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-221 Berco Adrei

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2024

TABLE OF CONTENTS

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:

Empirical analysis provides an alternative approach to understanding the efficiency of algorithms when mathematical complexity analysis is impractical or insufficient. This method proves beneficial in various scenarios:

1. Initial Insights: It offers preliminary insights into an algorithm's complexity class, aiding in the understanding of its efficiency characteristics.
2. Comparative Analysis: It facilitates the comparison of multiple algorithms tackling the same problem, allowing for informed decisions regarding efficiency.
3. Implementation Comparison: Empirical analysis enables the comparison of different implementations of the same algorithm, providing insights into which may perform better in practice.
4. Hardware-specific Evaluation: It helps in assessing an algorithm's efficiency on a particular computing platform, taking into account hardware constraints and capabilities.

The empirical analysis of an algorithm typically involves the following steps:

Establishing Analysis Goals: Clearly define the objectives and scope of the analysis.

1. Choosing Efficiency Metrics: Select appropriate metrics, such as the number of operations executed or the execution time, based on the analysis goals.
2. Defining Input Data Properties: Determine the characteristics of the input data relevant to the analysis, including data size or specific attributes.
3. Implementation: Develop the algorithm in a programming language, ensuring it accurately reflects the intended logic.
4. Generating Input Data Sets: Create multiple sets of input data to cover a range of scenarios and edge cases.
5. Execution and Data Collection: Execute the program for each input data set, recording

relevant performance metrics.

6. Data Analysis: Analyze the collected data, either by computing synthetic quantities like mean and standard deviation or by plotting graphs to visualize the relationship between problem size and efficiency metrics.
7. The choice of efficiency measure depends on the analysis's objectives. For instance, if assessing complexity class or verifying theoretical estimates, counting the number of operations may be suitable. Conversely, if evaluating algorithm implementation behavior, measuring execution time becomes more relevant.

8. Post-execution, recorded results undergo analysis. This involves computing statistical measures or plotting graphs to visualize the algorithm's performance characteristics in terms of problem size and efficiency metrics. Such analyses aid in making informed decisions regarding algorithm selection and optimization strategies.

Introduction:

The Fibonacci sequence, a series of numbers where each number is the sum of the two preceding numbers (e.g., 0, 1, 1, 2, 3, 5, 8, 13, ...), has been historically attributed to Leonardo Fibonacci, an Italian mathematician born around A.D. 1170. However, there are debates regarding its origin, with some pointing to ancient Sanskrit texts predating Fibonacci.

In 1202, Leonardo of Pisa, also known as Fibonacci, published Liber Abaci, a mathematical text introducing the Hindu-Arabic numeral system to the Western world. This "cookbook" for tradespeople contained arithmetic methods, including the Fibonacci sequence, useful for financial calculations.

Traditionally, the sequence was generated by adding the two preceding numbers. However, with advancements in computer science, various methods for determining the sequence have emerged, categorized into Recursive, Iterative, Dynamic Programming, Matrix Power, Binet Formula, Bottom Up methods. These methods can be implemented naively or optimized for improved performance.

In this laboratory, we will empirically analyze six naive algorithms for generating the Fibonacci sequence. Empirical analysis involves experimental observation of algorithm performance rather than mathematical derivation. Through this approach, we aim to gain insights into the practical efficiency of these algorithms.

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 6 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 o memory of the device used.

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.

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 recursiveAproach(self, n):
        if n == 0:
            return 0
        elif n == 1:
            return 1
        else:
            return Algorithms.recursiveAproach(self, n-1) + Algorithms.recursiveAproach(self,
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:

Input Size	Recursive	Iterative	Matrix	Memoization	Binet	Bottom Up
5	4.6e-06	5.7e-06	1.17e-05	8.4e-06	1.4e-05	1.56e-05
7	3.2e-06	1.7e-06	4.2e-06	1.6e-06	4.4e-06	3.8e-06
10	1.19e-05	9e-07	3.9e-06	1.1e-06	1.8e-06	1.99999e-06
12	3.38e-05	6.99998e-07	3.2e-06	1.70001e-06	9e-07	1.4e-06
15	0.0001296	1.2e-06	1.16e-05	1.2e-06	1e-06	2e-06
17	0.0003398	1.2e-06	4.7e-06	9e-07	7.99999e-07	1.8e-06
20	0.001581	1.19999e-06	3.8e-06	1e-06	6.99998e-07	2.6e-06
22	0.0039793	1.3e-06	4.6e-06	6.99998e-07	6.00005e-07	1.99999e-06
25	0.0166956	1.3e-06	4.5e-06	1.9e-06	6.99998e-07	2.60001e-06
27	0.0446022	1.4e-06	5.2e-06	6.99998e-07	7.99999e-07	2.7e-06
30	0.18794	1.5e-06	5.2e-06	1e-06	6.99998e-07	2.9e-06
32	0.519529	1.6e-06	3.99999e-06	6.99998e-07	7.99999e-07	2.9e-06
35	2.04253	1.70001e-06	5.6e-06	7.99999e-07	7.99999e-07	3.3e-06
37	5.39167	1.70001e-06	5.2e-06	5.99997e-07	7.99999e-07	3.4e-06
40	22.8748	1.80001e-06	4.50001e-06	7.99999e-07	7.00005e-07	3.4e-06
42	59.7098	1.8e-06	5.1e-06	5.99997e-07	6.00005e-07	3.5e-06
45	264.086	2.1e-06	6.5e-06	1.7e-06	7.00005e-07	3.89999e-06

Figure 1 Results for first set of inputs

In Figure 1 is represented the table of results for the first set of inputs. The first column is the input size. Next columns represent the time needed to execute each algorithm. We may notice that the only function whose time was growing for this few n terms was the Recursive Method Fibonacci function.

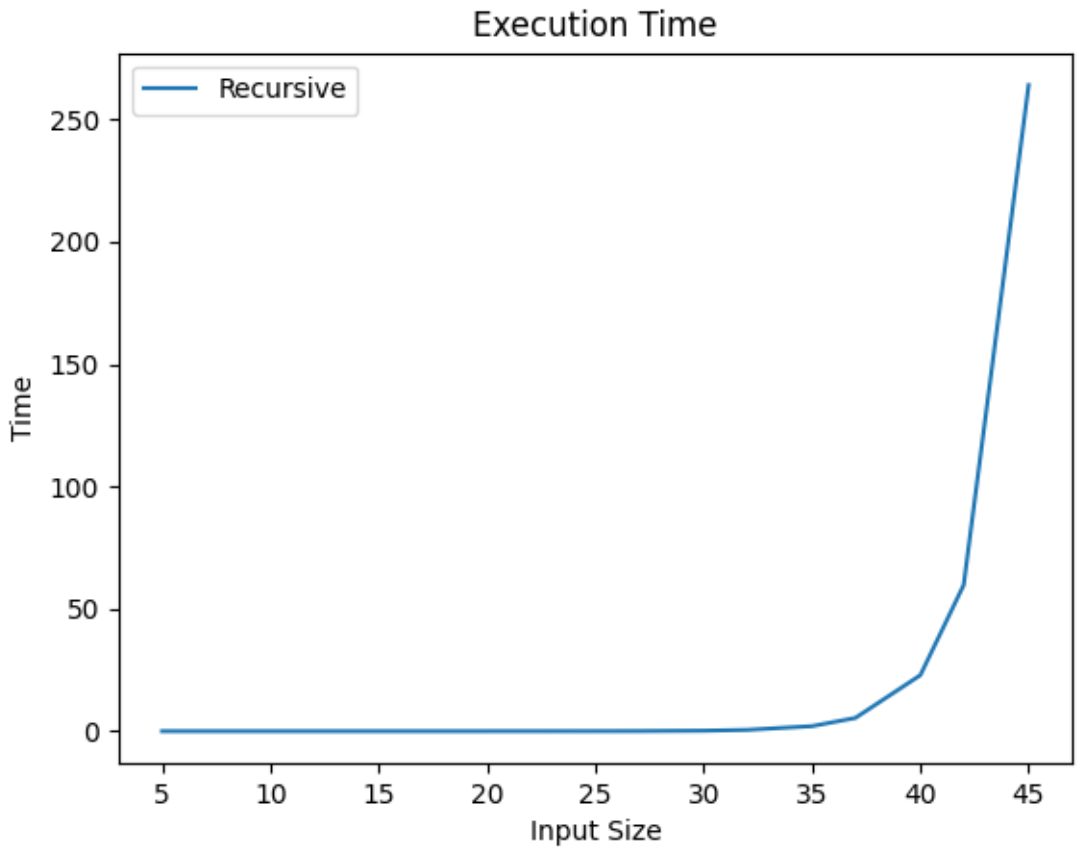

Figure 2 Graph of Recursive Fibonacci Function

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 42^{nd} term, leading us to deduce that the Time Complexity is exponential. $T(2^n)$.

Iterative Method

The iterative method for generating the Fibonacci sequence involves iteratively computing each Fibonacci number from the beginning of the sequence up to the desired nth element. Starting with the first two numbers (0 and 1), the algorithm iterates through a loop, updating the values of the previous two Fibonacci numbers to compute the next Fibonacci number. This process continues until reaching the desired nth element.

Algorithm Description (pseudocode):

```
function fibonacci_iterative(n):
    initialize variables a = 0 and b = 1
    repeat n - 1 times:
       next_fibonacci = a + b
       a = b
       b = next_fibonacci
    return b
```

Implementation:

```
    def iterativeAproach(self, n):
        a = 0
        b = 1
        for i in range(n):
            a, b = b, a + b
        return a
```

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:

Input Size	Iterative	Matrix	Memoization	Binet	Bottom Up
501	3.66e-05	2.38e-05	0.0003135	6.40001e-06	0.0001053
631	3.22e-05	1.86e-05	5.21e-05	1.5e-06	7.25e-05
794	4e-05	1.58e-05	9.96e-05	1e-06	0.0001174
1000	5.26e-05	2.21e-05	7.79e-05	6.99998e-07	0.0001154
1259	6.93e-05	1.95e-05	0.0001213	6.99998e-07	0.000172
1585	0.0001518	1.96e-05	0.0001409	7.00005e-07	0.0003299
1995	0.0002065	2.41e-05	0.0001867	6.00005e-07	0.0003436
2512	0.0001787	2.54e-05	0.0003546	7.99999e-07	0.0005723
3162	0.0003243	3.21e-05	0.0003333	7.99999e-07	0.0006178
3981	0.0003036	4.32e-05	0.0004255	1e-06	0.0007745
5012	0.0004901	8e-05	0.0005657	6.99998e-07	0.0012637
6310	0.0006831	8.13e-05	0.0008512	6.99998e-07	0.0014741
7943	0.0010432	0.0001007	0.0010863	6.99998e-07	0.0030217
10000	0.001578	0.0001386	0.0021577	6.99998e-07	0.0050098
12589	0.0021425	0.0002483	0.003189	7.00005e-07	0.0082782
15849	0.0031563	0.0002682	0.0039347	7.00005e-07	0.0090401

Figure 3 Fibonacci Iterative Results

With the Iterative Method (2nd column) showing excellent results with a timecomplexity denoted in a corresponding graph of T(n)

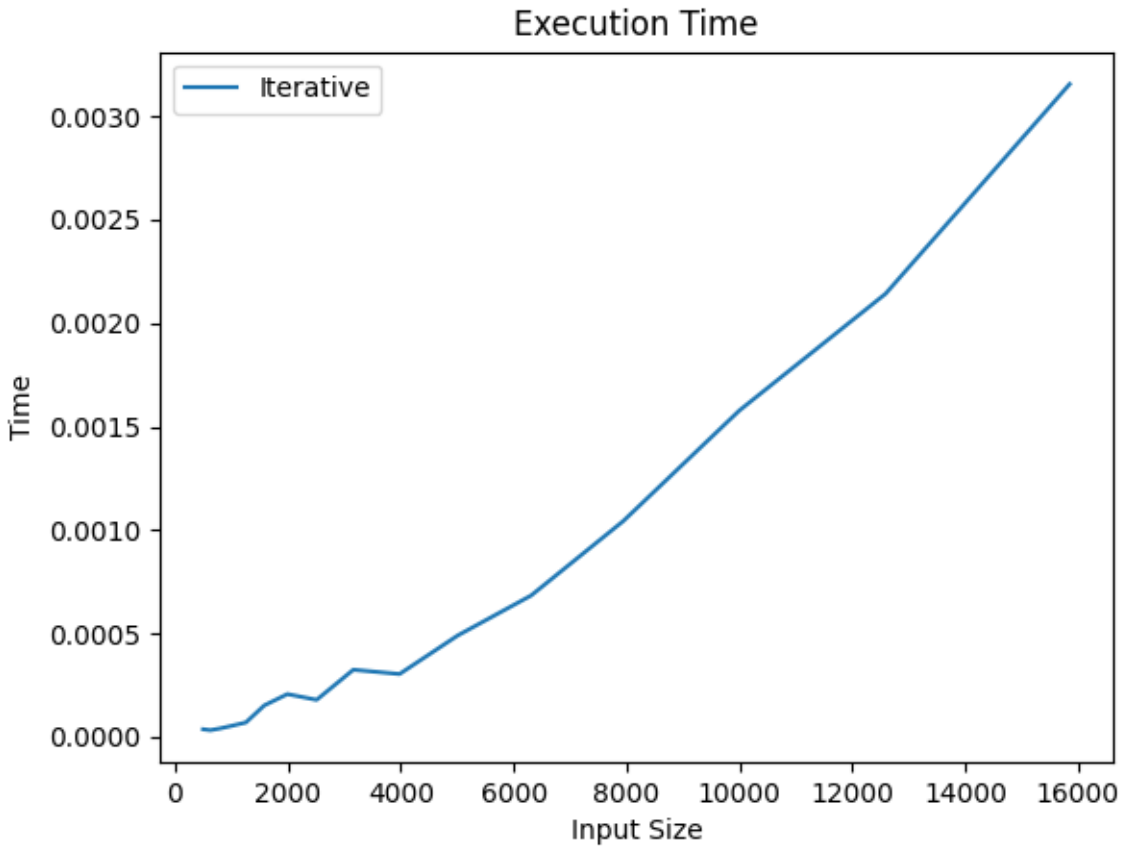

Figure 4 Fibonacci Iterative Results Graph

Dynamic Programming Method:

The Dynamic Programming method, akin to the recursive approach, adopts a direct calculation of the n-th term. However, rather than recursively invoking the function from the top down, it leverages an array data structure to store previously computed terms. This strategic use of memory ensures that previously computed values are readily available, obviating the need for redundant recalculations

Algorithm Description:

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

```
function fibonacci_bottom_up(n):
    if n <= 1:
        return n
    fib = array of size (n + 1) initialized with zeros
    fib[1] = 1
    for i from 2 to n:
        fib[i] = fib[i - 1] + fib[i - 2]
    return fib[n]
```

Implementation:

```
def fibonacci_bottom_up(self, 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]
```

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:

Input Size	Iterative	Matrix	Memoization	Binet	Bottom Up
501	3.66e-05	2.38e-05	0.0003135	6.40001e-06	0.0001053
631	3.22e-05	1.86e-05	5.21e-05	1.5e-06	7.25e-05
794	4e-05	1.58e-05	9.96e-05	1e-06	0.0001174
1000	5.26e-05	2.21e-05	7.79e-05	6.99998e-07	0.0001154
1259	6.93e-05	1.95e-05	0.0001213	6.99998e-07	0.000172
1585	0.0001518	1.96e-05	0.0001409	7.00005e-07	0.0003299
1995	0.0002065	2.41e-05	0.0001867	6.00005e-07	0.0003436
2512	0.0001787	2.54e-05	0.0003546	7.99999e-07	0.0005723
3162	0.0003243	3.21e-05	0.0003333	7.99999e-07	0.0006178
3981	0.0003036	4.32e-05	0.0004255	1e-06	0.0007745
5012	0.0004901	8e-05	0.0005657	6.99998e-07	0.0012637
6310	0.0006831	8.13e-05	0.0008512	6.99998e-07	0.0014741
7943	0.0010432	0.0001007	0.0010863	6.99998e-07	0.0030217
10000	0.001578	0.0001386	0.0021577	6.99998e-07	0.0050098
12589	0.0021425	0.0002483	0.003189	7.00005e-07	0.0082782
15849	0.0031563	0.0002682	0.0039347	7.00005e-07	0.0090401

Figure 5 Fibonacci DP Results

With the Dynamic Programming Method (last column, Bottom Up in my case) showing excellent results with a timecomplexity denoted in a corresponding graph of T(n),

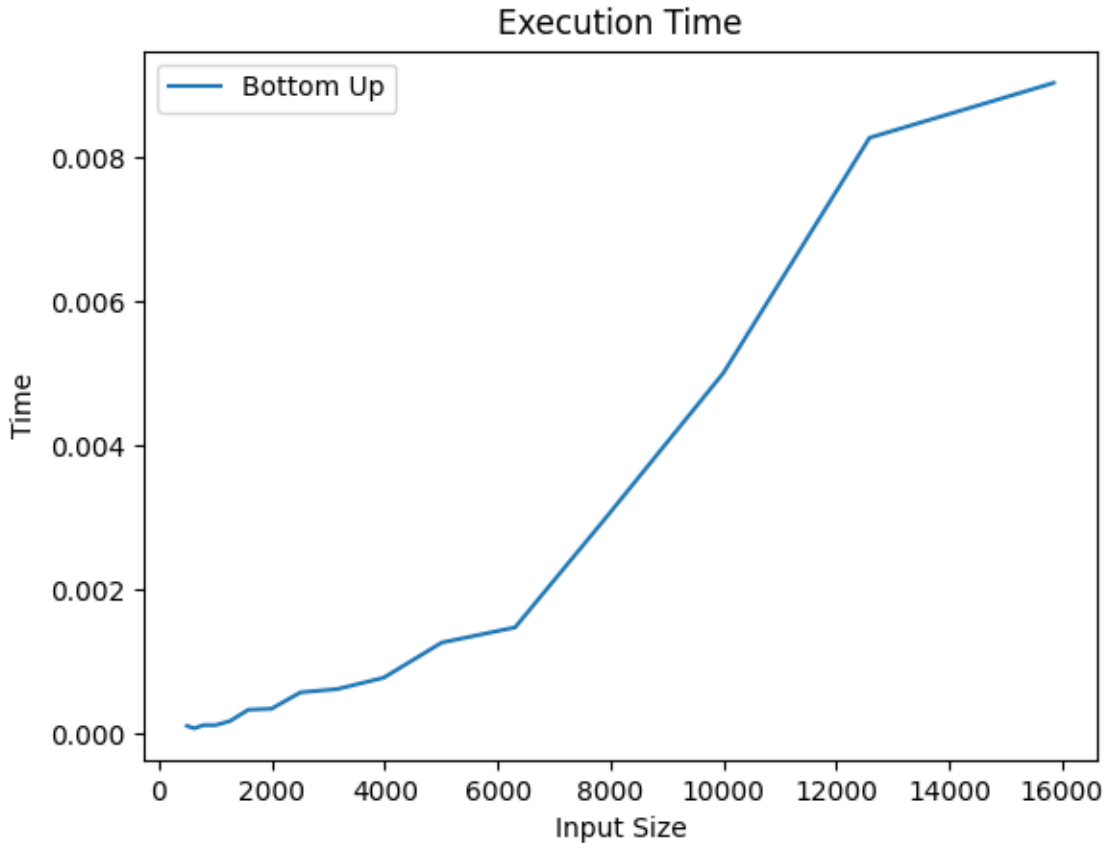

Figure 6 Fibonacci DP Graph

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:

```
def fibonacci_matrix(self, n):
        def matrix_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 matrix_power(matrix, n):
            if n == 1:
                return matrix
            elif n % 2 == 0:
                half_power = matrix_power(matrix, n//2)
                return matrix_multiply(half_power, half_power)
            else:
                half_power = matrix_power(matrix, n//2)
                return matrix_multiply(matrix_multiply(half_power, half_power), matrix)

        base_matrix = [[1, 1], [1, 0]]
        result_matrix = matrix_power(base_matrix, n)
        return result_matrix[0][1]
```

Results:

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

Input Size	Iterative	Matrix	Memoization	Binet	Bottom Up
501	3.66e-05	2.38e-05	0.0003135	6.40001e-06	0.0001053
631	3.22e-05	1.86e-05	5.21e-05	1.5e-06	7.25e-05
794	4e-05	1.58e-05	9.96e-05	1e-06	0.0001174
1000	5.26e-05	2.21e-05	7.79e-05	6.99998e-07	0.0001154
1259	6.93e-05	1.95e-05	0.0001213	6.99998e-07	0.000172
1585	0.0001518	1.96e-05	0.0001409	7.00005e-07	0.0003299
1995	0.0002065	2.41e-05	0.0001867	6.00005e-07	0.0003436
2512	0.0001787	2.54e-05	0.0003546	7.99999e-07	0.0005723
3162	0.0003243	3.21e-05	0.0003333	7.99999e-07	0.0006178
3981	0.0003036	4.32e-05	0.0004255	1e-06	0.0007745
5012	0.0004901	8e-05	0.0005657	6.99998e-07	0.0012637
6310	0.0006831	8.13e-05	0.0008512	6.99998e-07	0.0014741
7943	0.0010432	0.0001007	0.0010863	6.99998e-07	0.0030217
10000	0.001578	0.0001386	0.0021577	6.99998e-07	0.0050098
12589	0.0021425	0.0002483	0.003189	7.00005e-07	0.0082782
15849	0.0031563	0.0002682	0.0039347	7.00005e-07	0.0090401

Figure 7 Matrix Method Fibonacci Results

With the naïve Matrix method (3rd column), although being slower than the Binet, still performing pretty well, with the form f the graph indicating apretty solid T(n) time complexity.

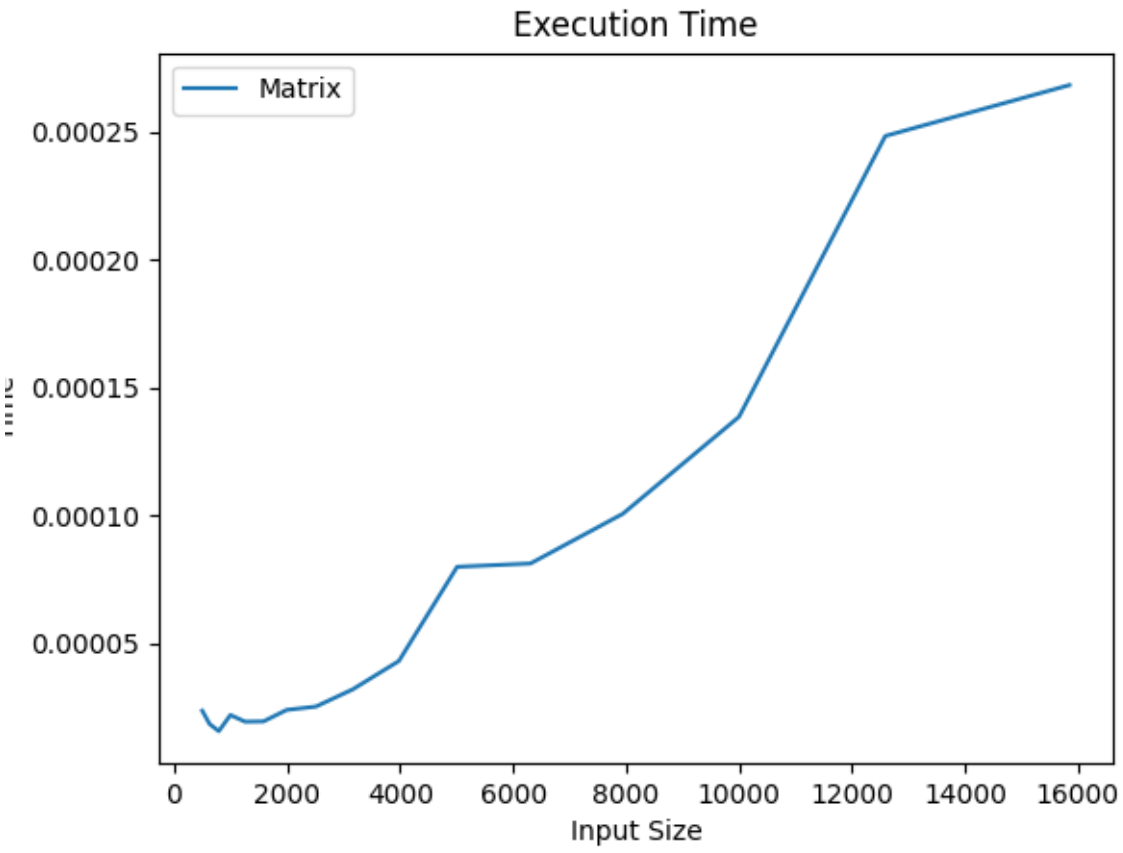

Figure 8 Matrix Method Fibonacci graph

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))
      psi <-(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:

```
    def fibonacci_binet(self, 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)
```

Results:

Although the most performant with its time, as shown in the table of results, in column 5,

Input Size	Iterative	Matrix	Memoization	Binet	Bottom Up
501	3.66e-05	2.38e-05	0.0003135	6.40001e-06	0.0001053
631	3.22e-05	1.86e-05	5.21e-05	1.5e-06	7.25e-05
794	4e-05	1.58e-05	9.96e-05	1e-06	0.0001174
1000	5.26e-05	2.21e-05	7.79e-05	6.99998e-07	0.0001154
1259	6.93e-05	1.95e-05	0.0001213	6.99998e-07	0.000172
1585	0.0001518	1.96e-05	0.0001409	7.00005e-07	0.0003299
1995	0.0002065	2.41e-05	0.0001867	6.00005e-07	0.0003436
2512	0.0001787	2.54e-05	0.0003546	7.99999e-07	0.0005723
3162	0.0003243	3.21e-05	0.0003333	7.99999e-07	0.0006178
3981	0.0003036	4.32e-05	0.0004255	1e-06	0.0007745
5012	0.0004901	8e-05	0.0005657	6.99998e-07	0.0012637
6310	0.0006831	8.13e-05	0.0008512	6.99998e-07	0.0014741
7943	0.0010432	0.0001007	0.0010863	6.99998e-07	0.0030217
10000	0.001578	0.0001386	0.0021577	6.99998e-07	0.0050098
12589	0.0021425	0.0002483	0.003189	7.00005e-07	0.0082782
15849	0.0031563	0.0002682	0.0039347	7.00005e-07	0.0090401

Figure 9 Fibonacci Binet Formula Method results

And as shown in its performance graph,

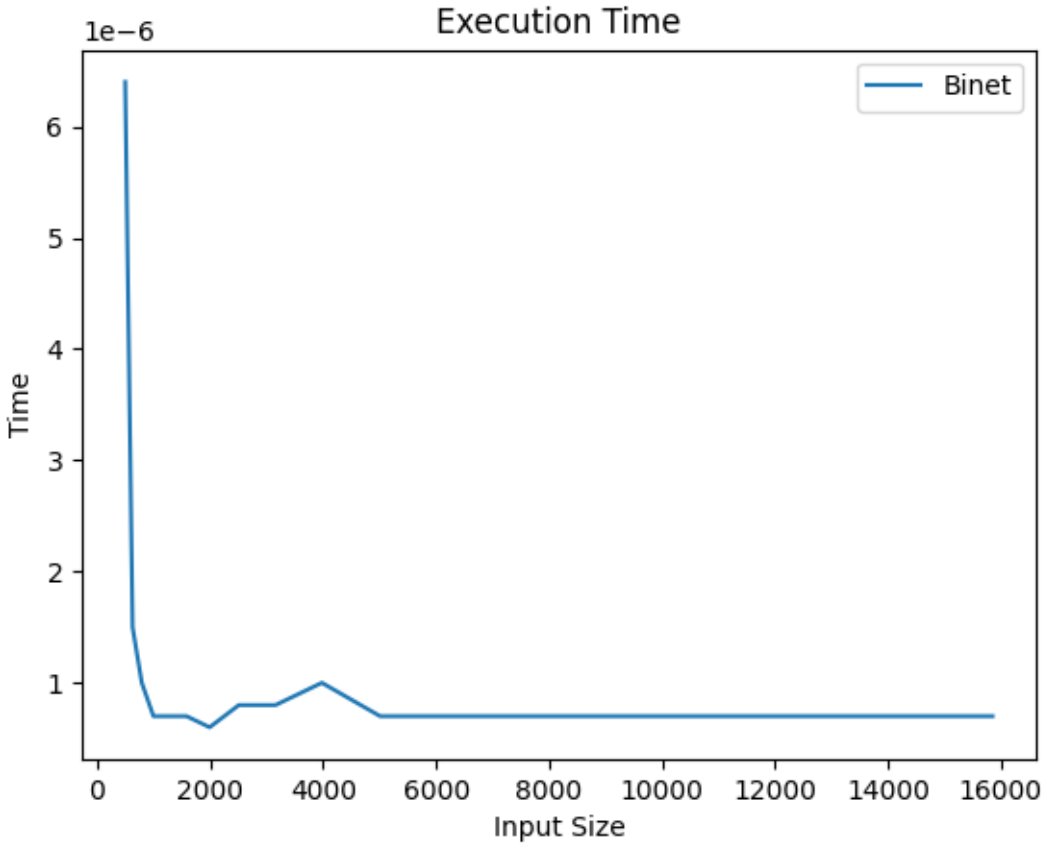

Figure 10 Fibonacci Binet formula Method

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.

Memoization Method

The Memoization Method for computing the Fibonacci sequence improves upon the recursive approach by avoiding redundant calculations through the use of memoization, which is the technique of storing previously computed results for later use.In summary, the Memoization Method optimizes the recursive approach by caching previously computed Fibonacci numbers, leading to improved efficiency and performance.

Algorithm Description

The Memoization Method for computing the Fibonacci sequence improves upon the recursive approach by avoiding redundant calculations through the use of memoization, which is the technique of storing previously computed results for later use.In summary, the Memoization Method optimizes the recursive approach by caching previously computed Fibonacci numbers, leading to improved efficiency and performance.

Implementation:

```
function fibonacci_memoization(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci_memoization(n-1, memo) + fibonacci_memoization(n-2, memo)
    return memo[n]
```

Results:

Although using recursion, the time is similar to the iterative one, because previous results are stored and not computed each time as in recursion method

Input Size	Iterative	Matrix	Memoization	Binet	Bottom Up
501	3.66e-05	2.38e-05	0.0003135	6.40001e-06	0.0001053
631	3.22e-05	1.86e-05	5.21e-05	1.5e-06	7.25e-05
794	4e-05	1.58e-05	9.96e-05	1e-06	0.0001174
1000	5.26e-05	2.21e-05	7.79e-05	6.99998e-07	0.0001154
1259	6.93e-05	1.95e-05	0.0001213	6.99998e-07	0.000172
1585	0.0001518	1.96e-05	0.0001409	7.00005e-07	0.0003299
1995	0.0002065	2.41e-05	0.0001867	6.00005e-07	0.0003436
2512	0.0001787	2.54e-05	0.0003546	7.99999e-07	0.0005723
3162	0.0003243	3.21e-05	0.0003333	7.99999e-07	0.0006178
3981	0.0003036	4.32e-05	0.0004255	1e-06	0.0007745
5012	0.0004901	8e-05	0.0005657	6.99998e-07	0.0012637
6310	0.0006831	8.13e-05	0.0008512	6.99998e-07	0.0014741
7943	0.0010432	0.0001007	0.0010863	6.99998e-07	0.0030217
10000	0.001578	0.0001386	0.0021577	6.99998e-07	0.0050098
12589	0.0021425	0.0002483	0.003189	7.00005e-07	0.0082782
15849	0.0031563	0.0002682	0.0039347	7.00005e-07	0.0090401

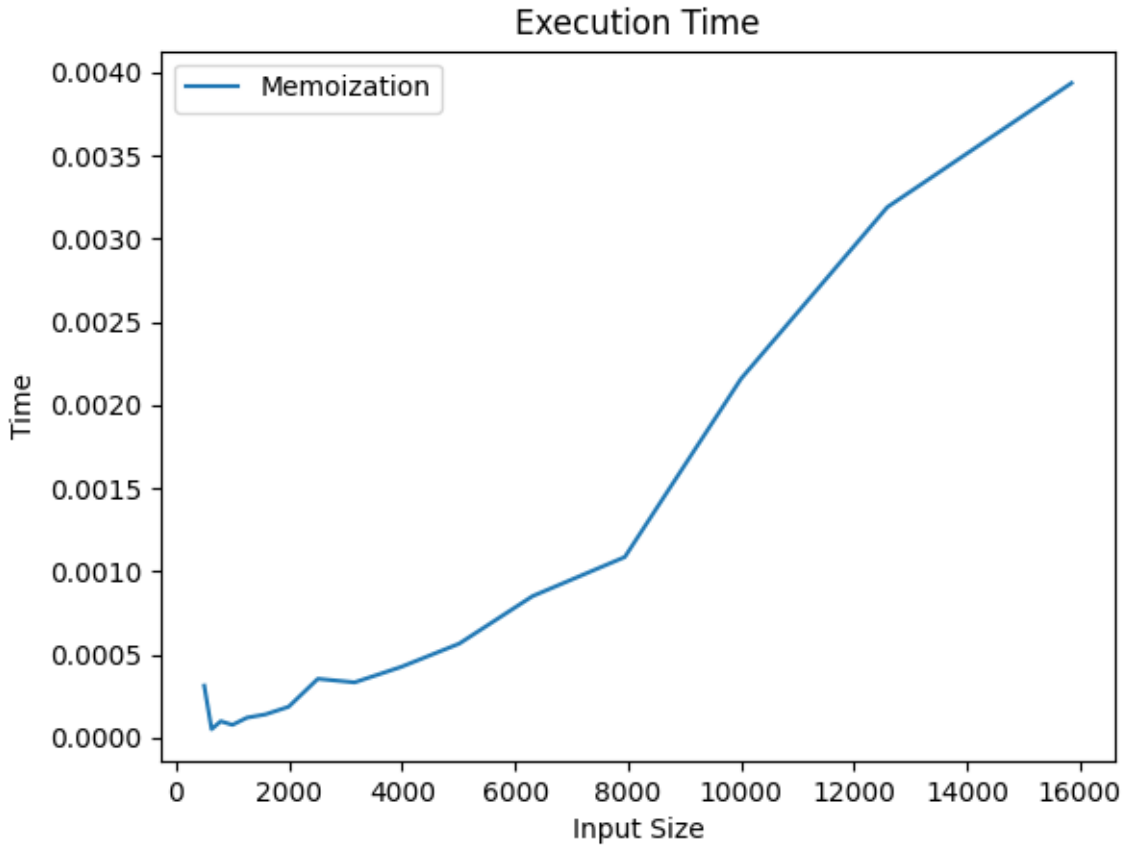

Figure 11 Fibonacci Memoization Method

CONCLUSION

Through empirical analysis, this paper evaluates six methods for computing Fibonacci numbers, considering their accuracy and time complexity. Each method is assessed to determine its suitability for various scenarios and potential areas for improvement.

Recursive Method: This method, while straightforward to implement, suffers from exponential time complexity. It is suitable for smaller Fibonacci numbers, typically up to around order 30.

Iterative Method: The iterative approach offers a more efficient alternative to the recursive method, with linear time complexity. It is suitable for computing Fibonacci numbers up to larger orders without experiencing exponential increases in execution time.

Dynamic Programming: By utilizing memoization or bottom-up approaches, dynamic programming efficiently computes Fibonacci numbers by storing previously computed results. This method provides exact results with linear time complexity, making it suitable for computing Fibonacci numbers beyond the limitations of the recursive approach.

Matrix Power Method: This method leverages matrix exponentiation techniques to compute Fibonacci numbers efficiently. While it requires additional computation overhead for matrix operations, it offers the potential for optimization to achieve logarithmic time complexity.

Binet Formula Method: The Binet formula provides an elegant and efficient closed-form expression for computing Fibonacci numbers. However, caution is advised due to potential rounding errors, particularly when dealing with floating-point arithmetic.

Bottom-Up Method: Similar to dynamic programming, the bottom-up approach efficiently computes Fibonacci numbers by iteratively building up the sequence from the bottom. It offers exact results with linear time complexity, making it suitable for larger Fibonacci numbers.

Overall, this empirical analysis highlights the strengths and weaknesses of each method, providing insights into their practical applications and potential optimization avenues. Depending on the specific requirements and constraints of the problem at hand, appropriate selection and optimization of these methods can lead to improved performance and accuracy in Fibonacci number computation.