

**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-213

Iațco Sorin

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
Dynamic Programming Method:.....	6
Matrix Power Method: .....	8
Binet Formula Method: .....	11
Space Optimisation Method:.....	13
O(logn) arithmetic Method:.....	15
<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 empirical analysis is another way to study the complexity of an algorithm, instead of mathematical analysis. Empirical analysis can be used to gain initial insights into the complexity class of an algorithm, compare the efficiency of different algorithms or implementations, and assess the efficiency on a specific computer.

The process of empirical analysis involves defining the purpose of the analysis, choosing an efficiency metric, defining the properties of the input data, implementing the algorithm in code, generating input data, running the program, and analyzing the results.

The choice of efficiency metric depends on the goal of the analysis, for example, the number of operations for complexity class estimation, or execution time for implementation analysis.

The results are recorded and either summarized with statistics or plotted in a graph of problem size versus efficiency measure.

The Fibonacci sequence is a well-known sequence of numbers, and there are various algorithms for determining the n-th term of the sequence. The choice of algorithm depends on the desired trade-off between computational time and accuracy, as well as the specific requirements of the problem.

**Brute Force Approach:** The brute force approach involves calculating each term of the sequence sequentially until the n-th term is reached. This is the simplest and most straightforward approach, but it can be computationally expensive for large values of n.

**Recursive Approach:** The recursive approach involves defining the n-th term of the sequence in terms of the previous two terms and recursively computing each term until the n-th term is reached. This approach is more efficient than the brute force approach but can still be slow for large values of n due to repeated calculations.

**Dynamic Programming Approach:** The dynamic programming approach involves storing previously calculated terms in a table to avoid recalculations. This approach is much more efficient than the recursive approach and is widely used in practice.

**Matrix Approach:** The matrix approach involves using matrix multiplication to calculate the n-th term in  $O(\log n)$  time. This approach is the most efficient but can be challenging to implement and understand.

In conclusion, the choice of algorithm for determining the n-th term of the Fibonacci sequence depends on the specific requirements of the problem and the desired trade-off between computational time and accuracy. The dynamic programming approach is widely used in practice, but the matrix approach is the most efficient for large values of n.

**Introduction:**

The Fibonacci sequence is a series of numbers where each number is the result of adding the two previous numbers together. For example, the sequence begins with 0, 1, 1, 2, 3, 5, 8, 13, 21, and so on. Mathematically, it can be expressed as  $x_n = x_{n-1} + x_{n-2}$ . Some sources credit Leonardo Fibonacci as the first person to discover or create this sequence, while others dispute this claim. Some historians argue that the sequence was already present in ancient Sanskrit texts that used the Hindu-Arabic numeral system. However, it is generally accepted that Leonardo of Pisa, who lived in the 12th century and was later nicknamed Fibonacci, introduced the sequence to the Western world through his mathematical text, *Liber Abaci*, which was written for traders to do calculations. In this laboratory, we will be analyzing the performance of four naive algorithms through empirical analysis, as opposed to mathematical analysis.

**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 their naive 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 approach is a straightforward method for determining the  $n$ -th term in the Fibonacci sequence. This method involves computing the previous two terms in the sequence and then adding them to get the  $n$ -th term. However, this approach is considered inefficient due to the large amount of memory it uses and the repeated calculations of the same terms, leading to longer execution time. The method calls upon itself multiple times, which can lead to a doubling of the execution time in theory.

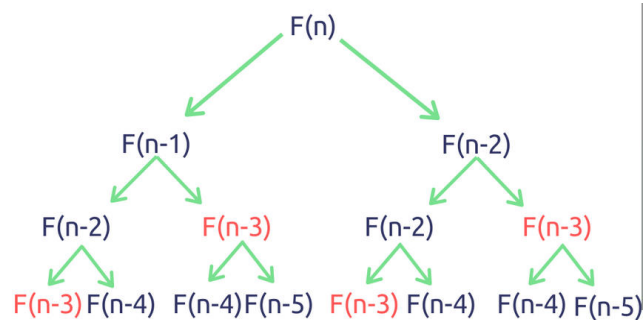


Figure 1 Fibonacci Recursion

*Algorithm Description:*

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:*

```
# Fibonacci series using recursion
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Figure 2 Fibonacci recursion in Python

*Results:*

After running the function for each n Fibonacci term saving the time for each n, we obtained the following results:

41	40	39	38	37	36
165580141	102334155	63245986	39088169	24157817	14930352
36	22	14	8	5	3

Figure 3 Results for last set of inputs

In Figure 3 is represented the table of results for the last set of inputs. The highest line(the name of the columns) denotes the Fibonacci n-th term for which the functions were run. The second row, we get the fibonacci number, and the third row is 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.

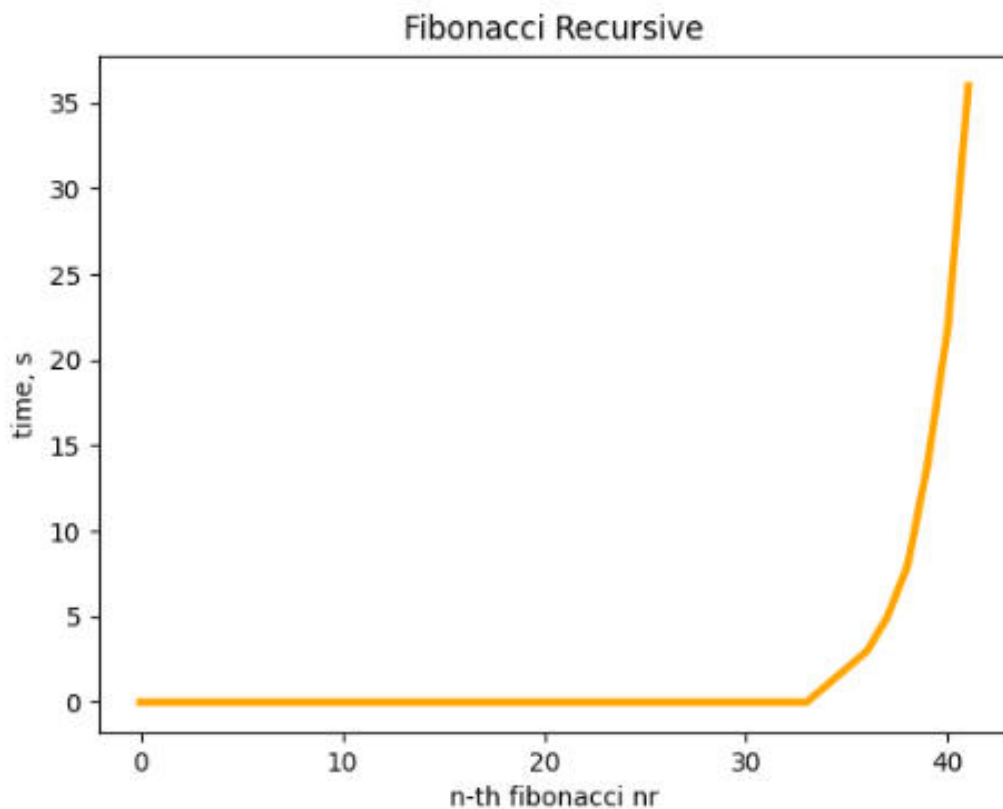


Figure 4 Graph of Recursive Fibonacci Function

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 41<sup>st</sup> term, leading us to deduce that the Time Complexity is exponential.  $T(2^{nn})$ .

#### Dynamic Programming Method:

The Dynamic Programming approach uses a similar method to the recursive approach to determine the n-th term in the Fibonacci sequence. Unlike the recursive method, it operates

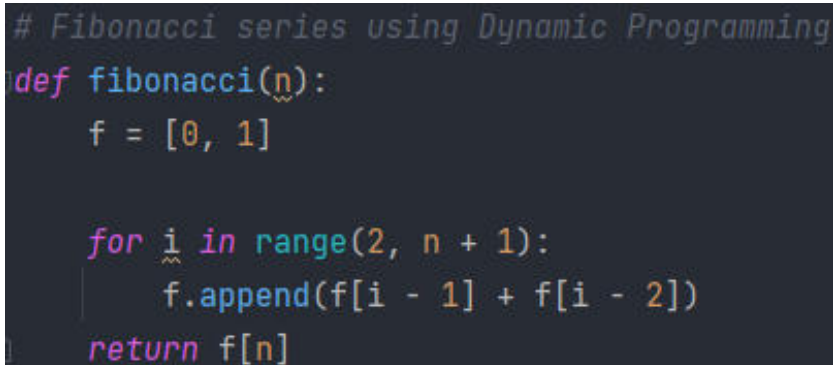
in a bottom-up manner and utilizes an array data structure to store previously computed terms, avoiding the need to recompute them. This eliminates the redundant calculations and memory usage, making the dynamic programming approach more efficient than the recursive method.

*Algorithm Description:*

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

```
Fibonacci(n) :
    Array A;
    A[0] ← 0;
    A[1] ← 1;
    for i ← 2 to n - 1 do
        A[i] ← A[i-1] + A[i-2];
    return A[n-1]
```

*Implementation:*



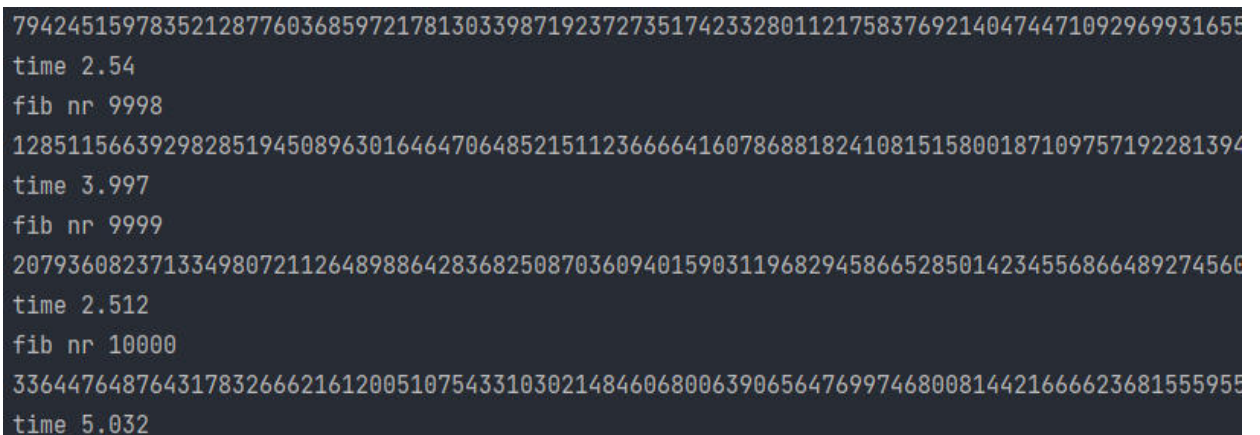
```
# Fibonacci series using Dynamic Programming
def fibonacci(n):
    f = [0, 1]

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

Figure 5 Fibonacci DP in Python

*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:



```
794245159783521287760368597217813033987192372735174233280112175837692140474471092969931655
time 2.54
fib nr 9998
128511566392982851945089630164647064852151123666641607868818241081515800187109757192281394
time 3.997
fib nr 9999
207936082371334980721126489886428368250870360940159031196829458665285014234556866489274568
time 2.512
fib nr 10000
336447648764317832666216120051075433103021484606800639065647699746800814421666623681555955
time 5.032
```

Figure 6 Fibonacci DP 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),

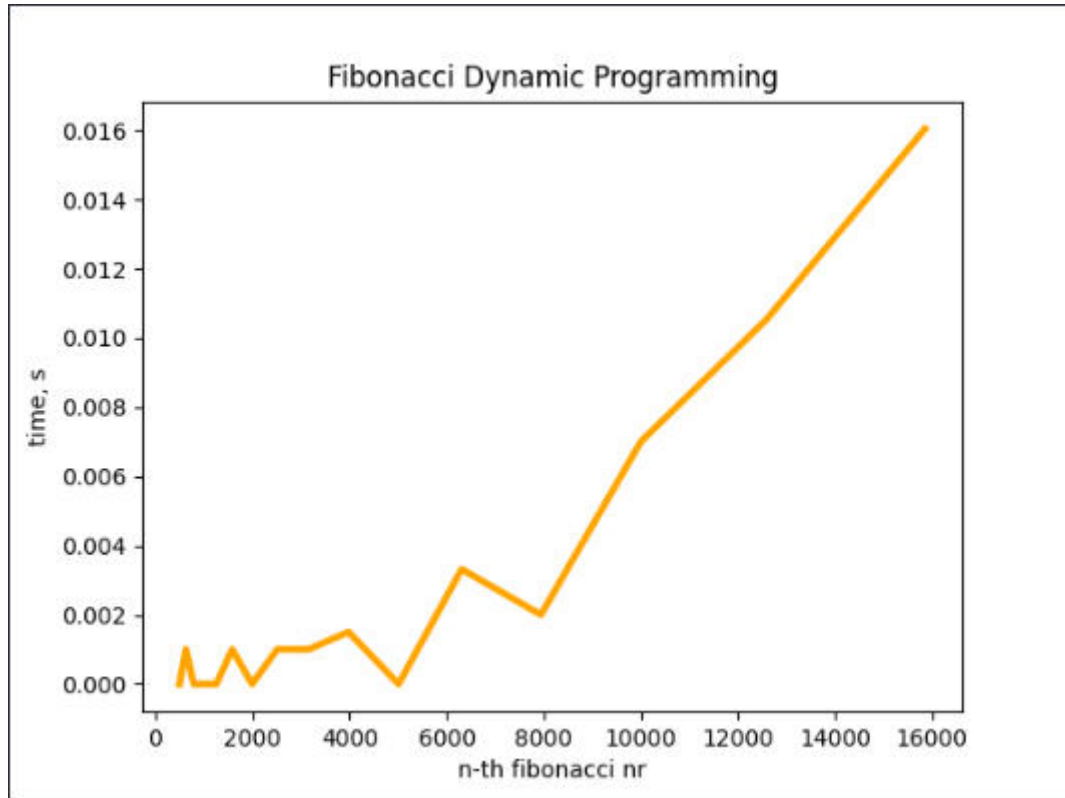


Figure 7 Fibonacci DP Graph

### 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:



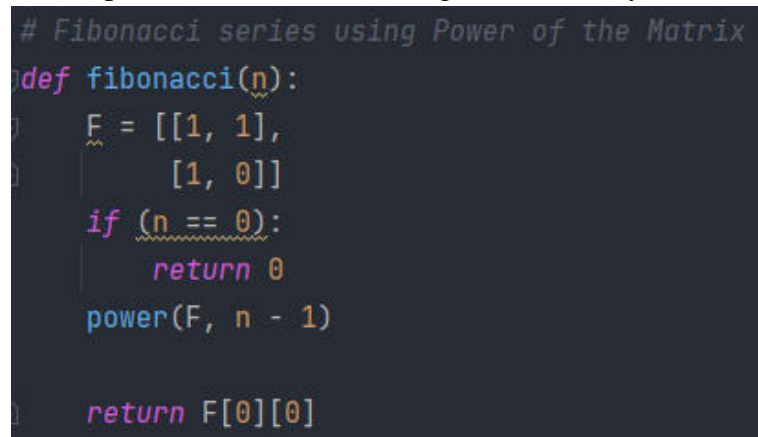
```

Fibonacci(n):
    F<- []
    vec <- [[0], [1]]
    Matrix <- [[0, 1],[1, 1]]
    F <-power(Matrix, n)
    F <- F * vec
    Return F[0][0]

```

### *Implementation:*

The implementation of the driving function in Python is as follows:



```

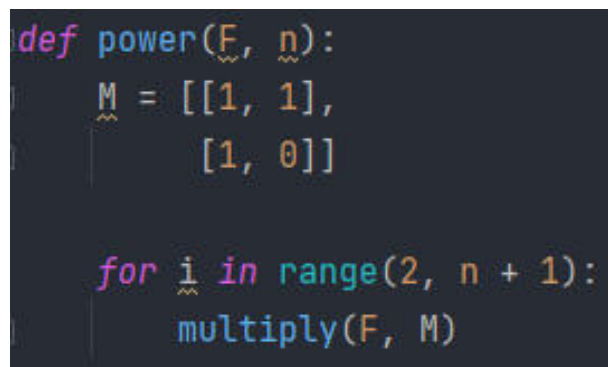
# Fibonacci series using Power of the Matrix
def fibonacci(n):
    F = [[1, 1],
         [1, 0]]
    if (n == 0):
        return 0
    power(F, n - 1)

    return F[0][0]

```

*Figure 8 Fibonacci Matrix Power Method in Python*

With additional miscellaneous functions:



```

def power(F, n):
    M = [[1, 1],
         [1, 0]]

    for i in range(2, n + 1):
        multiply(F, M)

```

*Figure 9 Power Function Python*

Where the power function (Figure 8) handles the part of raising the Matrix to the power n, while the multiplying function (Figure 9) handles the matrix multiplication with itself.

```
def multiply(F, M):
    x = (F[0][0] * M[0][0] +
         F[0][1] * M[1][0])
    y = (F[0][0] * M[0][1] +
         F[0][1] * M[1][1])
    z = (F[1][0] * M[0][0] +
         F[1][1] * M[1][0])
    w = (F[1][0] * M[0][1] +
         F[1][1] * M[1][1])

    F[0][0] = x
    F[0][1] = y
    F[1][0] = z
    F[1][1] = w
```

Figure 10 Multiply Function Python

### 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:

```
fib nr 9999
20793608237133498072112648988642836825087036094015903
time 21.584
fib nr 10000
33644764876431783266621612005107543310302148460680063
time 21.505
```

Figure 11 Matrix Method Fibonacci 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 of the graph indicating a pretty solid  $T(n)$  time complexity.

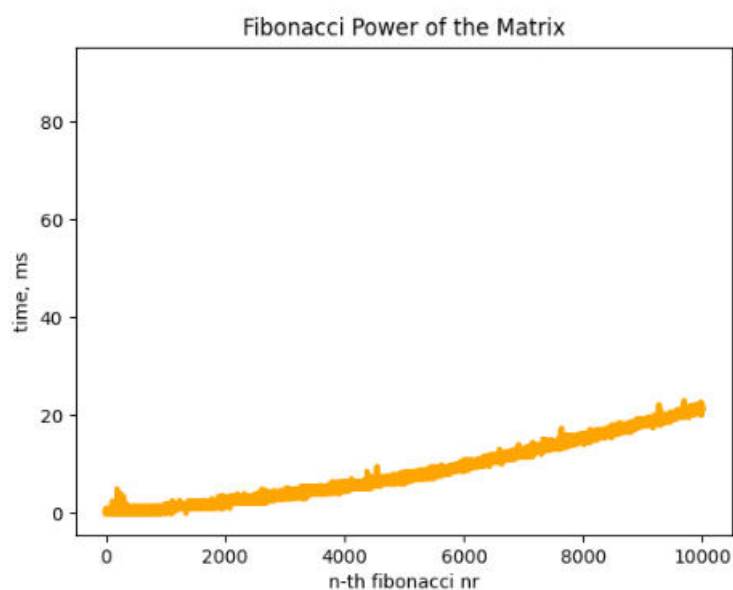


Figure 12 Matrix Method Fibonacci graph

### 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))
```

#### Implementation:

The implementation of the function in Python is as follows, with some alterations that would increase the number of terms that could be obtain through it:

```
# Fibonacci series using Binet
def fibonacci(n):
    decimal.getcontext().prec = 10000

    root_5 = decimal.Decimal(5).sqrt()
    phi = ((1 + root_5) / 2)

    a = ((phi ** n) - ((-phi) ** -n)) / root_5

    return round(a)
```

Figure 13 Fibonacci Binet Formula Method in Python

#### Results:

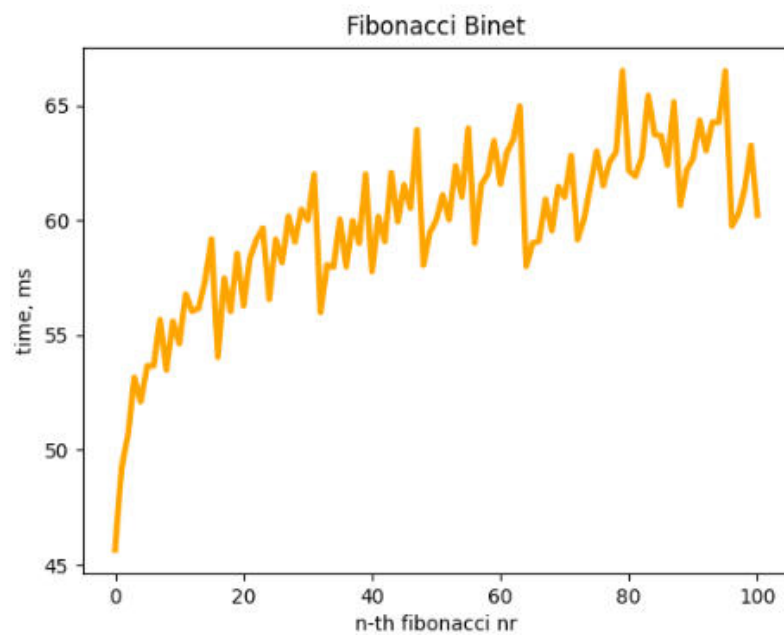
Although the most performant with its time, as shown in the table of results, in row [1],

```
fib nr 100
354224848179261915075
time 60.224
```

Figure 14 Fibonacci Binet Formula Method results

And as shown in its performance graph,

---



*Figure 15 Fibonacci Binet formula Method*

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.

### Space Optimisation Formula Method:

The space optimization method of Fibonacci refers to a technique used to calculate the nth Fibonacci number using a minimum amount of memory space. The traditional method of calculating Fibonacci numbers involves storing all intermediate values in an array or list, which can be memory-intensive. The space optimization method reduces the amount of memory required by using only two variables to store the current and previous values, effectively reducing the memory usage.

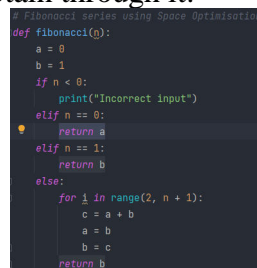
#### Algorithm Description:

The set of operation for the Space Optimisation Method can be described in pseudocode as follows:

```
Fibonacci(n):  
    a = 0  
    b = 1  
    if n == 0:  
        return a ...
```

#### Implementation:

The implementation of the function in Python is as follows, with some alterations that would increase the number of terms that could be obtain through it:

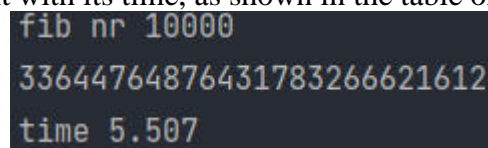
A screenshot of a code editor showing a Python function named 'fibonacci(n)'. The code initializes 'a' to 0 and 'b' to 1. It checks if 'n' is less than 0, printing an error message. If 'n' is 0, it returns 'a'. If 'n' is 1, it returns 'b'. Otherwise, it enters a loop from 1 to 'n', updating 'c' to 'a + b', then 'a' to 'b', and 'b' to 'c'. Finally, it returns 'b'.

```
# Fibonacci series using space optimization  
def fibonacci(n):  
    a = 0  
    b = 1  
    if n < 0:  
        print("Incorrect input")  
    elif n == 0:  
        return a  
    elif n == 1:  
        return b  
    else:  
        for i in range(2, n + 1):  
            c = a + b  
            a = b  
            b = c  
        return b
```

Figure 13 Fibonacci Space Optimisation Method in Python

#### Results:

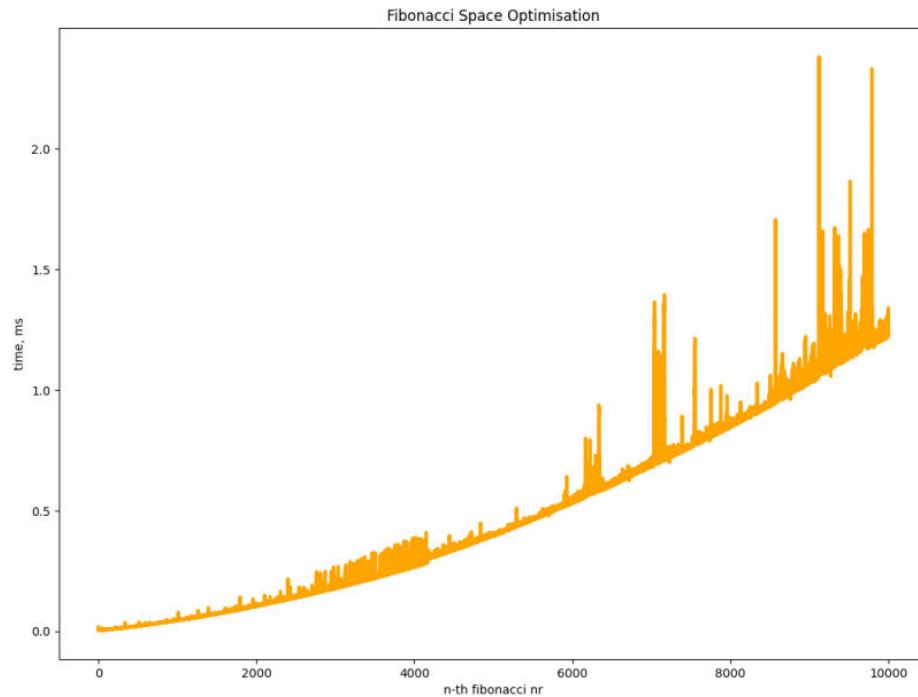
Although the most performant with its time, as shown in the table of results, in row [1],

A screenshot of a terminal window showing the output of the Fibonacci function for 'fib nr 10000'. The output displays the 10000th Fibonacci number, '336447648764317832666216120', and the execution time, 'time 5.507'.

```
fib nr 10000  
336447648764317832666216120  
time 5.507
```

Figure 14 Fibonacci Space Optimisation Method results

And as shown in its performance graph,



*Figure 15 Fibonacci Space Optimisation Method*

This technique is often used in competitive programming and coding interviews to test a candidate's ability to optimize for space and time.

### O(Log n) arithmetic operations Method:

The  $O(\log n)$  arithmetic operations method of Fibonacci refers to an efficient algorithm for calculating the  $n$ th Fibonacci number using only  $O(\log n)$  arithmetic operations. This method is based on matrix exponentiation and takes advantage of the fact that the Fibonacci sequence satisfies a certain matrix equation. By exponentiating a matrix to the  $n$ th power, the algorithm can calculate the  $n$ th Fibonacci number in  $O(\log n)$  time, which is much faster than the traditional linear-time algorithm.

#### Algorithm Description:

The set of operation for the  $O(\log n)$  arithmetic operations Method can be described in pseudocode as follows:

```
Fibonacci(n) :  
    if (n == 1 or n == 2) :  
        f[n] = 1  
    return (f[n])  
    ...
```

#### Implementation:

The implementation of the function in Python is as follows, with some alterations that would increase the number of terms that could be obtain through it:

```
# Fibonacci series using O(Log n) arithmetic operations  
MAX = 100000  
  
f = [0] * MAX  
  
def fibonacci(n):  
    if (n == 0):  
        return 0  
    if (n == 1 or n == 2):  
        f[n] = 1  
        return (f[n])  
  
    if (f[n]):  
        return f[n]  
  
    if (n & 1):  
        k = (n + 1) // 2  
    else:  
        k = n // 2  
  
    if ((n & 1)):  
        f[n] = (fibonacci(k) * fibonacci(k) + fibonacci(k - 1) * fibonacci(k - 1))  
    else:  
        f[n] = (2 * fibonacci(k - 1) + fibonacci(k)) * fibonacci(k)
```

Figure 13 Fibonacci  $O(\log n)$  arithmetic operations Method in Python

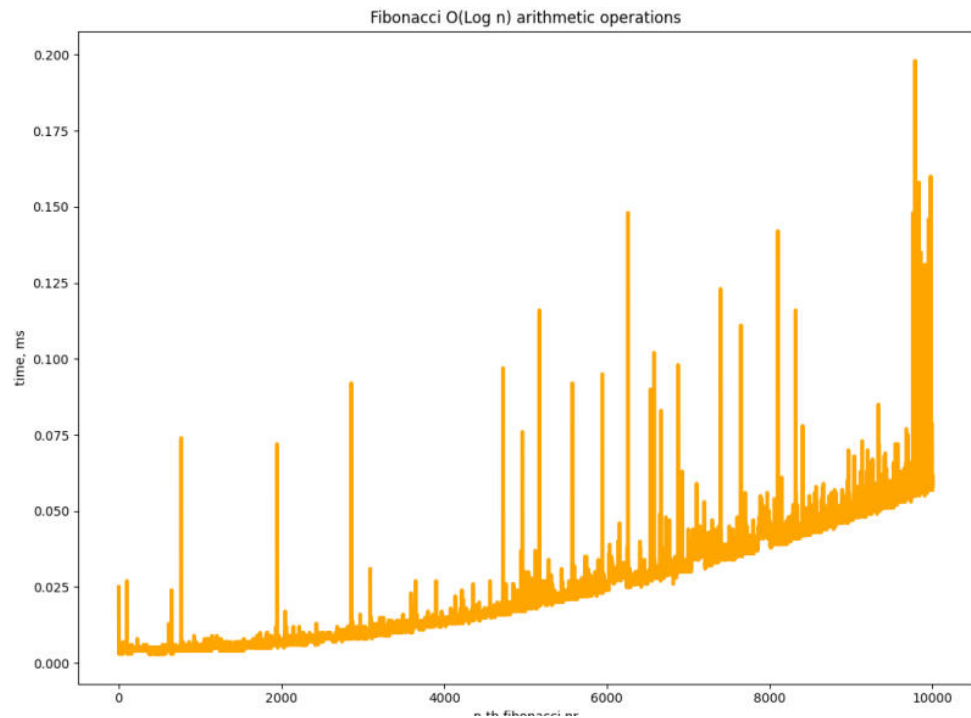
#### Results:

Although the most performant with its time, as shown in the table of results, in row [1],

```
fib nr 9999  
20793608237133498072  
time 2.003
```

Figure 14 Fibonacci  $O(\log n)$  arithmetic operations Method results

And as shown in its performance graph,



*Figure 15 Fibonacci  $O(\log n)$  arithmetic operations Method*

This method is commonly used in computer science and engineering and is considered a highly efficient and elegant solution to the problem of calculating Fibonacci numbers.



## CONCLUSION

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.

The Matrix Power Method is based on the observation that the Fibonacci sequence satisfies a particular matrix equation, and that exponentiating a matrix to the  $n$ th power results in a matrix whose elements contain the  $n$ th Fibonacci number.

The Space Optimisation Method reduces the amount of memory required by using only two variables to store the current and previous values, effectively reducing the memory usage.

The  $O(\log n)$  arithmetic is based on matrix exponentiation and takes advantage of the fact that the Fibonacci sequence satisfies a certain matrix equation. By exponentiating a matrix to the  $n$ th power, the algorithm can calculate the  $n$ th Fibonacci number in  $O(\log n)$  time, which is much faster than the traditional linear-time algorithm.

**link to github repository: <https://github.com/Syn4z/AA-Labs/tree/main/Lab1>**