

Laboratory work 1:

Study and Empirical Analysis of Algorithms for Determining Fibonacci N-th Term

Elaborated:
st. gr. FAF-221

Cuzmin Simion

Verified:

asist. univ.

Fiștic Cristofor

Chişinău - 2023

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:.....	6
Dynamic Programming Method:.....	8
Matrix Power Method:.....	9
Binet Formula Method:.....	12
Recursive Method with Cache:.....	14
Bottom - up Method:.....	17
CONCLUSION.....	20

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.

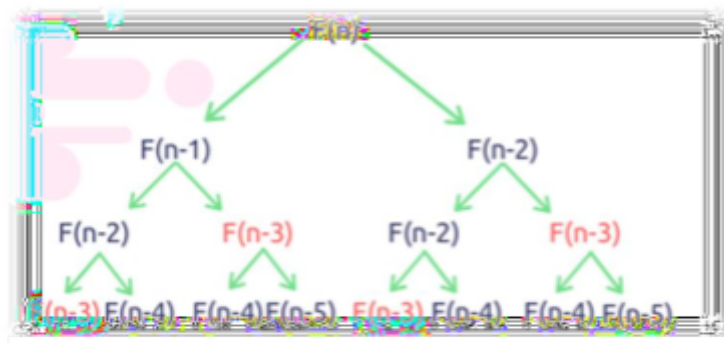


Figure 1 Fibonacci Recursion

Algorithm Description:

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

```
fibonacci_recursive(n):  
    if n <= 1:  
        return n  
    otherwise:  
        return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
```

Implementation:

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

Figure 2 Fibonacci recursion in Python

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:

Model	5	7	10	12	15	17	20	22	25	27	30	32	35	37	40	42	45
fibonacci_recursive	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.06	0.25	0.60	2.50	6.34	26.38	62.45	273.81

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.

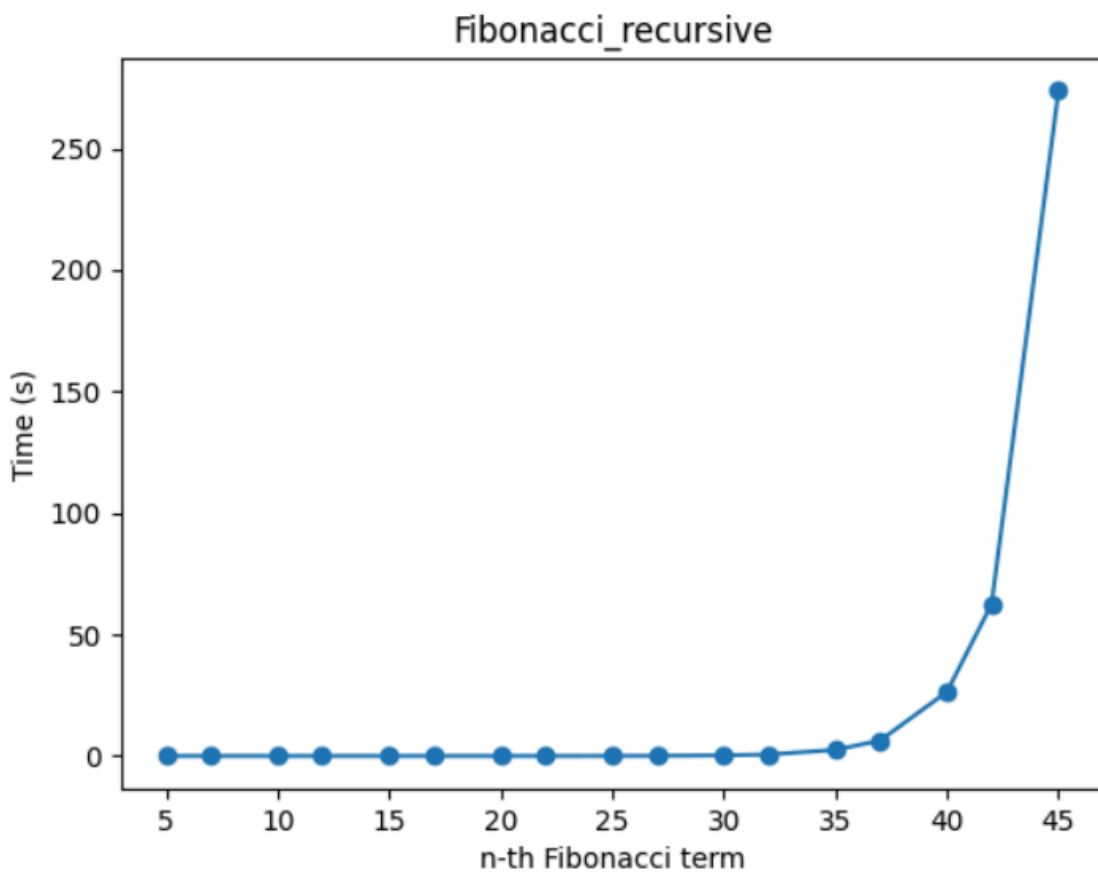


Figure 4 Graph of Recursive Fibonacci Function

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. $T(2^n)$.

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:

```
def fibonacci_iterative(n):
    a, b, c = 0, 1, 0
    if n == 0:
        return a
    for i in range(n):
        c = a + b
        a = b
        b = c
    return b
```

Implementation:

```
def fibonacci_iterative(n):
    a, b, c = 0, 1, 0
    if n == 0:
        return a
    for i in range(n):
        c = a + b
        a = b
        b = c
    return b
```

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:

Model	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
fibonacci_iterative	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.001	0.002	0.004	0.005	0.009

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),

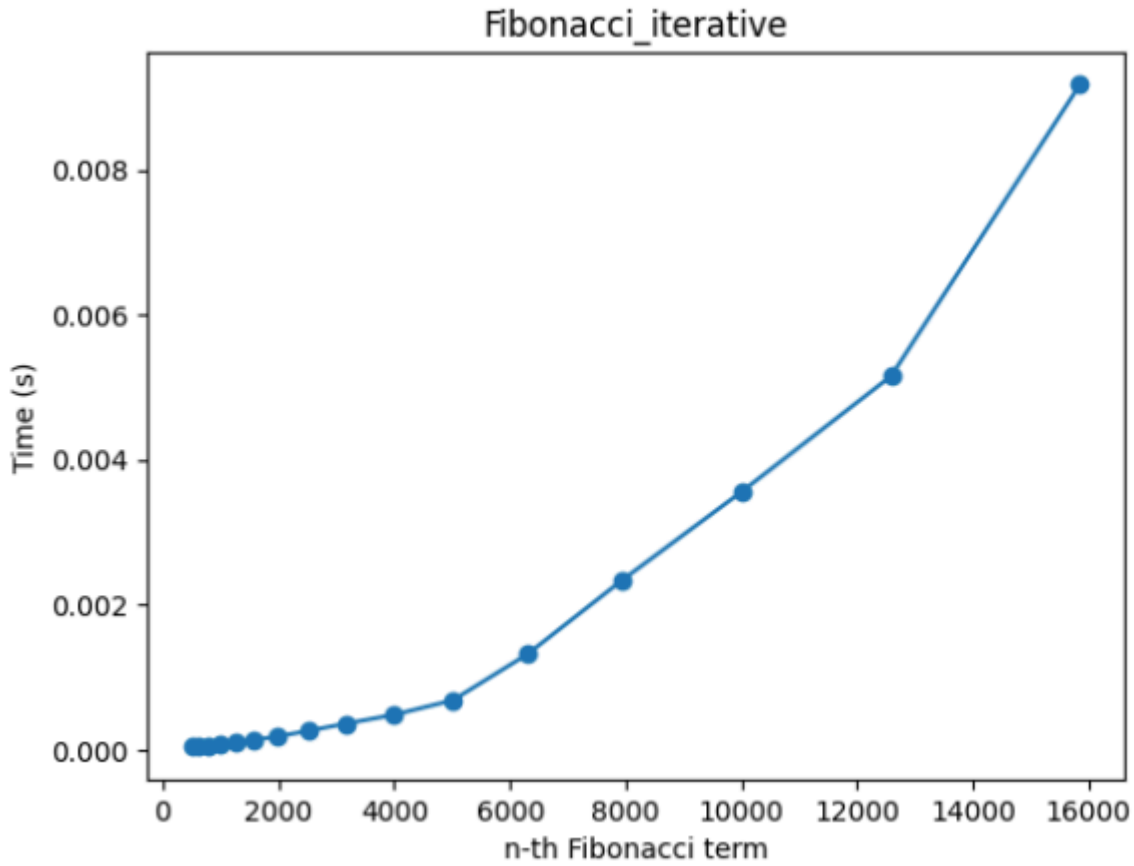


Figure 7 Fibonacci Matrix Power Method in Python

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:

```

def fibonacci_matrix(n):
    def multiply(A, B):
        return [[A[0][0]*B[0][0] + A[0][1]*B[1][0], A[0][0]*B[0][1] + A[0][1]*B[1][1]],
                [A[1][0]*B[0][0] + A[1][1]*B[1][0], A[1][0]*B[0][1] + A[1][1]*B[1][1]]]

    def power(A, n):
        if n == 1:
            return A
        if n % 2 == 0:
            half = power(A, n // 2)
            return multiply(half, half)
        else:
            return multiply(A, power(A, n - 1))

    if n == 0:
        return 0
    matrix = [[1, 1], [1, 0]]
    result_matrix = power(matrix, n - 1)
    return result_matrix[0][0]

```

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:

Model	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
fibonacci_matrix	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001

Figure 9 Matrix Method Fibonacci Results

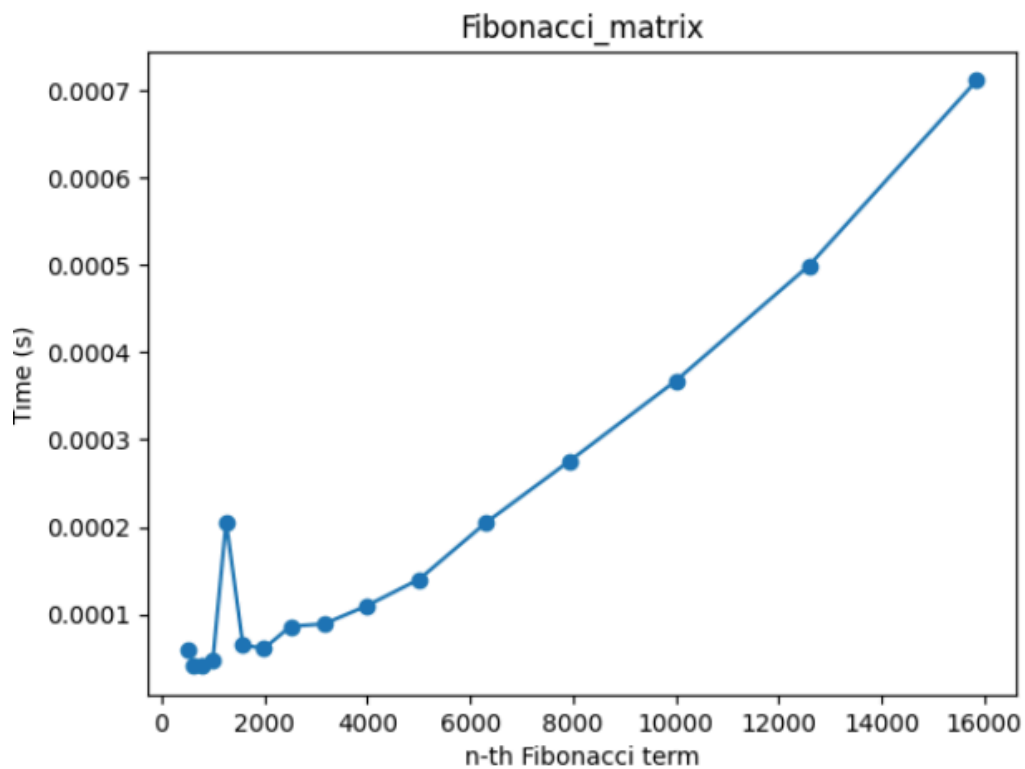


Figure 10 Matrix Method Fibonacci Results

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.

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:

```
def fibonacci_binet(n):  
    sqrt_5 = int(math.sqrt(5))  
    phi = (1 + sqrt_5) // 2  
    psi = (1 - sqrt_5) // 2  
    return int((phi ** n - psi ** 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:

```
def fibonacci_binet(n):  
    sqrt_5 = int(math.sqrt(5))  
    phi = (1 + sqrt_5) // 2  
    psi = (1 - sqrt_5) // 2  
    return int((phi ** n - psi ** n) // sqrt_5)
```

Figure 11 Fibonacci Binet Formula Method in Python

Results:

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

Model	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
fibonacci_binet	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Figure 12 Fibonacci Binet Formula Method results

And as shown in its performance graph,

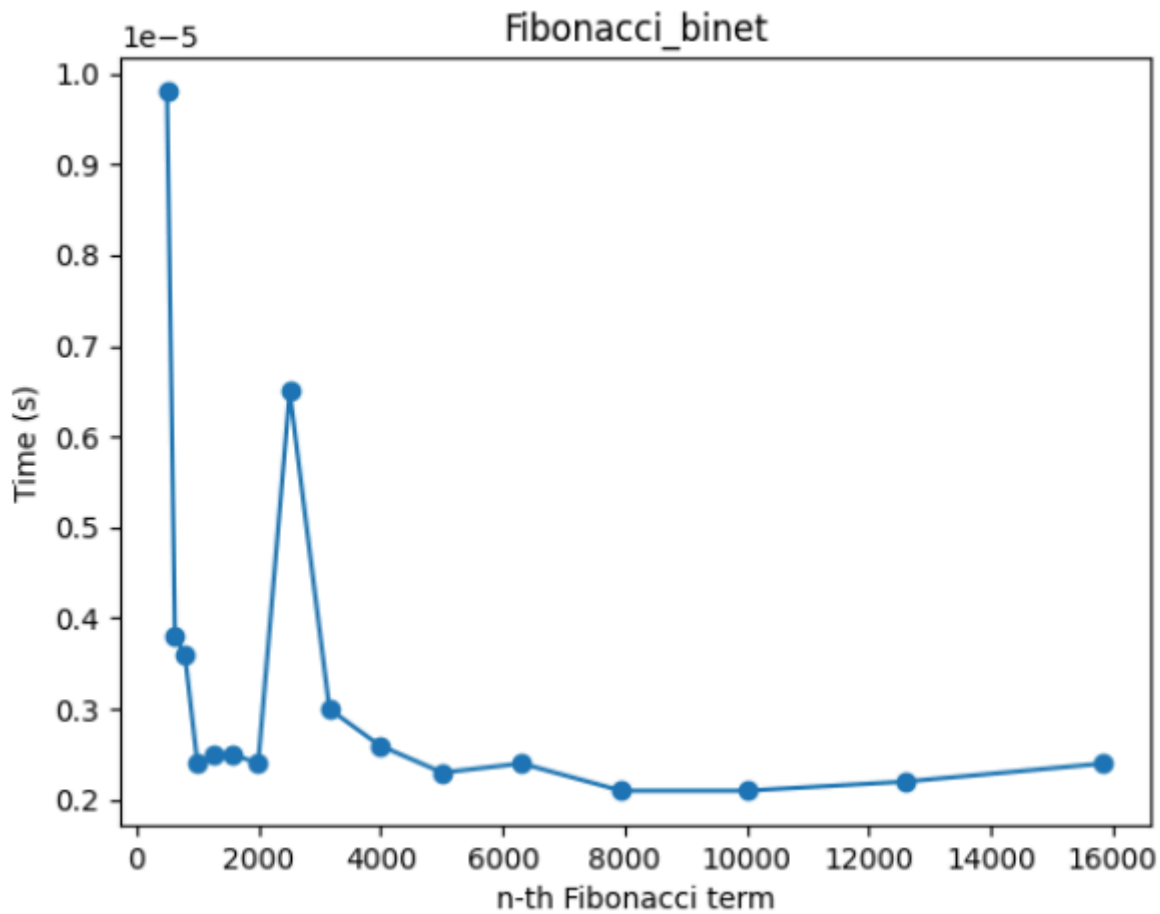


Figure 13 Fibonacci Binet formula Method

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 naïve form in python, as further modification and change of language may extend its usability further.

Recursive Method with Cache:

The Recursive Method with Cache is an optimized version of the standard recursive approach for calculating Fibonacci numbers. In this method, previously computed Fibonacci numbers are stored in a cache to avoid redundant calculations. This cache helps reduce the time complexity of the algorithm by eliminating repetitive computations. However, it is important to note that while this method improves performance compared to the naive recursive approach, it may still face limitations with larger inputs due to Python's recursion depth limit. Therefore, although it offers efficiency gains, it may not be suitable for extremely large Fibonacci numbers.

Algorithm Description:

The set of operation for Fibonacci Recursive Method with Cache the can be described in pseudocode as follows:

```
def fibonacci_recursive_cache(n, cache=None):
    if cache is None:
        cache = {}
    if n in cache:
        return cache[n]
    if n <= 1:
        return n
    cache[n] = fibonacci_recursive_cache(n - 1, cache) +
    fibonacci_recursive_cache(n - 2, cache)
    return cache[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:

```
def fibonacci_recursive_3(n, cache=None):
    if cache is None:
        cache = {}
    if n in cache:
        return cache[n]
    if n <= 1:
        return n
    cache[n] = fibonacci_recursive_3(n - 1, cache) + fibonacci_recursive_3(n - 2, cache)
    return cache[n]
```

Figure 14 Fibonacci Recursive Method with Cache in Python

Results:

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

Model	5	7	10	12	15	17	20	22	25	27	30	32	35	37	40	42	45
fibonacci_recursive_cache	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Figure 15 Fibonacci Recursive Method with Cache in Python

And as shown in its performance graph,

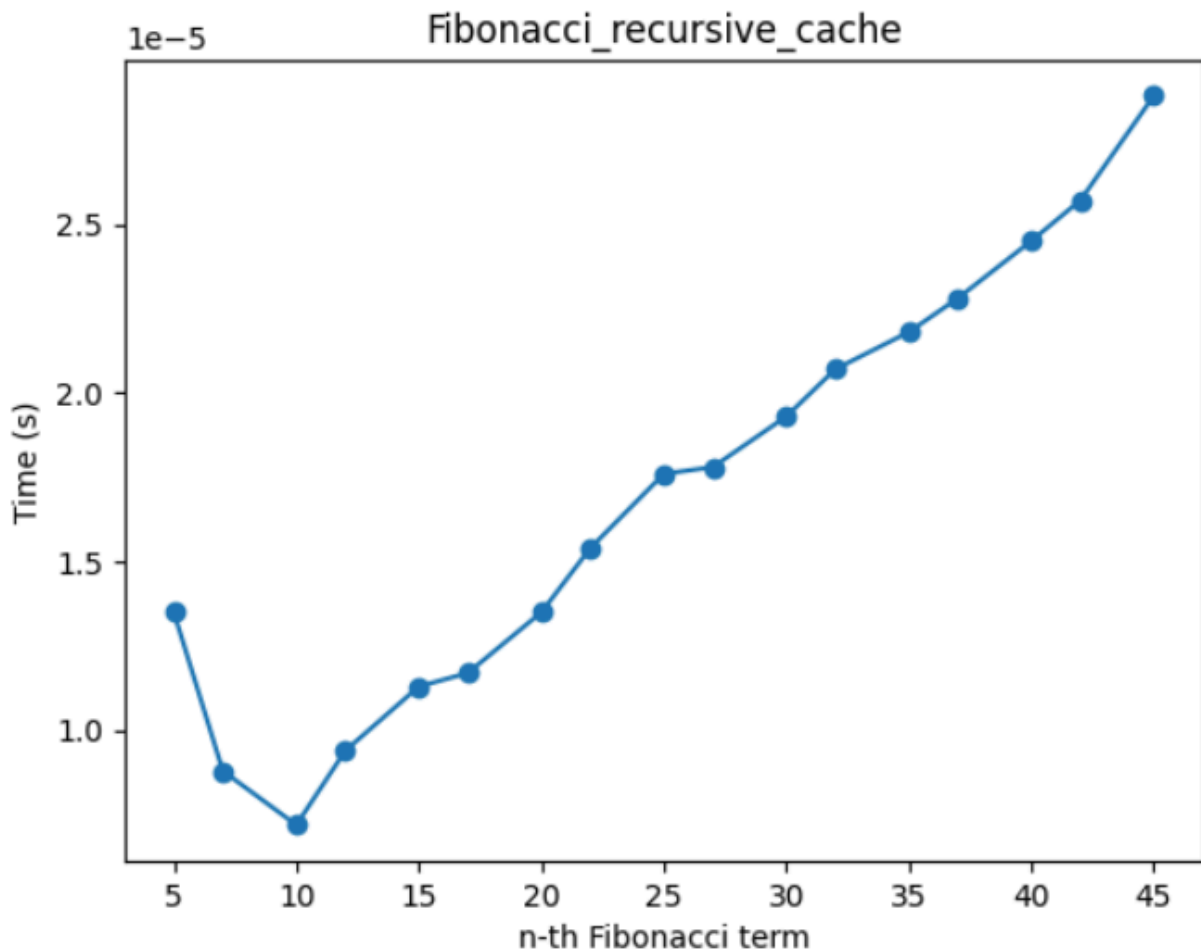


Figure 16 Fibonacci Recursive Method with Cache in Python

In summary, the Recursive Method with Cache offers an optimized approach for calculating Fibonacci numbers by storing previously computed values, thereby reducing redundant calculations and improving performance compared to the naive recursive method. However, it remains constrained by Python's recursion depth limit, making it suitable for Fibonacci terms within a certain range, typically up to around 80, to ensure accurate results and efficient computation. Further enhancements or exploring alternative programming languages may extend its usability beyond these limitations.

Bottom - up Method:

The Bottom-Up Method provides an efficient approach to computing Fibonacci numbers. Unlike the Binet Formula Method, which relies on the Golden Ratio (ϕ) and may encounter rounding errors, the Bottom-Up Method systematically constructs the Fibonacci sequence from the ground up. It starts with the smallest values and iteratively calculates each subsequent term without recursion. By leveraging precomputed values and avoiding recursive calls, this method circumvents issues related to rounding errors and recursion depth limitations. The provided Python function, `fibonacci_bottom_up(n)`, exemplifies this approach by using an iterative process to generate Fibonacci numbers up to the n th term reliably and efficiently.

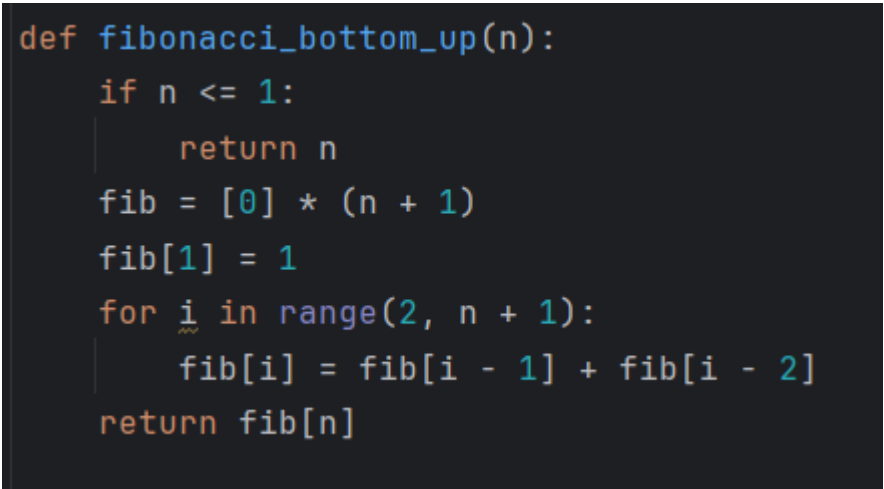
Algorithm Description:

The set of operation for the Fibonacci Bottom - up Method can be described in pseudocode as follows:

```
def fibonacci_bottom_up(n):  
    if n <= 1:  
        return n  
    fib = [0] * (n + 1)  
    fib[1] = 1  
    for i in range(2, n + 1):  
        fib[i] = fib[i - 1] + fib[i - 2]  
    return fib[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:



```
def fibonacci_bottom_up(n):  
    if n <= 1:  
        return n  
    fib = [0] * (n + 1)  
    fib[1] = 1  
    for i in range(2, n + 1):  
        fib[i] = fib[i - 1] + fib[i - 2]  
    return fib[n]
```

Figure 13 Fibonacci Bottom - up Method in Python

Results:

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

Model	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
fibonacci_bottom_up	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.001	0.001	0.001	0.002	0.004	0.007	0.011	0.015	0.014

Figure 14 Fibonacci Bottom - up Method results

And as shown in its performance graph,

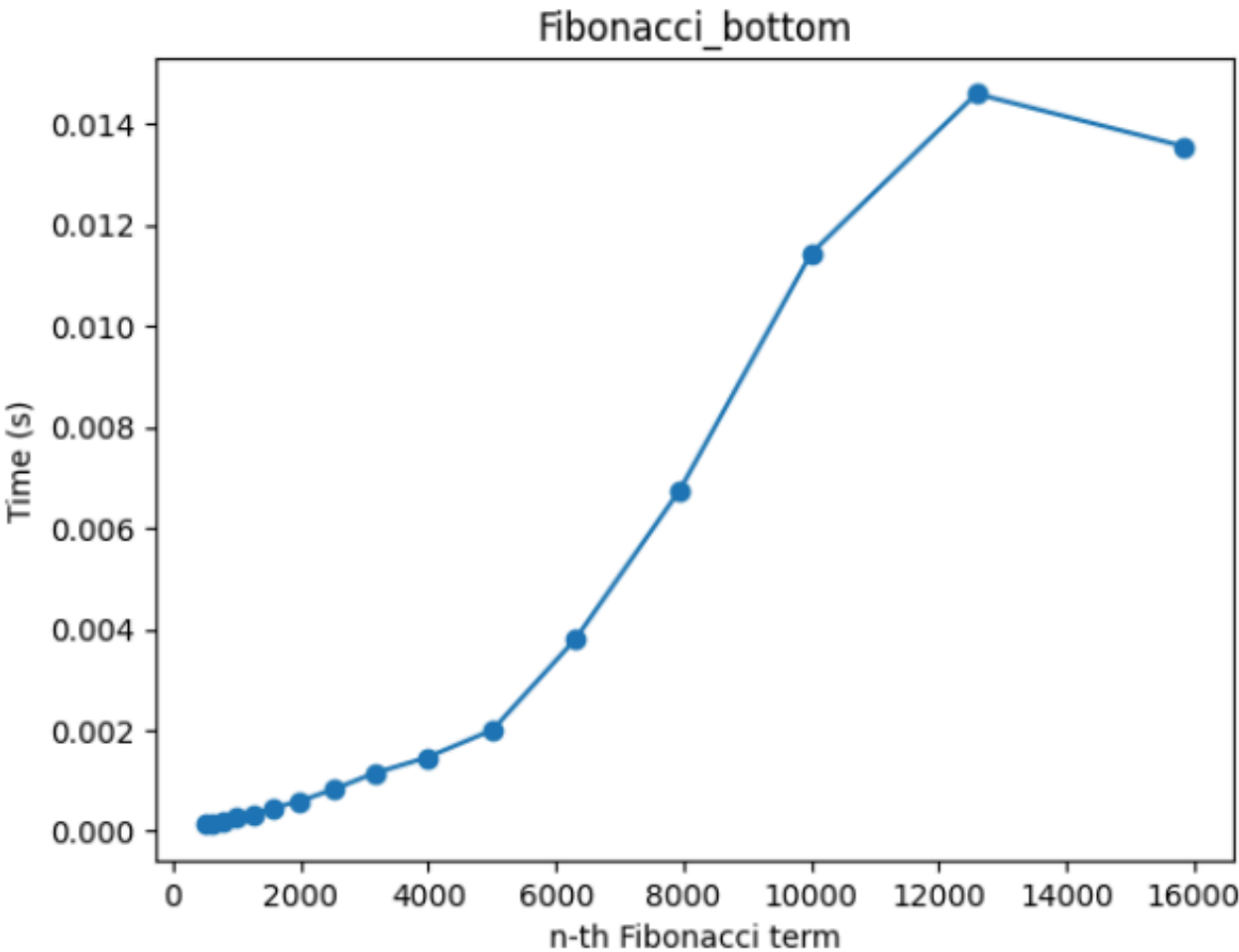


Figure 15 Fibonacci Bottom - up Method

The Bottom-Up Method is a reliable and efficient approach for computing Fibonacci numbers, even for large values of n . By systematically building the sequence from the smallest values upwards and iteratively computing each subsequent term, this method ensures accuracy without the limitations of recursive approaches. It relies on precomputed values and an iterative process, making it well-suited for handling Fibonacci terms of varying magnitudes with consistent performance. This method stands as a practical solution for Fibonacci number computation in diverse applications.

CONCLUSION

Determining which method is superior depends on various factors, including the specific requirements of the task at hand, the computational resources available, and the desired balance between accuracy and efficiency.

For tasks requiring simplicity and ease of implementation, the Recursive Method may suffice, especially for smaller order numbers. However, its exponential time complexity makes it impractical for larger inputs.

The Binet Method offers a swift alternative with near-constant time complexity, making it suitable for calculating Fibonacci numbers up to a certain threshold. However, potential rounding inaccuracies may arise, particularly for larger values of n .

The Dynamic Programming and Matrix Multiplication Methods excel in scalability, providing precise results and demonstrating linear complexity. These methods are particularly advantageous for large values of n , where accuracy and efficiency are paramount.

The Bottom-Up Method and Recursive with Cache Method offer practical solutions for Fibonacci number computation, especially for larger values of n , as they avoid the recursion depth limitations of the Recursive Method.

In summary, the choice of method depends on the specific requirements of the task. For small inputs or simplicity, the Recursive Method may suffice, while for larger inputs and precise results, the Dynamic Programming and Matrix Multiplication Methods are preferable. The Bottom-Up Method and Recursive with Cache Method offer practical alternatives for large values of n , where accuracy and efficiency are crucial.