

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

Grama Alexandru

Verified:
asist. univ.

Fiștic Cristofor

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
Reccurence Matrix Method:	9
Matrix Exponential Method:	10
Binet Formula Method:	11
CONCLUSION.....	11

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 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 four algorithms will be implemented in their naïve form in python and 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 2.5 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 its 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 its execution time.

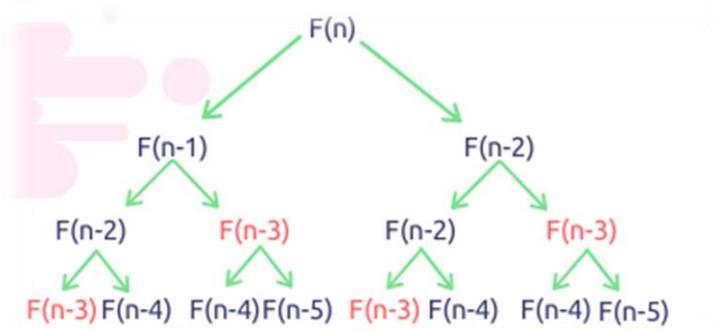


Figure 1 Fibonacci Recursion

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:

```
// Dynamic Programming  
public int fib(int n) {  
    /* Declare an array to store Fibonacci numbers. */  
    int f[] = new int[n + 1]; // 1 extra to handle case, n = 0  
    int i;  
  
    /* 0th and 1st number of the series are 0 and 1*/  
    f[0] = 1;  
    f[1] = 1;  
  
    for (i = 2; i <= n; i++) {  
        /* Add the previous 2 numbers in the series and store it */  
        f[i] = f[i - 1] + f[i - 2];  
    }  
  
    return f[n];  
}
```

Figure 2 Fibonacci recursion

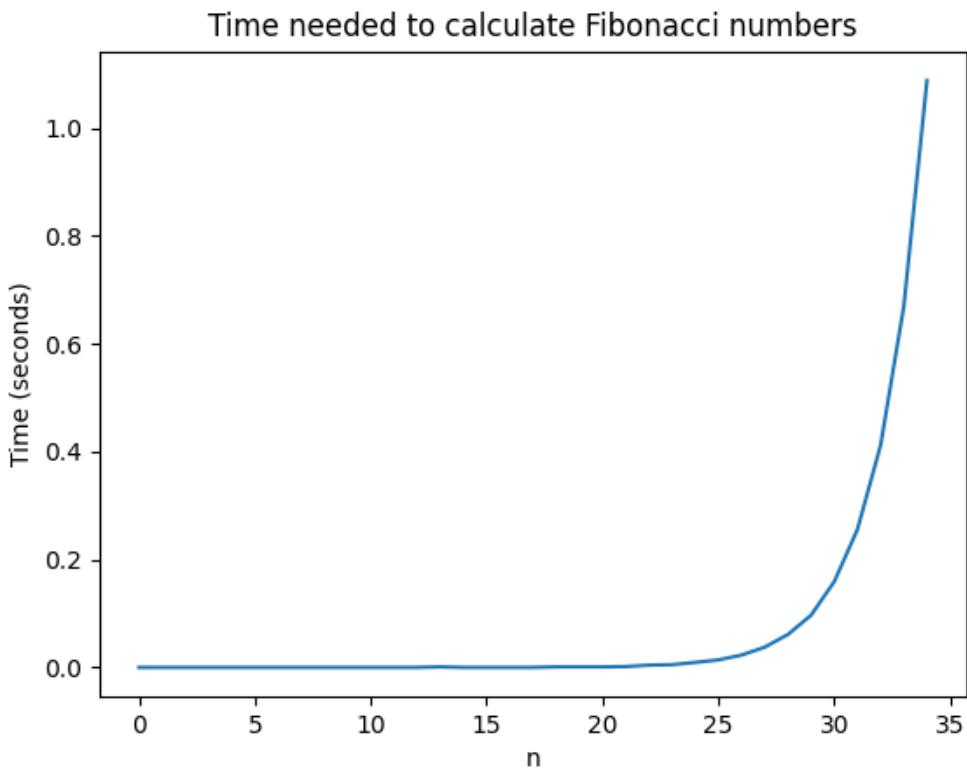


Figure 4 Graph of Recursive Fibonacci Function

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:

```
// Fibonacci Series using Space Optimized Method
class Solution {
    public int fib(int n) {
        if (n < 2) {
            return n;
        }
        int p = 0, q = 0, r = 1;
        for (int i = 2; i <= n; ++i) {
            p = q;
            q = r;
            r = p + q;
        }
        return r;
    }
}
```

Figure 5 Fibonacci iterative

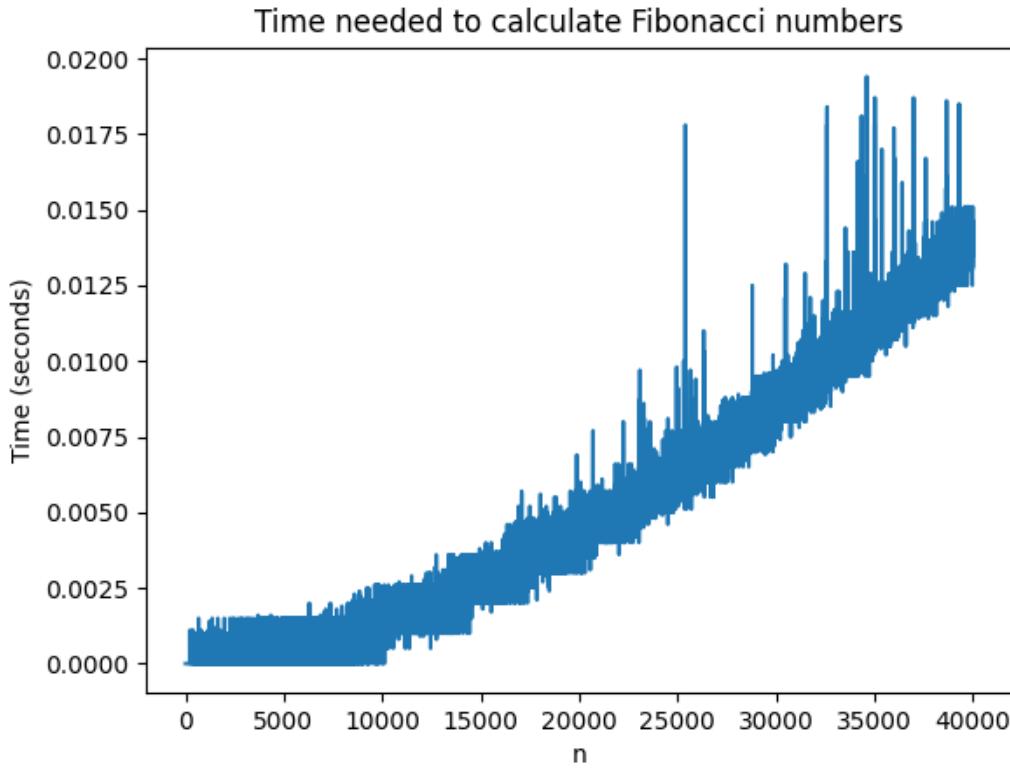


Figure 6 Graph of Iterative Fibonacci Function

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 40kth term.

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 pre-step answers when needed later. In terms of big O, this optimization method generally reduces time complexities from exponential to polynomial.

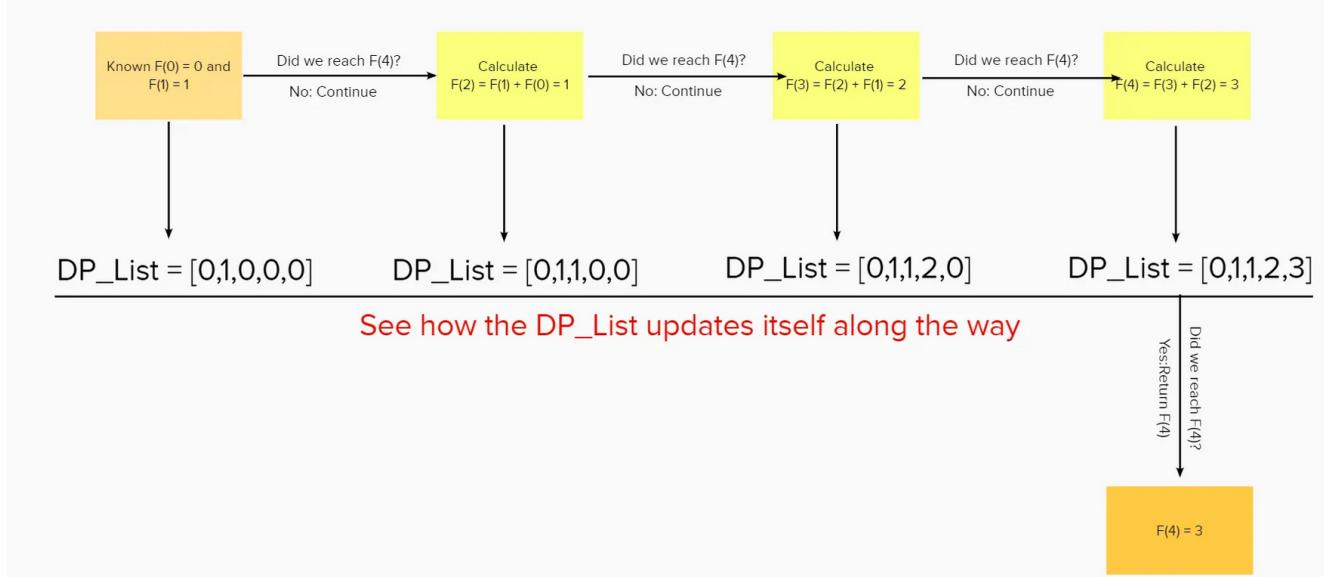


Figure 7 Dynamic Programming for Fibonacci Function

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

Implementation:

```
// Dynamic Programming
public int fib(int n) {
    /* Declare an array to store Fibonacci numbers. */
    int f[] = new int[n + 1]; // 1 extra to handle case, n = 0
    int i;

    /* 0th and 1st number of the series are 0 and 1*/
    f[0] = 1;
    f[1] = 1;

    for (i = 2; i <= n; i++) {
        /* Add the previous 2 numbers in the series and store it */
        f[i] = f[i - 1] + f[i - 2];
    }

    return f[n];
}
```

Figure 8 Dynamic Programming for Fibonacci Function

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 40kth term.

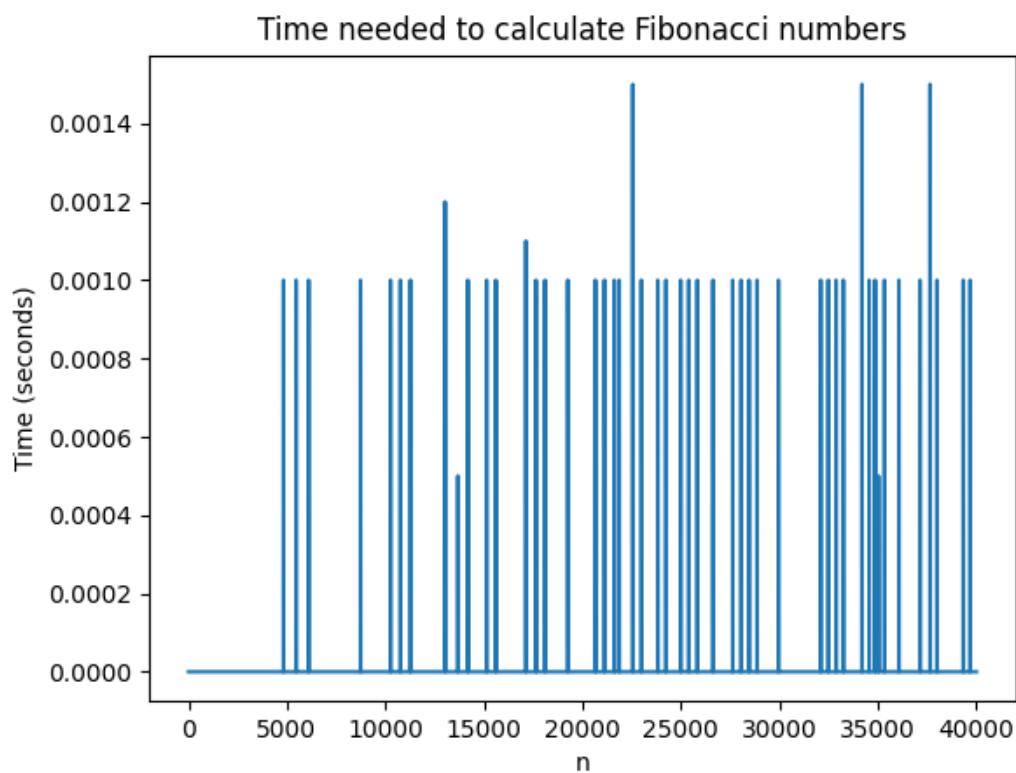


Figure 9 Graph of Dynamic Programming for Fibonacci Function

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:

```
class Solution {
    public int fib(int n) {
        if (n < 2) {
            return n;
        }
        int[][] q = {{1, 1}, {1, 0}};
        int[][] res = pow(q, n - 1);
        return res[0][0];
    }

    public int[][] pow(int[][] a, int n) {
        int[][] ret = {{1, 0}, {0, 1}};
        while (n > 0) {
            if ((n & 1) == 1) {
                ret = multiply(ret, a);
            }
            n >>= 1;
            a = multiply(a, a);
        }
        return ret;
    }

    public int[][] multiply(int[][] a, int[][] b) {
        int[][] c = new int[2][2];
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) {
                c[i][j] = a[i][0] * b[0][j] + a[i][1] * b[1][j];
            }
        }
        return c;
    }
}
```

Figure 10 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.

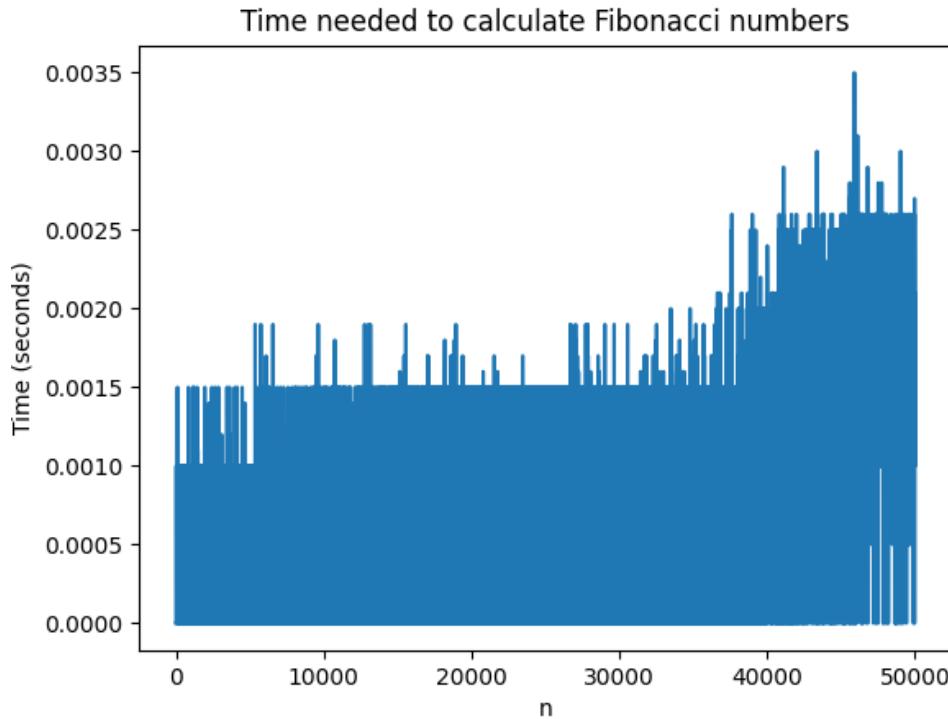


Figure 11 Matrix Method Fibonacci graph

Matrix Power Method: Method:

This is the most mathematical solution, and it requires some basic understanding of matrix multiplication. It is important to use exponentiation by squaring with this algorithm, because otherwise it degenerates into the dynamic programming algorithm. This algorithm takes $\Theta(1)$ space and $\Theta(\log n)$ operations. (Note: We are counting the number of bigint arithmetic operations, not fixed-width word operations.)

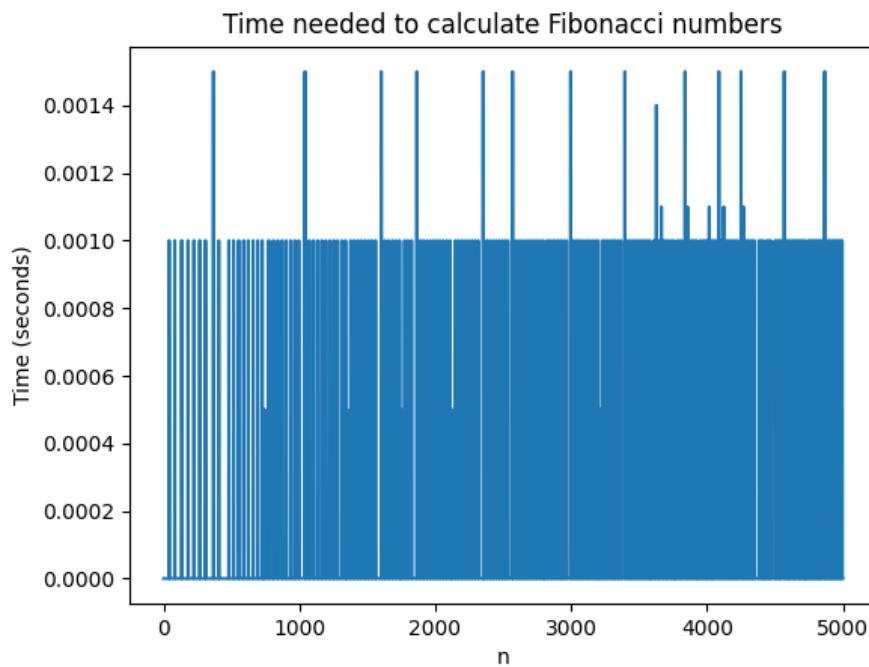


Figure 12 Matrix Method Fibonacci graph

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.

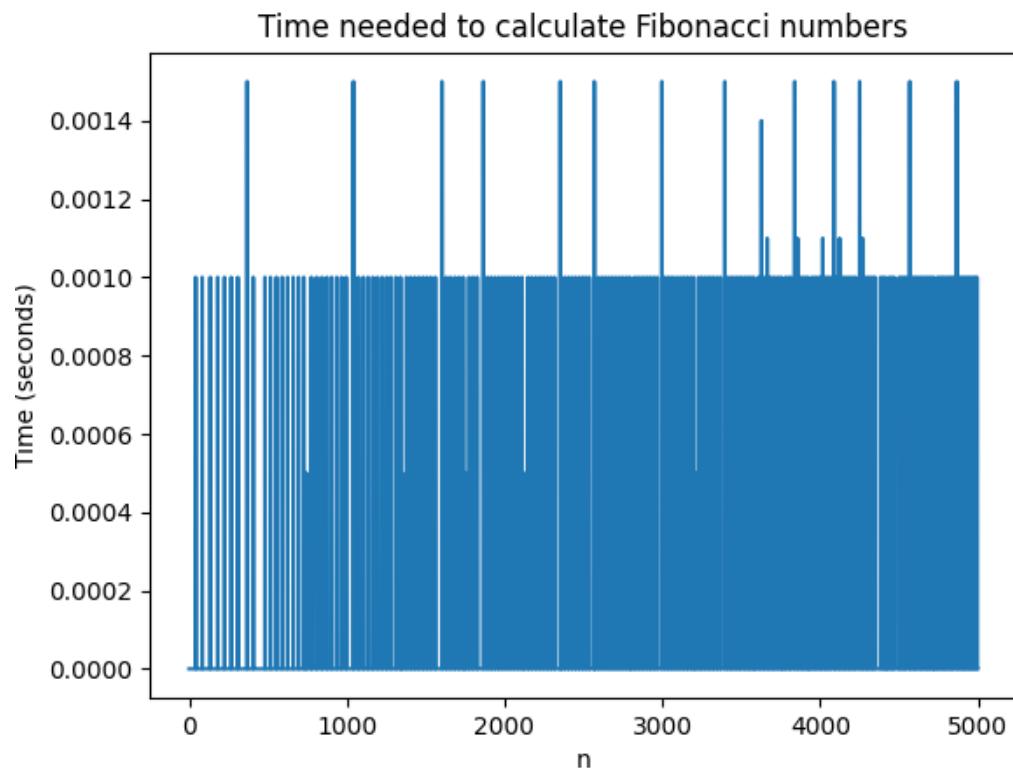
Algorithm Description:

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

Fibonacci (n) :

```
phi <- (1 + sqrt(5))
phi1 <- (1 - sqrt(5))
return pow(phi, n) - pow(phi1, n) / (pow(2, n) * sqrt(5))
```

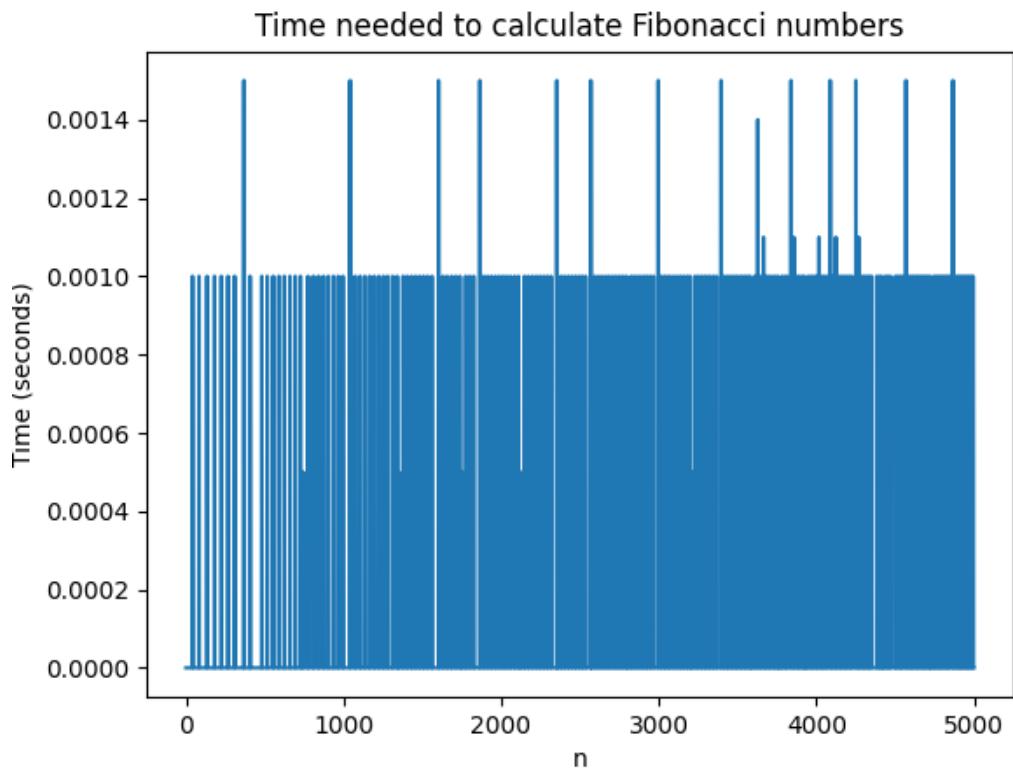
And as shown in its performance graph,



The Binet Formula Function is not accurate enough to be considered within the analysed limits and is recommended to be used for Fibonacci terms up to 80. At least in its naïve form in python, as further modification and change of language may extend its usability further.

Figure 13 Fibonacci Binet Formula Method in Python

And as shown in its performance graph,



The Binet Formula Function is not accurate enough to be considered within the analysed limits and is recommended to be used for Fibonacci terms up to 80. At least in its naïve form in python, as further modification and change of language may extend its usability further.

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.

