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-243 Poiata Calin

Verified:
asist. univ. Fiștic Cristofor

Chișinău – 2026

Table of Contents

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 5 categories, Recursive Methods, Dynamic Programming Methods, Matrix Power Methods, Binet Formula Methods and Fast Doubling 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 5 naïve algorithms empirically.

Comparison Metric

The comparison metric for this laboratory work will be considered the time of execution of each algorithm ($O(n)$)

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 five algorithms will be implemented in their naïve form in C with the help of GMP library (GNU Multiple Precision Arithmetic Library) to being able to handle large numbers efficiently and analyzed empirically based on the time required for their completion. To get better results from execution times, the CPU times will be measured and furthermore the algorithms are run a few times and averaged. 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 and the speed 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 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.

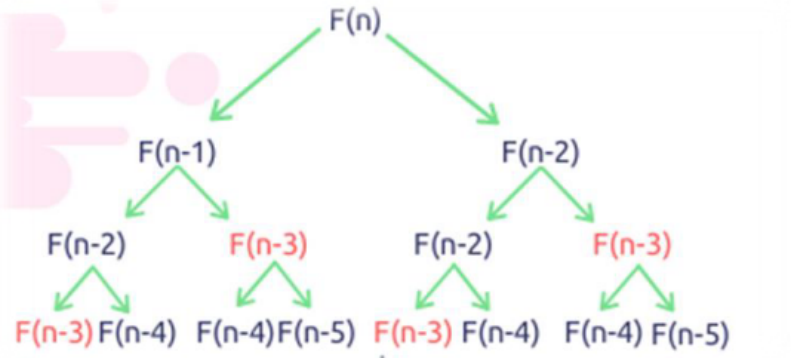

Figure 1 Fibonacci Recursion

Algorithm 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:

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

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:

n	Time (seconds)
5	0.000022
10	0.000015
15	0.000196
20	0.002357
22	0.003277
24	0.008572
26	0.022333
28	0.059306
30	0.154713
32	0.405016

Figure 2 Results for first set of inputs

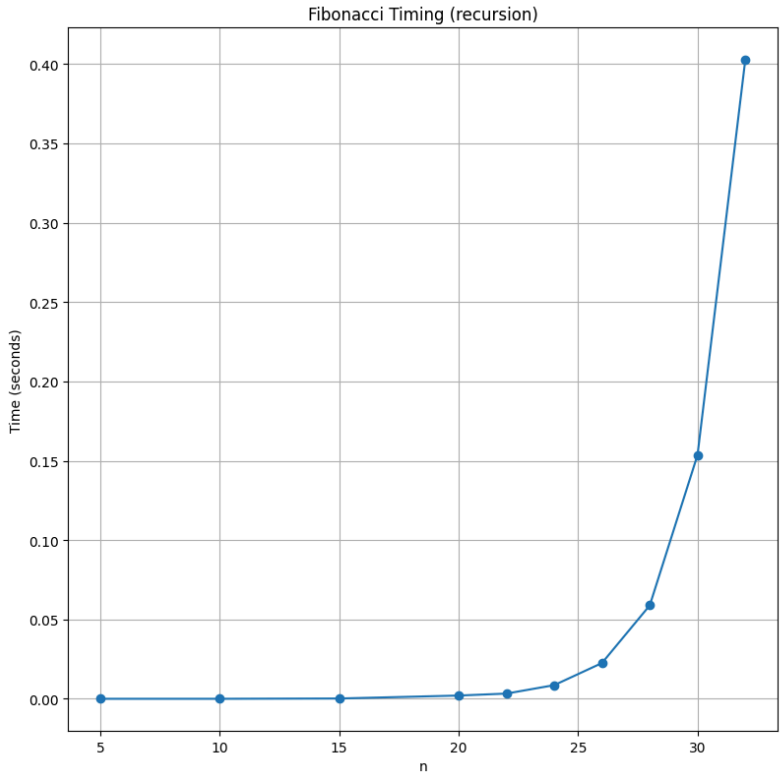

Figure 3 Graph of Recursive Fibonacci Function

In the graph from Figure 3 we can observe the growth of the time needed for the operations, we may easily see the spike in time complexity that happens after the 42nd term, leading us to deduce that the Time Complexity is exponential. $T(2^n)$.

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 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:

```
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:

```
def fib_dynamic(n):
    if n <= 1:
        return n

    dp = [0] * (n + 1)
    dp[1] = 1

    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]
```

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:

```
Computation Time Table
+-------+----------------+
|   n   | Time (seconds) |
+-------+----------------+
|   1   |    0.000007    |
|  1001 |    0.000375    |
|  2001 |    0.000770    |
|  3001 |    0.001132    |
|  4001 |    0.000953    |
|  5001 |    0.001280    |
|  6001 |    0.001677    |
|  7001 |    0.002110    |
|  8001 |    0.002705    |
|  9001 |    0.003177    |
+-------+----------------+
```

Figure 4 Results for DP Method

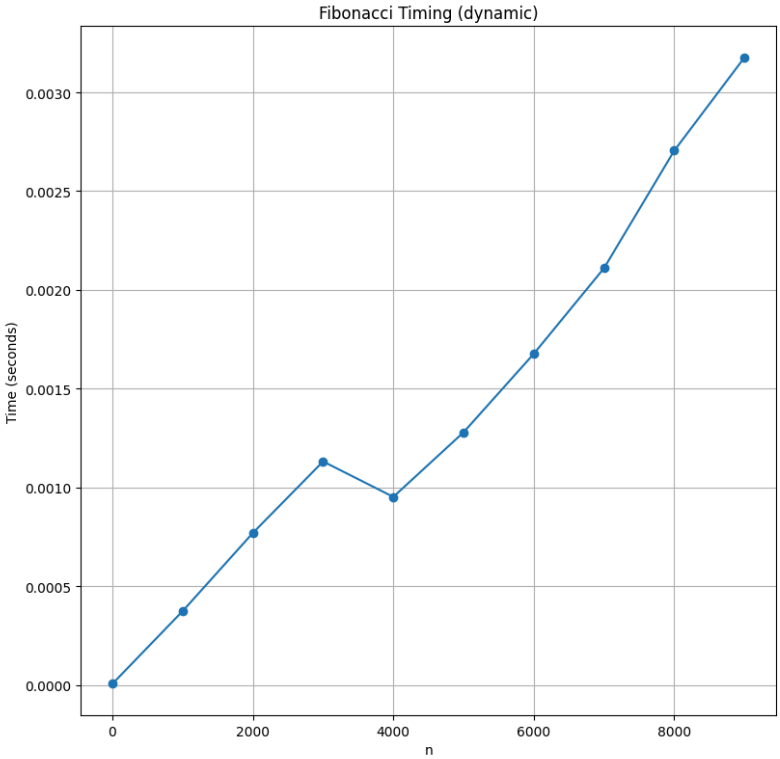

Figure 5 Graph for DP Method

From the resulting graph we can clearly observe that the time complexity of the dynamic programming is linear $O(n)$.

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 $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ 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:

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


def matrix_power(F, n):
    if n <= 1:
        return

    M = [[1, 1], [1, 0]]

    matrix_power(F, n // 2)
    multiply(F, F)

    if n % 2 != 0:
        multiply(F, M)


def fib_matrix(n):
    if n == 0:
        return 0

    F = [[1, 1], [1, 0]]
    matrix_power(F, n - 1)
    return F[0][0]
```

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:

n	Time (seconds)
0	0.000007
1000	0.000091
2000	0.000068
3000	0.000118
4000	0.000113
5000	0.000145
6000	0.000187
7000	0.000259
8000	0.000251
9000	0.000174
10000	0.000191
11000	0.000233
12000	0.000268
13000	0.000272
14000	0.000303
15000	0.000352
16000	0.000381

Figure 6 Results for MP Method

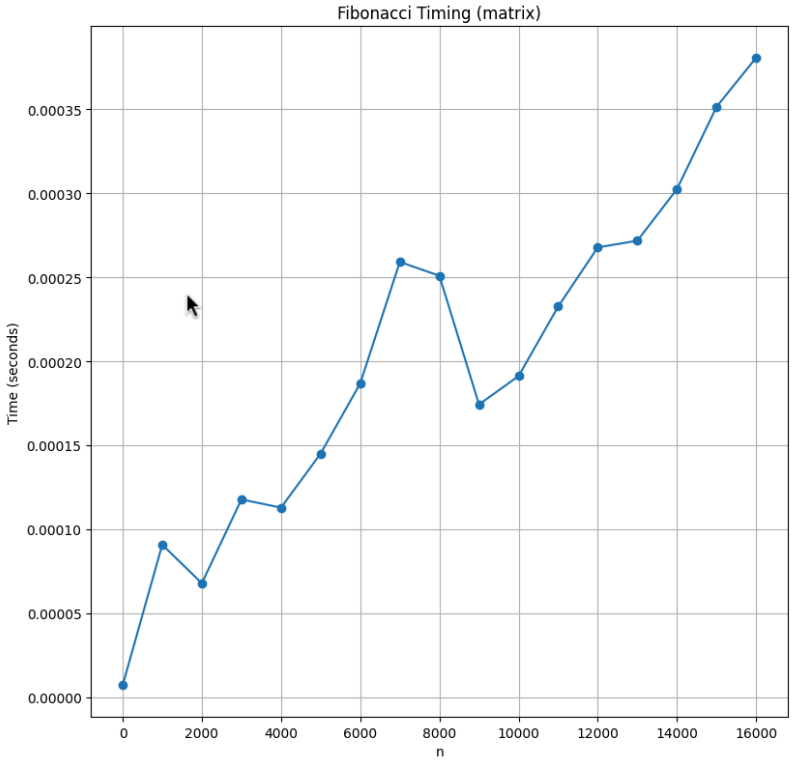

Figure 7 Graph for MP Method

With the naïve Matrix method, 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. This is because we implement the matrix multiplication in a naive way and if we would consider the optimized version of matrix multiplication, the time complexity would be reduced to $T(\log(n))$.

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 φ. However, due to its nature of requiring the usage of decimal numbers, at some point, the rounding error of uint64 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:

```
def fib_binet(n):
    if n == 0:
        return 0

    # high precision context
    ctx = Context(prec=100, rounding=ROUND_HALF_EVEN)

    # constants
    sqrt5 = ctx.sqrt(Decimal(5))
    phi = (Decimal(1) + sqrt5) / Decimal(2)
    psi = (Decimal(1) - sqrt5) / Decimal(2)

    # Binet formula with Decimal power
    a = ctx.power(phi, Decimal(n))
    b = ctx.power(psi, Decimal(n))

    result = (a - b) / sqrt5

    return int(result.to_integral_value())
```

Results:

n	Time (seconds)
0	0.000007
1000	0.000185
2000	0.000082
3000	0.000088
4000	0.000114
5000	0.000144
6000	0.000183
7000	0.000258
8000	0.000185
9000	0.000191
10000	0.000224
11000	0.000280
12000	0.000306
13000	0.000352
14000	0.000414
15000	0.000454
16000	0.000525

Figure 8 Results for Binet Method

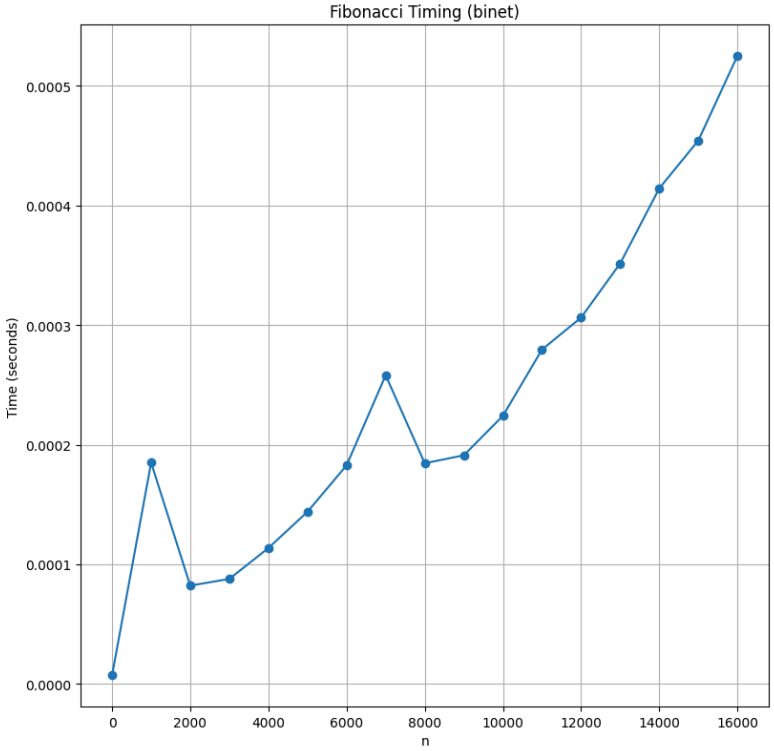

Figure 9 Graph for Binet Method

Although the most performant with its time, 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 C, this could have been improved by calculating the constants with a higher precision. The formula is in constant time as we can also see from the graph if we would ignore the inital CPU optimization spike.

Fast Doubling Method

The Fast Doubling Method is an improvement over the Matrix Method. And the following identities can be extracted from the Matrix Formula Method:

$$F(2k) = F(k)[2F(k+1) - F(k)]$$

$$F(2k+1) = F(k+1)^2 + F(k)^2$$

Algorithm Description:

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

```
Fibonacci(n):
    if n == 0:
        vec<-(0,1)
        return
    else:
        a, b <- Fibonacci(⌊n/2⌋)
        c <- a * (b * 2 -a)
        d <- a * a + b * b
        if n % 2 == 0:
            return (c, d)
        else:
            return (d, c + d)
```

Implementation:

```
def fib_fast_doubling(n):
    def helper(n):
        if n == 0:
            return (0, 1)
        else:
            a, b = helper(n // 2)
            c = a * (2*b - a)
            d = a*a + b*b
            if n % 2 == 0:
                return (c, d)
            else:
                return (d, c + d)

    return helper(n)[0]
```

Results:

Timing results for: fast

n	Time (seconds)
0	0.000015
1000	0.000041
2000	0.000025
3000	0.000037
4000	0.000034
5000	0.000048
6000	0.000057
7000	0.000095
8000	0.000078
9000	0.000108
10000	0.000114
11000	0.000168
12000	0.000147
13000	0.000157
14000	0.000175
15000	0.000197
16000	0.000209

Figure 10 Results for Fast Doubling Method

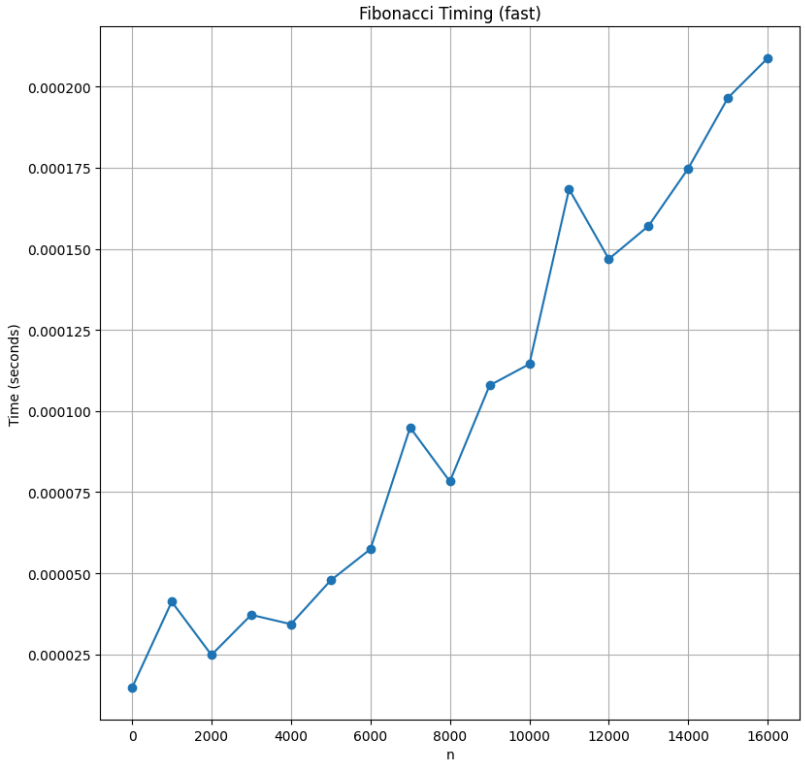

Figure 11 Graph for Fast Doubling Method

Fast Doubling Formula is one of the best formulas with its $O(\log(n))$ time complexity.

CONCLUSION

Through Empirical Analysis, within this paper, five 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 Fast Doubling Formula method, being the easiest to execute with a logarithmic time complexity, could be used when computing very large numbers of the series.

ANNEX

Github Repository