineering, and biology. The algorithms used to calculate Fibonacci numbers can vary in terms of time and space complexity, and the choice of algorithm depends on the specific requirements of a given problem. Some of the most popular algorithms include the recursive approach, dynamic programming, and the closed-form formula. The recursive approach is simple but can be slow and inefficient for large values of n, while dynamic programming provides a more efficient solution by storing previously calculated values. The closed-form formula, also known as Binet's formula, provides an exact solution for Fibonacci numbers and is useful for theoretical purposes. Overall, it is essential to understand the different algorithms for Fibonacci numbers to be able to choose the right approach for specific problem-solving needs. 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 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.

The Matrix Multiplication Fibonacci algorithm requires a lot of matrix operations, which can be slow and computationally expensive, especially for large values of n.

The Matrix Recurrence Fibonacci algorithm is particularly useful in computational mathematics and computer science, where the efficient calculation of large Fibonacci numbers is often required. For example, the Fibonacci numbers are used in cryptography, coding theory, and computer graphics, and the matrix recurrence algorithm provides a fast method for calculating these numbers in these applications.

The Iterative Fibonacci algorithm is widely used in many different fields and applications due to its simplicity and ease of implementation. For example, it is used in computer programming courses as a way to teach basic concepts such as loops, functions, and recursion. And also this method is much slower than the matrix methods and the dynamic programming method for finding the nth number in the Fibonacci sequence.

To sum up, if you want to find the exact value of a n-th number in the sequence, in the fastest way, you should use the matrix power algorithm for a greater numbers.

Github repo: https://github.com/eamtcPROG/AA/tree/main/Lab1

Console Screens:

```
F:\Programare\UTM_LAB\P4_2\Scripts\python.exe F:/UTM/AA/Lab1/main.py
Enter the n: 30
Fibonacci sequence Recursive method
-------------------------------------------------
30. - Value: 514229 - Time: 0.2633 - Total time: 0.5435624122619629
-------------------------------------------------

Enter the n: 1000

Fibonacci sequence Iterative method
-------------------------------------------------
1000. - Value: 268638100244853593861467272021429239676166093189869523401231759976179817002478
-------------------------------------------------

Enter the n: 1000

Fibonacci sequence Dynamic programming
-------------------------------------------------
1000. - Value: 268638100244853593861467272021429239676166093189869523401231759976179817002478
```

```
74626 - Time: 0.002 - Total time: 1.219712734222412



174626 - Time: 0.0 - Total time: 0.030883312225341797



192064 - Time: 0.0 - Total time: 0.0019948482513427734
```