**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-233 Nicologlo Victoria

Verified:

asist. univ. Fiștic Cristofor

Chişinău - 2025

**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 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

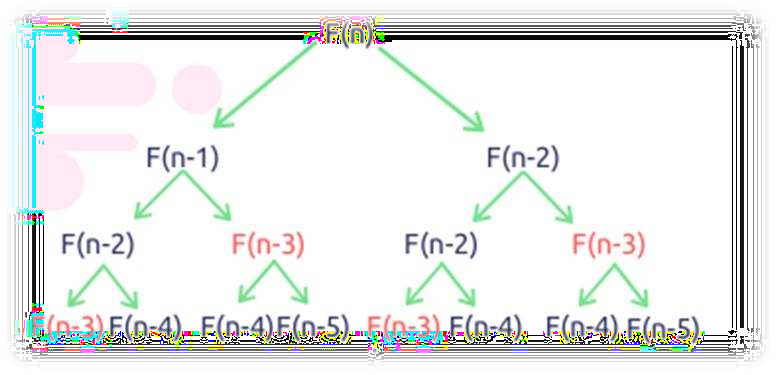
All four algorithms will be implemented in their naïve form in python an analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending 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 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:

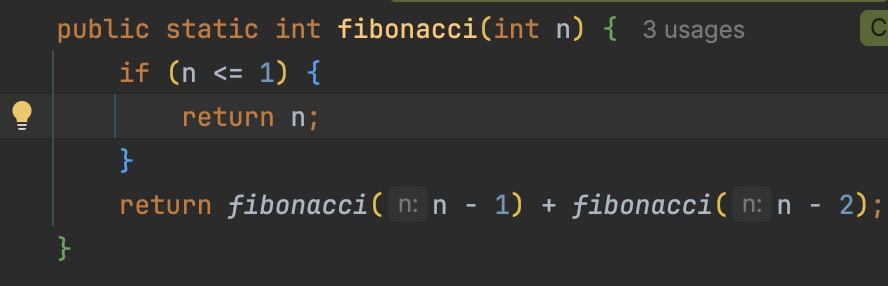
Fibonacci(n):

if n <= 1:

return n otherwise:

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

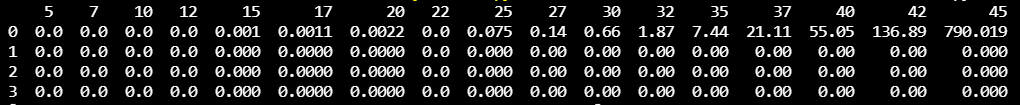
*Implementation:*

**

*Results:*

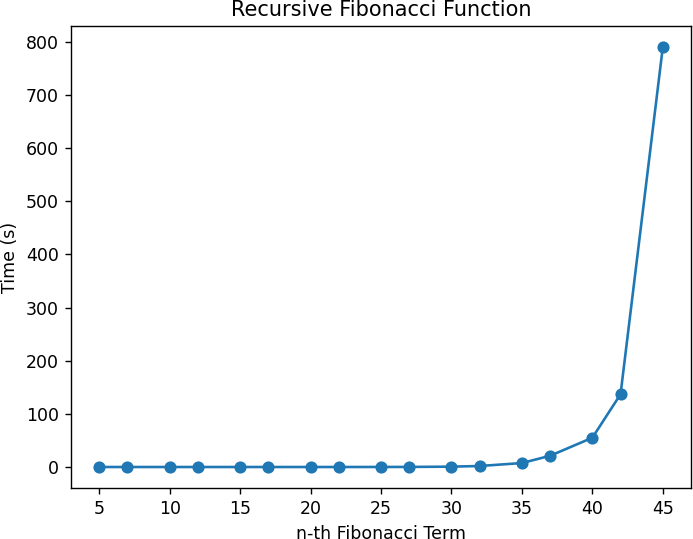
*Figure 2: Fibonacci recursion in Java*

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. The highest line(the name of the columns) denotes the Fibonacci n-th term for which the functions were run. Starting from the second row, we get 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 42nd 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 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):

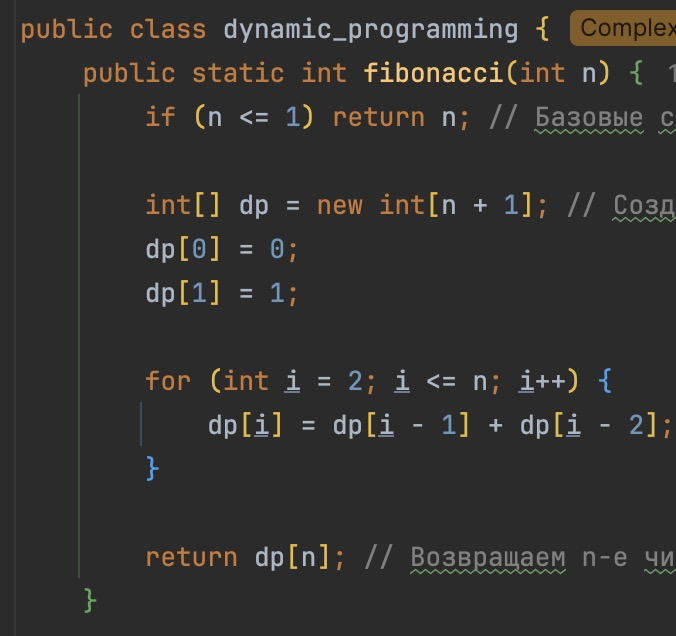
Array A; A[0]<-0;

A[1]<-1;

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

return A[n-1]

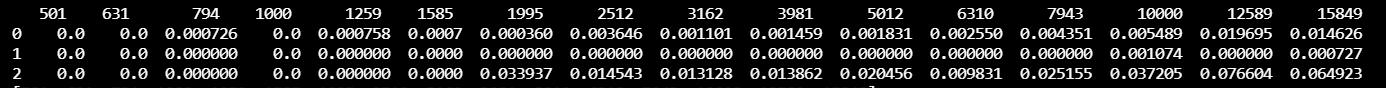
*Implementation:*

**

*Results:*

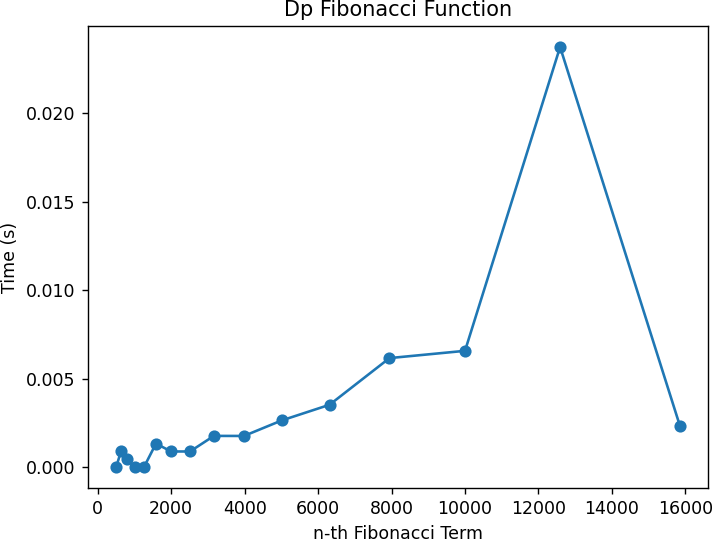
*Figure 5 Fibonacci DP in Java*

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),



## Matrix Power Method:

*Figure 7 Fibonacci DP Graph*

The Matrix Power method of determining the n-th Fibonacci number is based on, as expected, the

multiple multiplication of a naïve Matrix

*Algorithm Description:*

It is known that

0 1

( ) with itself.

1 1

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]]

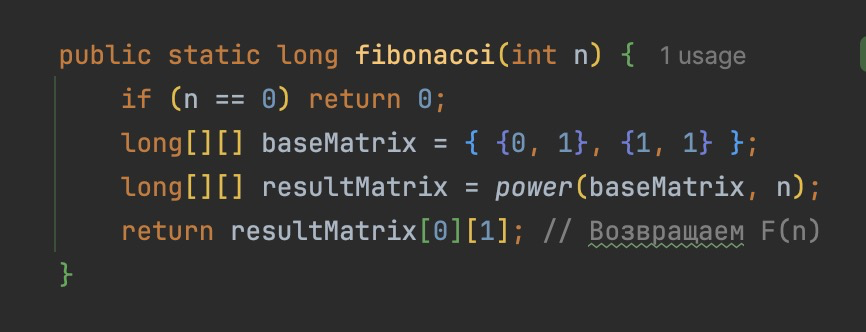
Matrix <- [[0, 1],[1, 1]]

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

Return F[0][0]

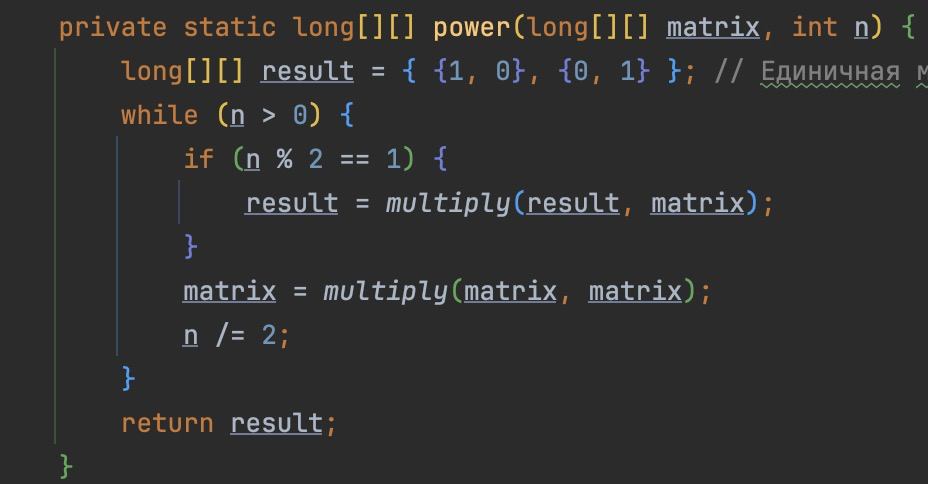
*Implementation:*

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



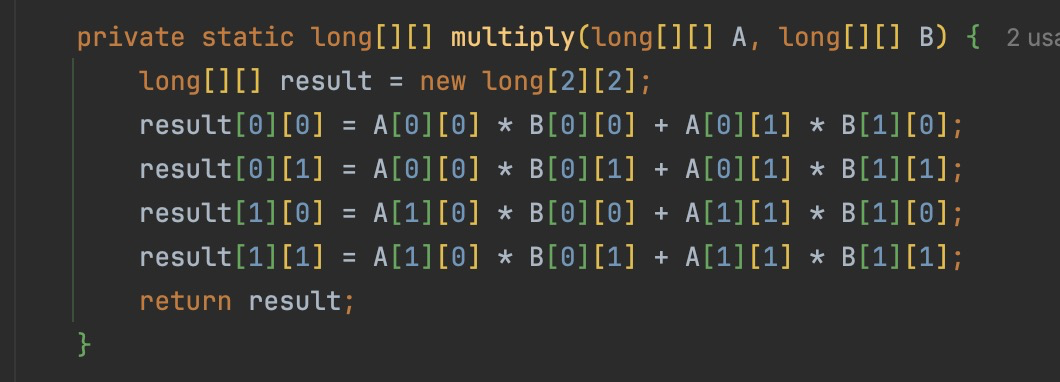
*Figure 8 Fibonacci Matrix Power Method in Java*

With additional miscellaneous functions:



*Figure 9 Power Function Java*

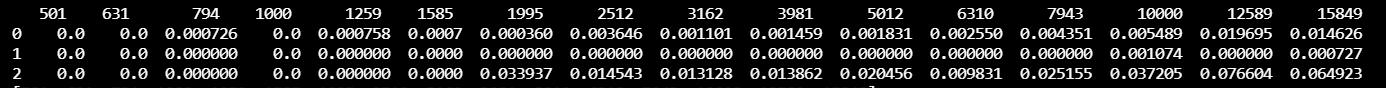
Where the power function (Figure 8) handles the part of raising the Matrix to the power n, while the multiplying function (Figure 9) handles the matrix multiplication with itself.



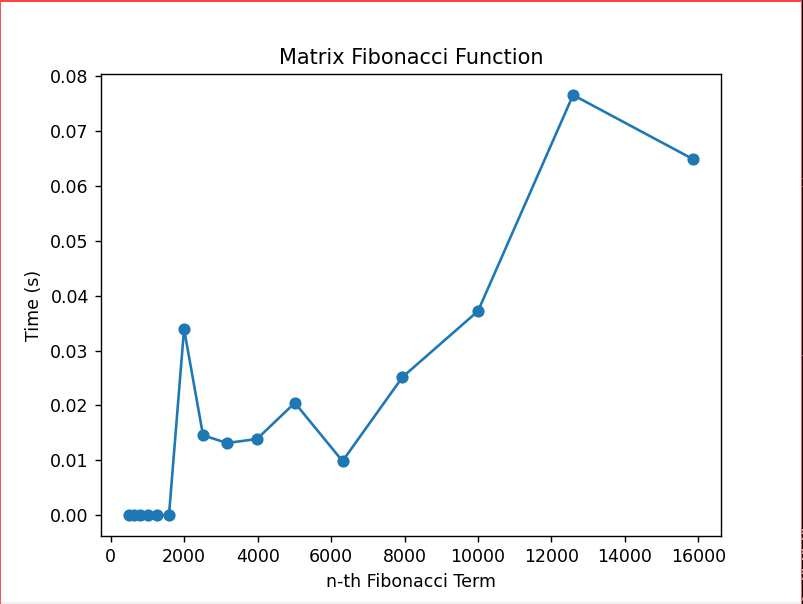
*Results:*

*Figure 10 Multiply Function Java*

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 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 f 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):

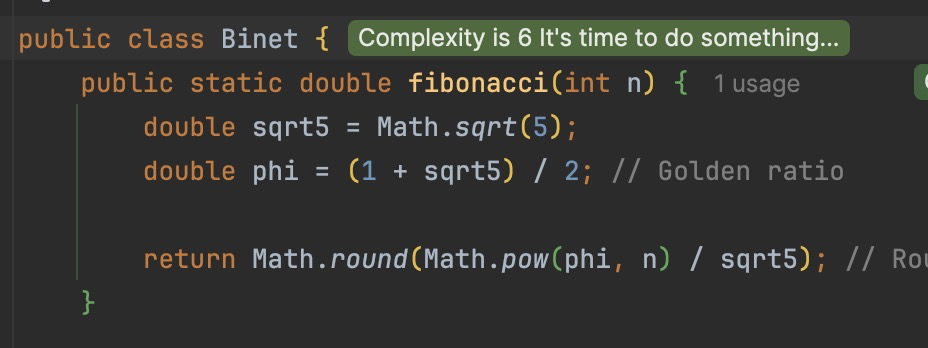
phi <- (1 + sqrt(5))

phi1 <-(1 – sqrt(5))

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

*Implementation:*

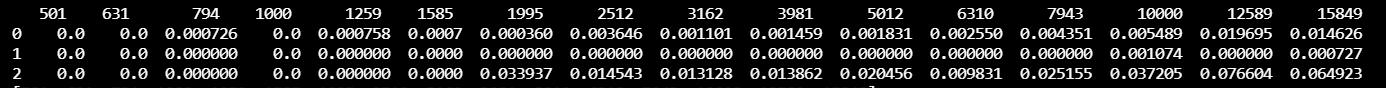
The implementation of the function in Java is as follows, with some alterations that would increase the number of terms that could be obtain through it:



*Results*:

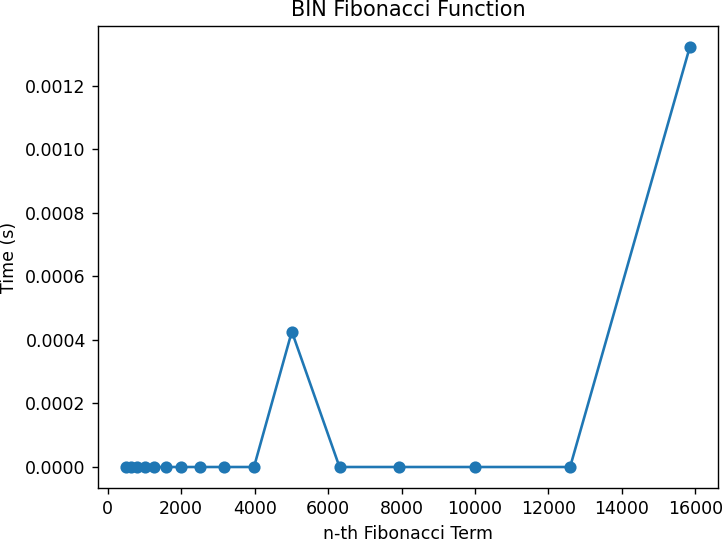
*Figure 13 Fibonacci Binet Formula Method in Java*

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



*Figure 14 Fibonacci Binet Formula Method results*

And as shown in its performance graph,

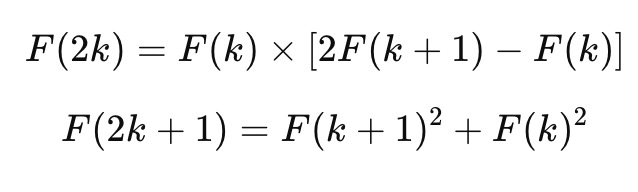


*Figure 15 Fibonacci Binet formula Method*

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 java, as further modification and change of language may extend its usability further. Time complexity: O(1).

**Fast Doubling Method**

The **Fast Doubling Method** is an efficient way to compute the **N-th Fibonacci number** using **divide and conquer** and **matrix exponentiation principles.** The Fast Doubling Fibonacci Algorithm is an optimized O(log n) method for computing Fibonacci numbers using divide and conquer principles. Instead of computing F(n-1) and F(n-2), it splits the problem in half by finding F(k) and F(k+1) for k = n/2 and then using mathematical recurrence relations to compute higher values efficiently:



*Figure 16 Formulas for fast doubling method*

This approach avoids redundant computations, making it significantly faster than naive recursion.

* The **time complexity is O(log n)** because at each step, n is divided by 2, resulting in **logarithmic depth recursion**.
* The **space complexity is O(log n) in the recursive version** due to the function call stack, but it can be **O(1) in an iterative implementation**.

*Algorithm Description:*

The set of operation for the Fast Doubling Method can be described in pseudocode as follows:

FUNCTION fastDoubling(n):

IF n == 0 THEN

RETURN (0, 1) // Base case: (F(0), F(1))

(Fk, Fk+1) ← fastDoubling(n // 2) // Recursively compute Fibonacci pair for k = n/2

F2k ← Fk × (2 × Fk+1 - Fk) // Formula: F(2k) = F(k) × [2F(k+1) - F(k)]

F2k+1 ← Fk+1 × Fk+1 + Fk × Fk // Formula: F(2k+1) = F(k+1)^2 + F(k)^2

IF n is even THEN

RETURN (F2k, F2k+1)

ELSE

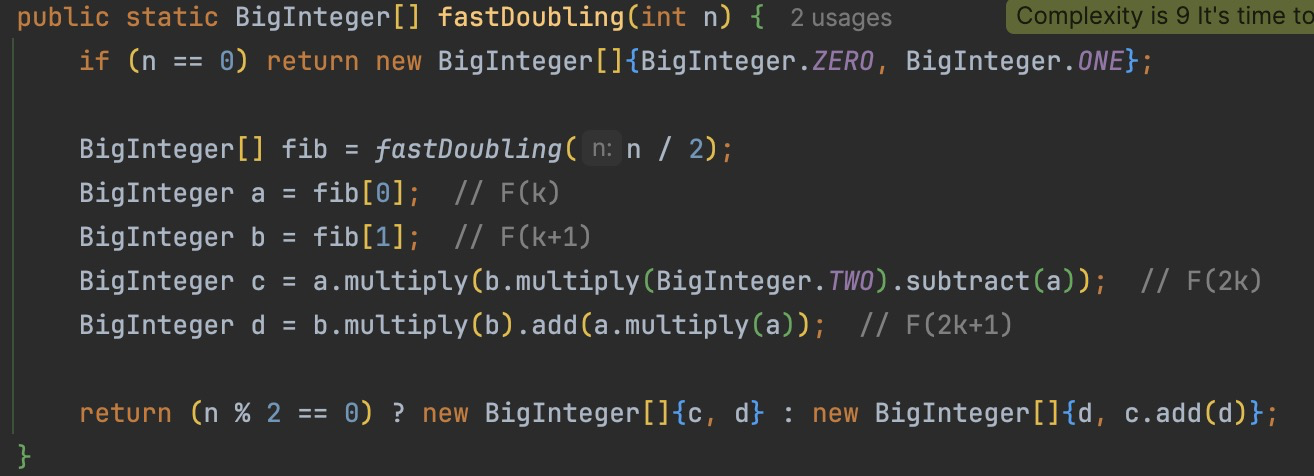
RETURN (F2k+1, F2k + F2k+1)

FUNCTION fibonacci(n):

RETURN fastDoubling(n)[0] // Return only F(n)

*Implementation:*

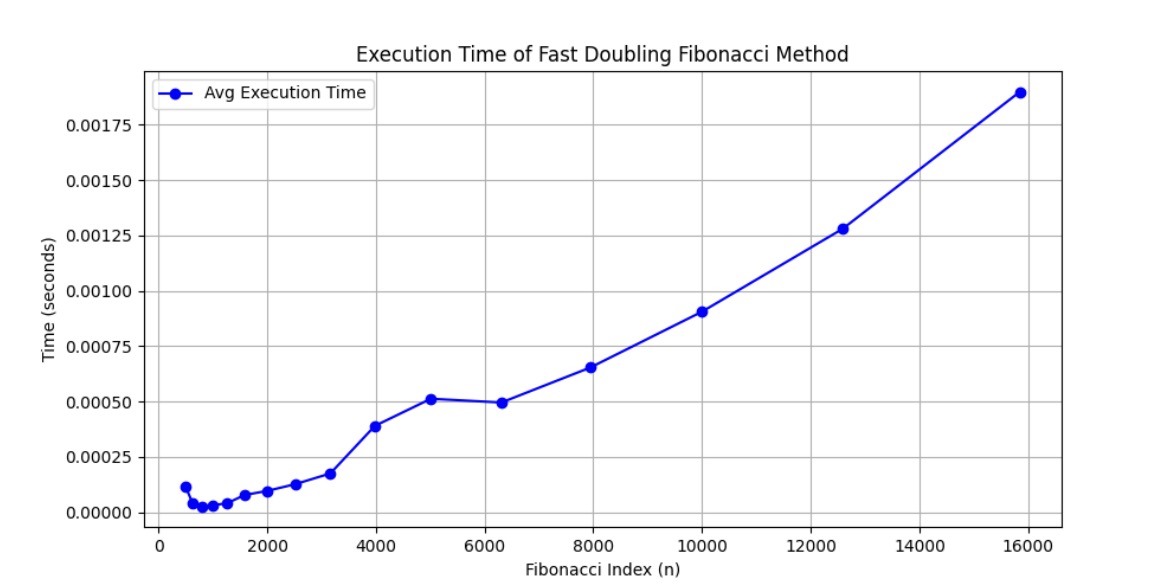
The implementation of the function in Java is as follows, with some alterations that would increase the number of terms that could be obtain through it:

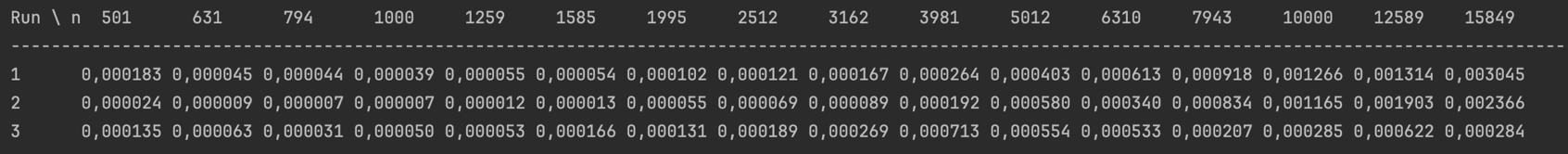


*Figure 17 fastDoubling Function Java*

*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 Fast Doubling Method Results*

*Figure 19 Fibonacci Binet formula Method*

The graph shows the execution time of the **Fast Doubling Fibonacci Method**, confirming its **O(log n) time complexity** as the runtime increases logarithmically with n. The curve remains relatively flat for small values but exhibits a steady growth pattern as n increases, demonstrating the efficiency of the method even for large Fibonacci indices.

**Iterative Fibonacci with HashMap Memoization**

The Iterative Fibonacci with HashMap Memoization algorithm computes the N-th Fibonacci number in O(n) time using a bottom-up dynamic programming approach, avoiding recursion to prevent StackOverflowError for large n. Instead of making recursive calls, it iterates from 0 to n, storing previously computed values in a HashMap for O(1) lookups, ensuring no redundant calculations. The algorithm maintains two variables (a and b) to track F(n-2) and F(n-1), updating them iteratively using the recurrence F(n) = F(n-1) + F(n-2). Its time complexity is O(n) as it processes n elements linearly, and space complexity is also O(n) due to HashMap storage. This method is faster than naive recursion (O(2ⁿ)), avoids deep recursion issues, and efficiently retrieves previously computed Fibonacci numbers, making it suitable for moderate-sized n where memory is available but recursion is undesirable.

*Algorithm Description:*

The set of operation for the Fast Doubling Method can be described in pseudocode as follows:

FUNCTION fibonacci(n):

IF n <= 1 THEN

RETURN n

CREATE a HashMap memo

SET a ← 0, b ← 1

FOR i FROM 2 TO n:

temp ← a + b

a ← b

b ← temp

memo[i] ← b // Store Fibonacci number in HashMap

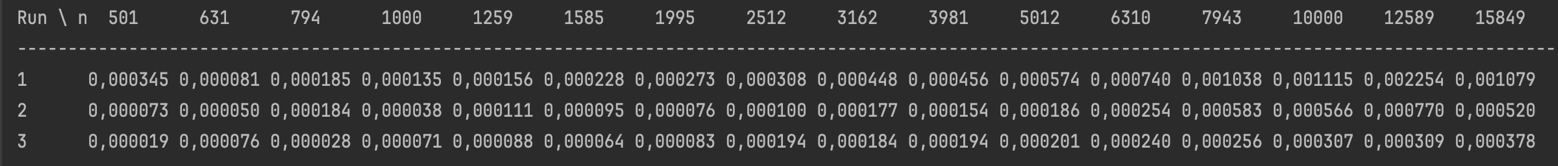
RETURN b // F(n) is stored in b

*Implementation:*

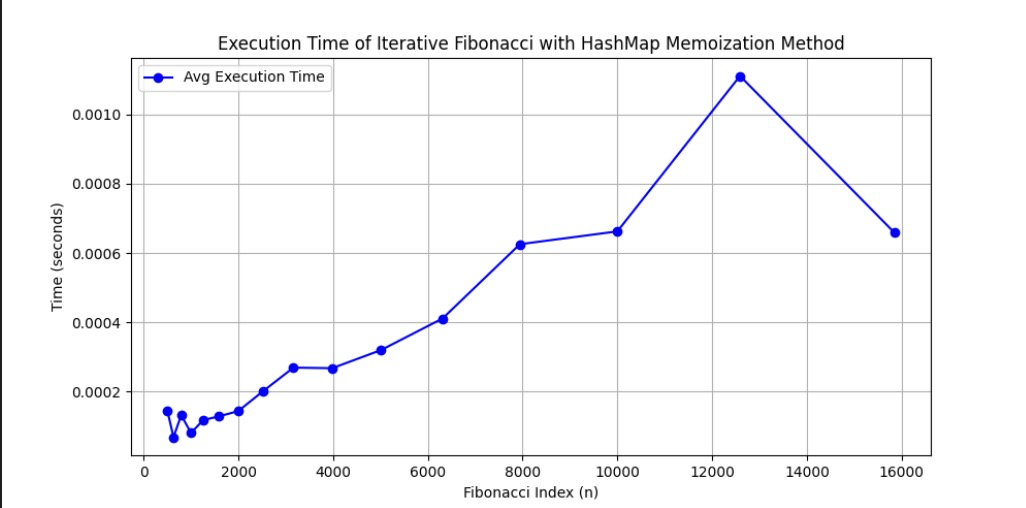
The implementation of the function in Java is as follows, with some alterations that would increase the number of terms that could be obtain through it:

*Figure 20 Iterative Fibonacci with HashMap Memoization Method*

*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:

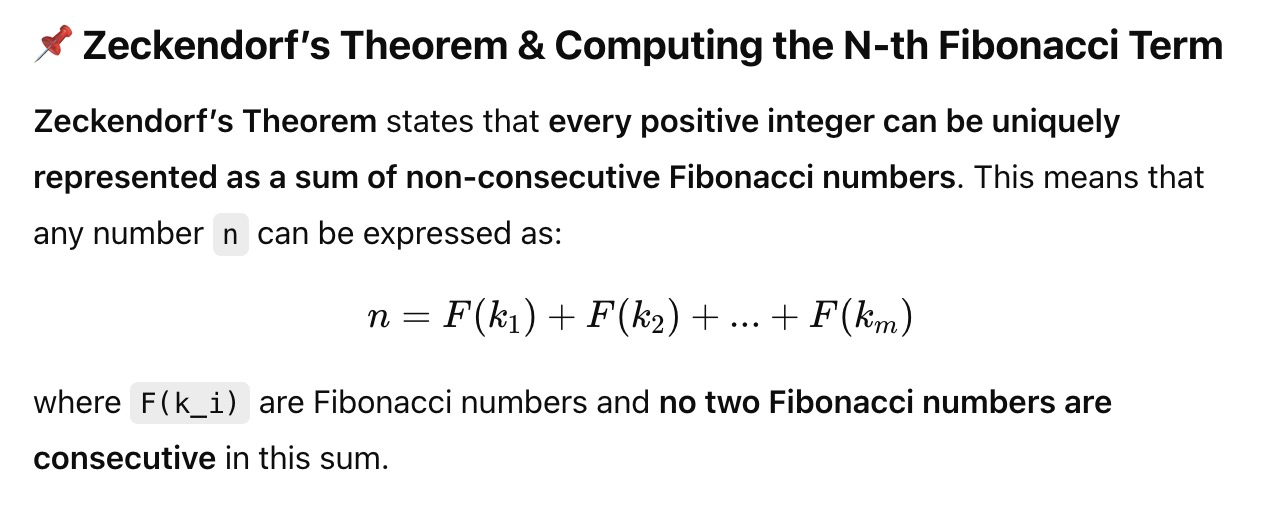
*Figure 21 Iterative Fibonacci with HashMap Memoization Method Results*

**

*Figure 22 Iterative Fibonacci with HashMap Memoization Method Graph*

The graph shows the **execution time of the Iterative Fibonacci with HashMap Memoization method**, demonstrating an overall **linear growth** as n increases, consistent with its **O(n) time complexity**. There is a noticeable peak around **n = 12,589**, likely due to **fluctuations in CPU scheduling or memory caching effects**, followed by a slight decrease, which may be attributed to Java's internal optimizations.

**The Zeckendorf Fibonacci Algorithm**

The Zeckendorf Fibonacci Algorithm computes F(n) using Zeckendorf’s Theorem, which states that any positive integer can be uniquely represented as the sum of non-consecutive Fibonacci numbers. Instead of computing Fibonacci numbers sequentially, this method precomputes Fibonacci values up to n, then selects the largest Fibonacci number ≤ n, subtracts it from n, and repeats the process while ensuring non-consecutive selection. This approach runs in O(log n) time since the number of Fibonacci terms grows exponentially, meaning the number of steps required to represent n is at most logarithmic. The space complexity is also O(log n), as only a logarithmic number of Fibonacci values need to be stored. This method is highly efficient for number theory applications, particularly in combinatorial mathematics, cryptography, and digital encoding.

*Figure 23 Zeckendorf’s Theorem*

*Algorithm Description:*

The set of operation for theZeckendorf Fibonacci Algorithm can be described in pseudocode as follows:

FUNCTION fibonacciZeckendorf(n):

IF n == 0 OR n == 1 THEN RETURN n

fibList ← [0, 1] // Store Fibonacci sequence up to n

WHILE fibList.last() ≤ n:

ADD (fibList[-1] + fibList[-2]) TO fibList

result ← 0

WHILE n > 0:

FIND largest Fibonacci number ≤ n (call it F(k))

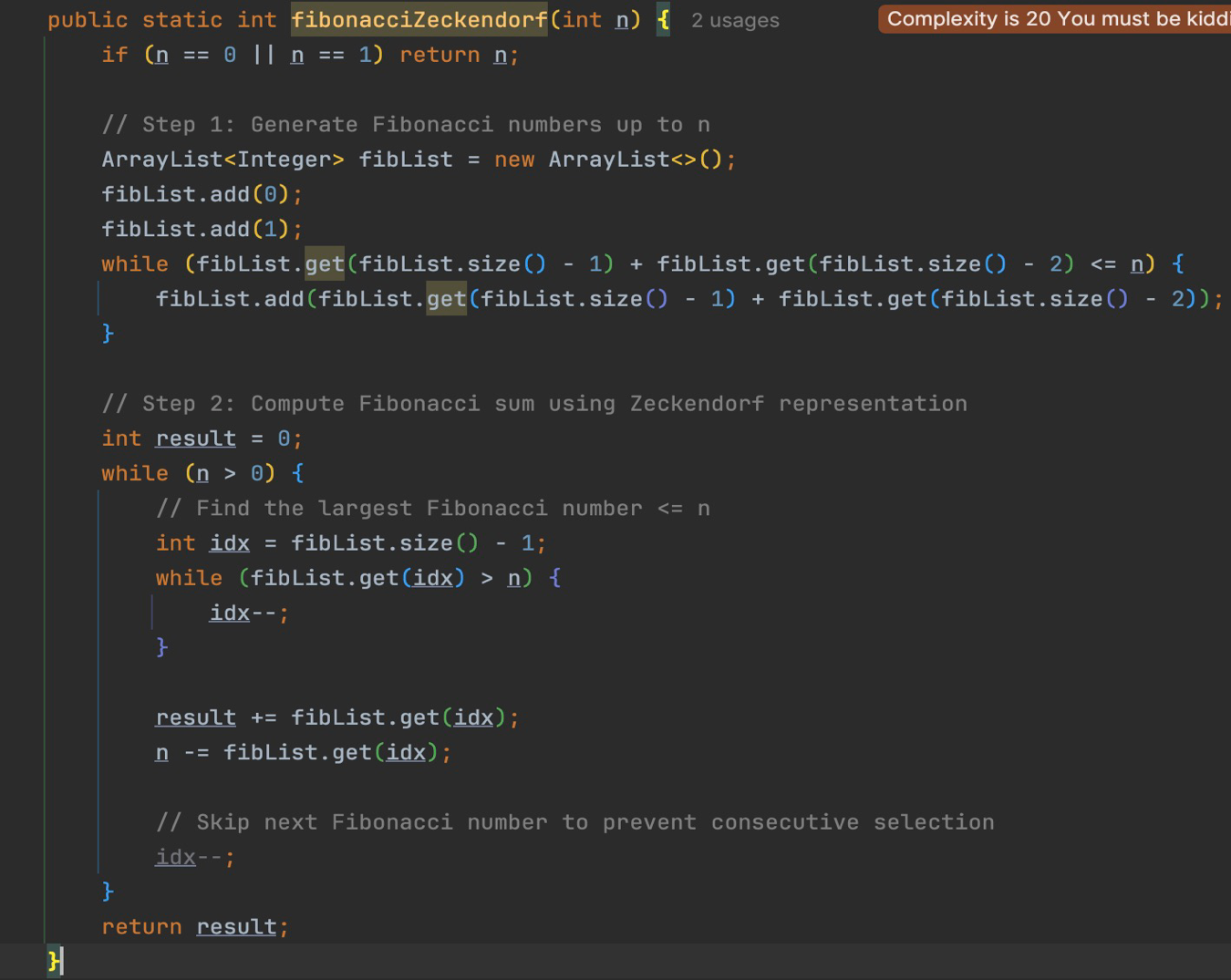
result += F(k)

n -= F(k) // Reduce n

SKIP the next Fibonacci number (to enforce non-consecutiveness)

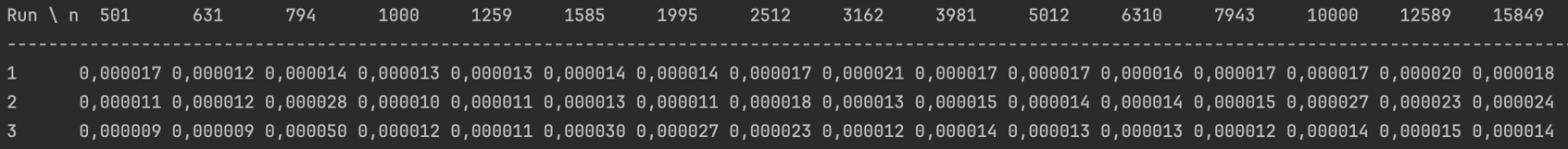
RETURN result

*Implementation:*

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

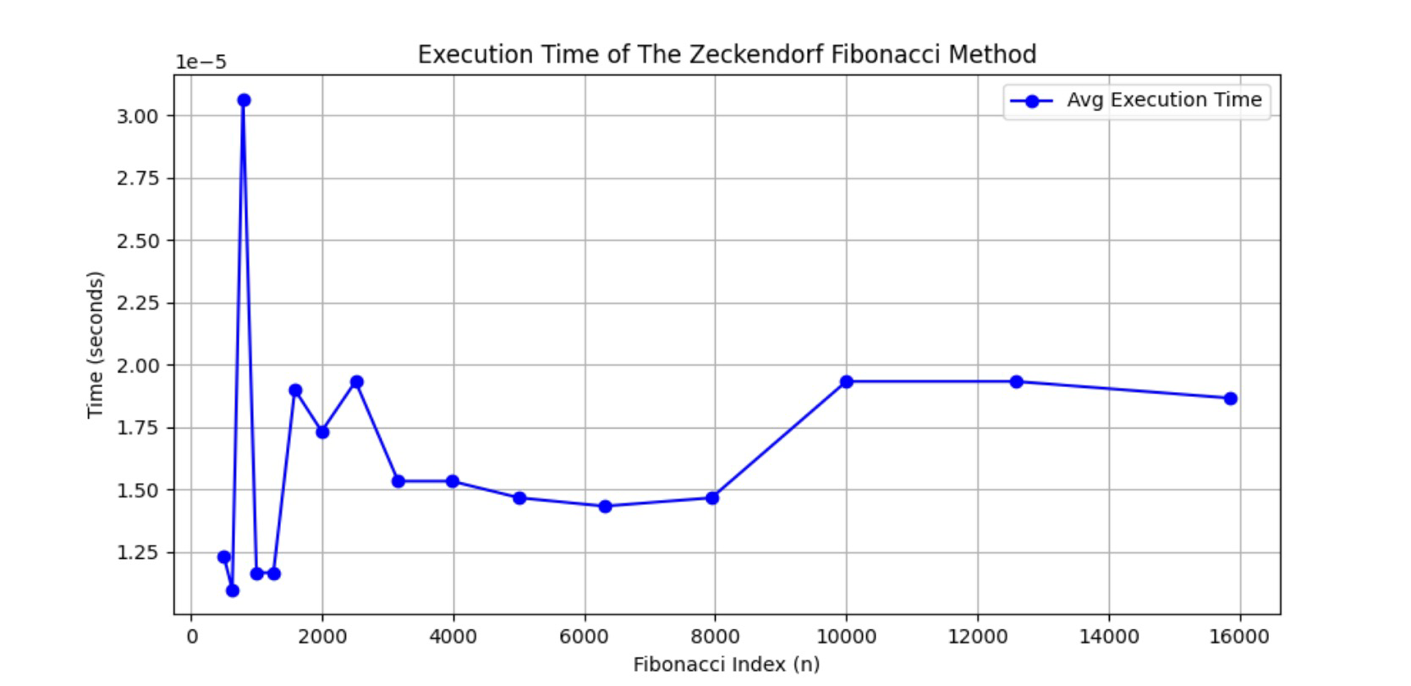
*Figure 24 Zeckendorf Fibonacci Algorithm*

*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 25 Zeckendorf Fibonacci Algorithm Results*

*Figure 25 Zeckendorf Fibonacci Algorithm Results*

**

*Figure 26 Zeckendorf Fibonacci Algorithm Graph*

The graph represents the execution time of the Zeckendorf Fibonacci Method, showing fluctuations for smaller values of n before stabilizing for larger inputs. The overall trend confirms its O(log n) complexity, as the execution time remains relatively low and consistent despite increasing n.

# 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.

Additionally, three more methods were tested for performance and efficiency.  
The Fast Doubling Method, leveraging matrix exponentiation, achieves an O(log n) time complexity, making it one of the fastest approaches for computing large Fibonacci numbers efficiently.  
The Iterative Fibonacci with HashMap Memoization follows a bottom-up approach with O(n) complexity, effectively storing computed values to reduce redundant calculations, preventing recursion depth issues.  
The Zeckendorf Fibonacci Algorithm, based on Zeckendorf’s Theorem, represents numbers as sums of non-consecutive Fibonacci numbers, resulting in an O(log n) complexity and making it useful in combinatorial mathematics and digital encoding applications.

With these additional methods, the Fast Doubling and Matrix Exponentiation methods remain the most optimal for large n, while Dynamic Programming and HashMap Memoization work best for moderately large values. The Zeckendorf approach introduces a unique representation of numbers, and Binet’s formula is best suited for quick estimations when slight precision loss is acceptable.