**Ministerul Educaţiei 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 Tudor Sclifos

Verified:

asist. univ. Fiștic Cristofor

Chişinău - 2023

**TABLE OF CONTENTS**

[ALGORITHM ANALYSIS 3](#_bookmark0)

[Objective 3](#_bookmark1)

[Tasks 3](#_bookmark2)

[Theoretical Notes: 3](#_bookmark3)

[Introduction: 4](#_bookmark4)

[Comparison Metric 4](#_bookmark5)

[Input Format: 4](#_bookmark6)

[IMPLEMENTATION 5](#_bookmark7)

[Recursive Method: 5](#_bookmark8)

[Dynamic Programming Method: 6](#_bookmark9)

[Matrix Power Method: 8](#_bookmark10)

[Binet Formula Method: 11](#_bookmark11)

[CONCLUSION 13](#_bookmark12)

**Objective**

# ALGORITHM ANALYSIS

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 naive 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 naive 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 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.

*Algorithm Description:*

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

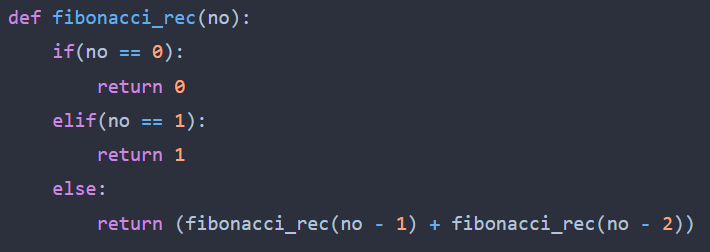
Fibonacci(n):

if n <= 1:

return n otherwise:

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

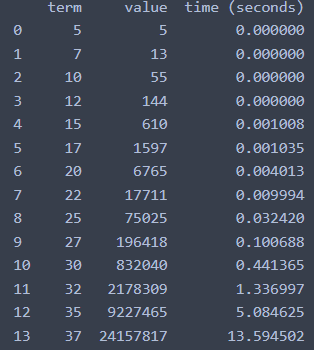
*Implementation:*



*Results:*

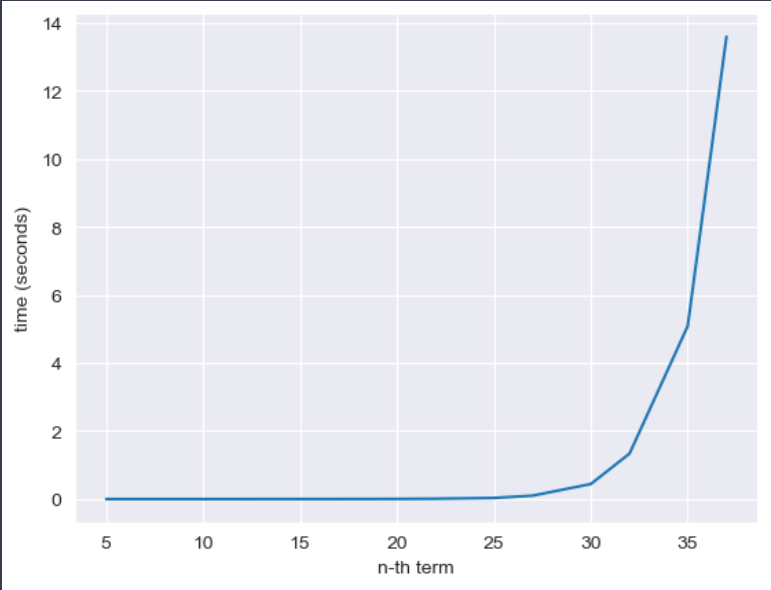
*Figure 2 Fibonacci recursion in Python*

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:



*Figure 3 Results for first set of inputs*

In Figure 3 is represented the table of results for the first set of inputs. First column shows the term, the second – the value and the 3rd – the execution time every value was computed.



*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 30th term, leading us to deduce that the Time Complexity is exponential. T(2𝑛).

## 

## Dynamic Programming Method:

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

*Algorithm Description:*

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

Fibonacci(n):

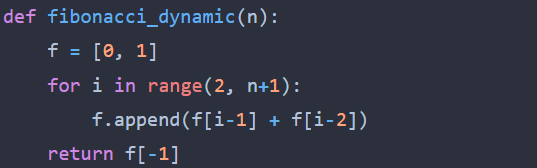
Array A; A[0]<-0;

A[1]<-1;

for i <- 2 to n – 1 do A[i]<-A[i-1]+A[i-2];

return A[n-1]

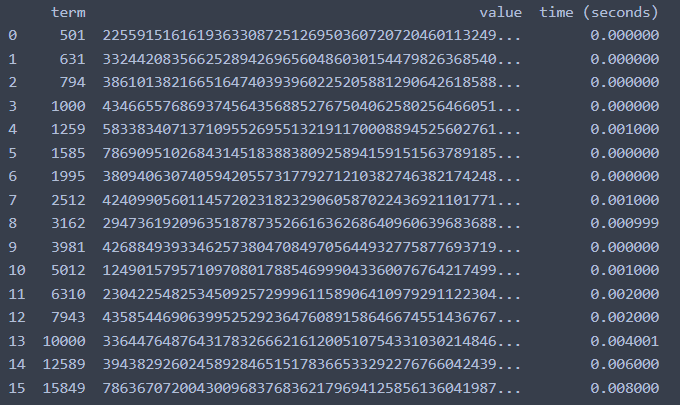
*Implementation:*



*Figure 5 Fibonacci DP in Python*

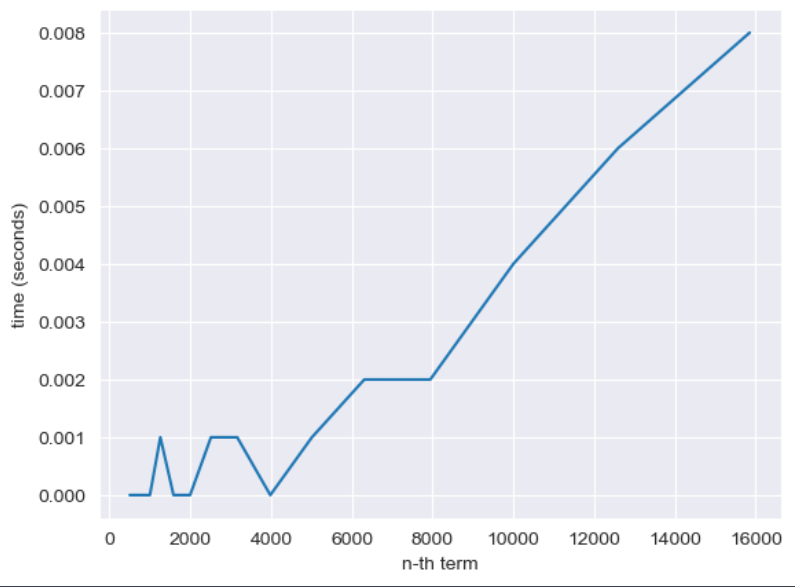
*Results:*

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



*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 naive Matrix (0 1) with itself.

1 1

*Algorithm Description:*

It is known that

0 1 𝑎 𝑏

( ) ( ) = ( )

1 1 𝑏 𝑎 + 𝑏

This property of Matrix multiplication can be used to represent

0 1 𝐹0 𝐹1

( ) ( ) = ( )

And similarly:

1 1

0 1 𝐹1

𝐹1

0 1

𝐹2

2 𝐹0

𝐹2

( ) (

) = (

) ( ) = ( )

1 1

Which turns into the general:

𝐹2

0 1 𝑛

1 1

𝐹0

𝐹1

𝐹𝑛

𝐹3

( ) ( ) = ( )

1 1 𝐹1

𝐹𝑛−1

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

Fibonacci(n):

F<- []

vec <- [[0], [1]]

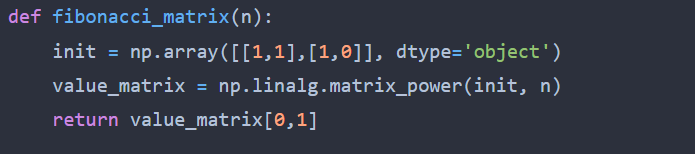
Matrix <- [[0, 1],[1, 1]]

F <-power(Matrix, n) F <- F \* vec

Return F[0][0]

*Implementation:*

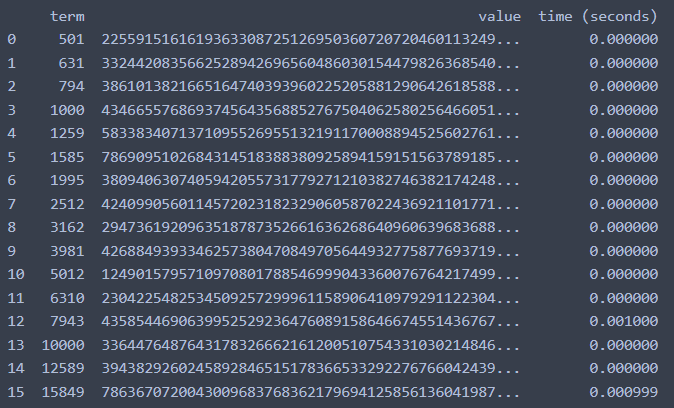
The implementation of the driving function in Python is as follows:



*Figure 8 Fibonacci Matrix Power Method in Python*

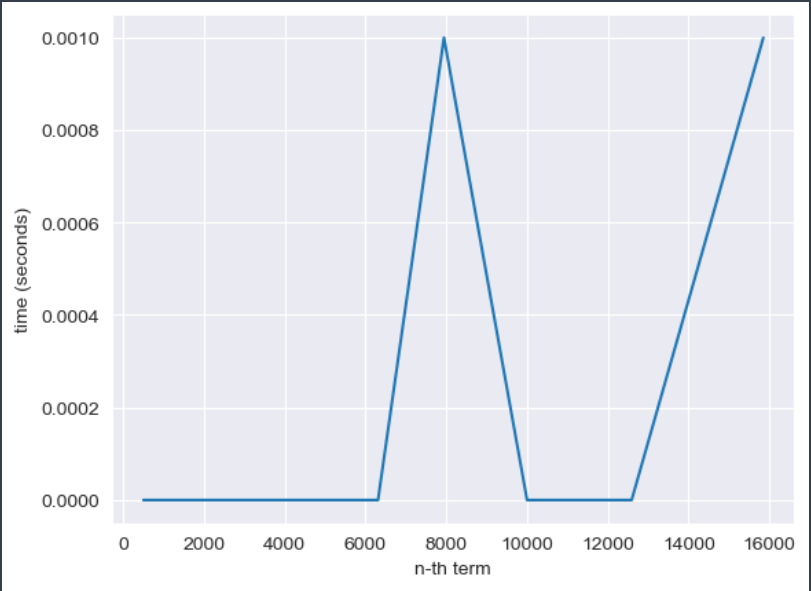
*Results:*

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



*Figure 9 Matrix Method Fibonacci Results*

With the naive 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.



*Figure 10 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):

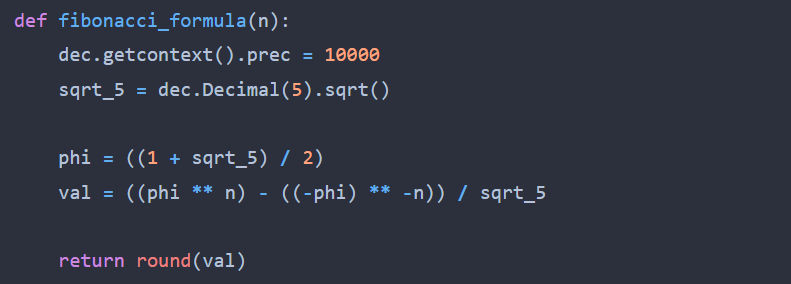
phi <- (1 + sqrt(5))

phi1 <-(1 – sqrt(5))

return pow(phi, n)- pow(phi1, n)/(pow(2, n)\*sqrt(5))

*Implementation:*

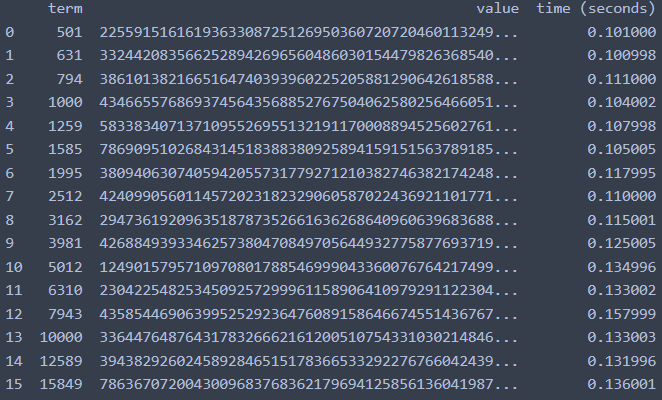
The implementation of the function in Python is as follows, with some alterations that would increase the number of terms that could be obtain through it:



*Figure 11 Fibonacci Binet Formula Method in Python*

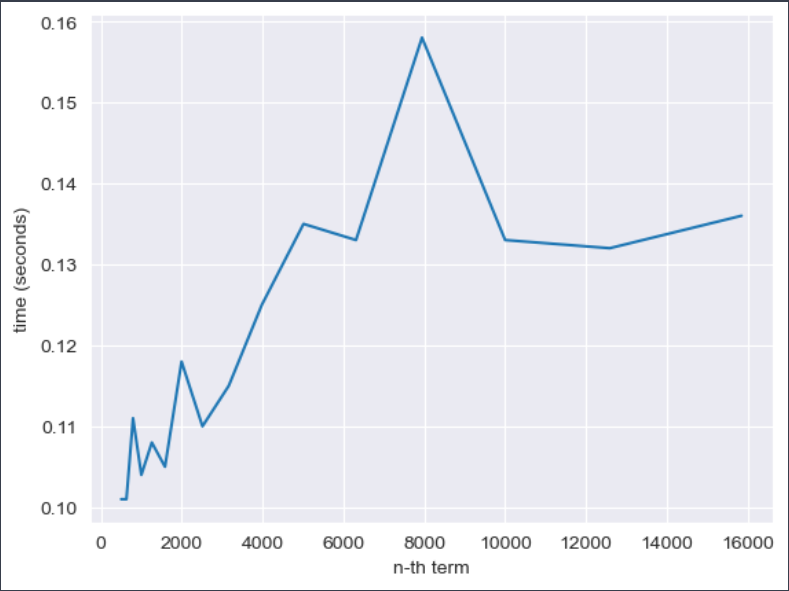
*Results*:

Although the most performant with its time, as shown in the table of results,



*Figure 12 Fibonacci Binet Formula Method results*

And as shown in its performance graph,



*Figure 13 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 naive form in python, as further modification and change of language may extend its usability further.

## Iteration Method:

The simplest way to calculate a Fibonacci number (n) is simply to start at the beginning and work forwards, iteratively. This solution calculates all previous values, giving it an exponential running time — larger numbers take increasingly longer to calculate.

*Algorithm Description:*

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

Fibonacci(n):

a<-0;

b<-1;

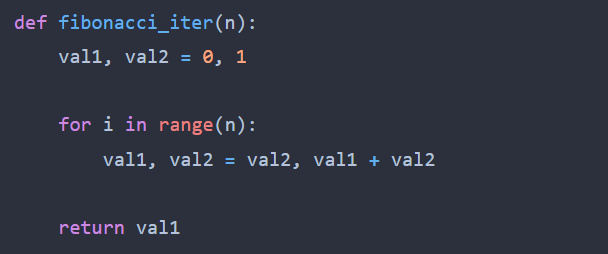
for i <- 2 to n – 1 do

a = b;

b = a+b;

return a

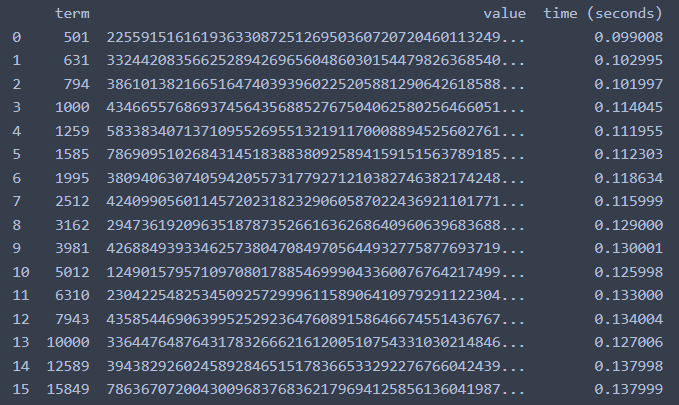
*Implementation:*



*Figure 14 Fibonacci Iterative in Python*

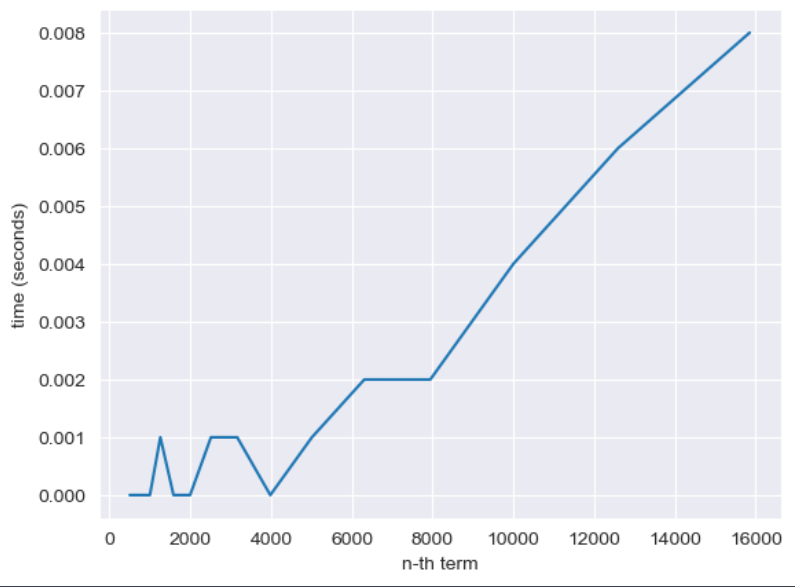
*Results:*

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



*Figure 15 Fibonacci Iterative Results*

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



*Figure 16 Fibonacci Iterative Graph*

## Fast Doubling Iterative Method:

We can implement iterative version of above method, by initializing array with two elements f = [F(0), F(1)] = [0, 1] and iteratively constructing F(n), on every step we will transform f into [F(2i), F(2i+1)] or [F(2i+1), F(2i+2)] , where i corresponds to the current value of i stored in f = [F(i), F(i+1)].

Approach:

Create array with two elements f = [0, 1] , which represents [F(0), F(1)] .

For finding F(n), iterate over binary representation of n from left to right, let kth bit from left be bk.

Iteratively apply the below steps for all bits in n .

Using bk we will decide whether to transform f = [F(i), F(i+1)] into [F(2i), F(2i+1)] or [F(2i+1), F(2i+2)]

*Algorithm Description:*

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

Fibonacci(n):

Array[] F;

bit\_no <- to\_array(to\_string(binary\_transform(n)));

F[0]<-0;

F[1]<-1;

for i <- 2 to length(bit\_no) do

f2i1<-F[1]^2+F[0]^2

f2i<-F[0]\*(2\*F[1]-F[0])

if bit\_no[i] == ‘0’ then

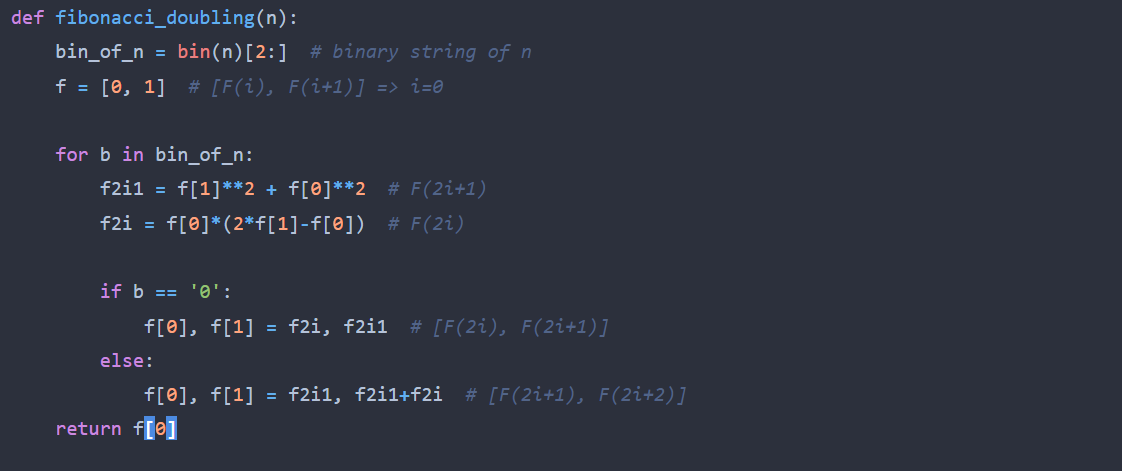
F[0]<-f2i; F[1]<-f2i1

else

F[0]<-f2i1; F[1]<-f2i1+f2i

return F[0]

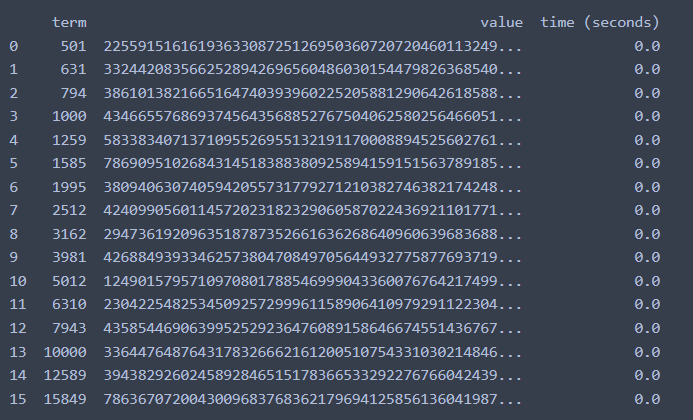
*Implementation:*



*Figure 17 Fibonacci Doubling in Python*

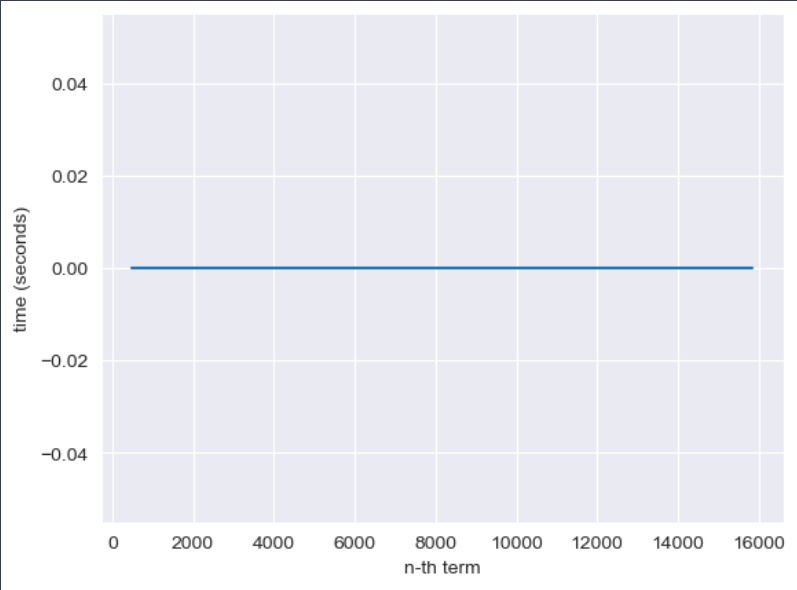
*Results:*

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



*Figure 18 Fibonacci Doubling Results*

With the Iterative Fast Doubling Method showing excellent results with a time complexity denoted in a corresponding graph of T(logn),



*Figure 19 Fibonacci Doubling Graph*

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

The last one, Iterative Fast Doubling Method is the best algorithm from the above, showing a logarithmical complexity. It works very fast, returning the results in no time. With additional optimizations it might become even faster.