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 Chichioi Iuliana

Verified:

asist. univ. Fiștic Cristofor

Chişinău - 2023

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: 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 six 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.

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.

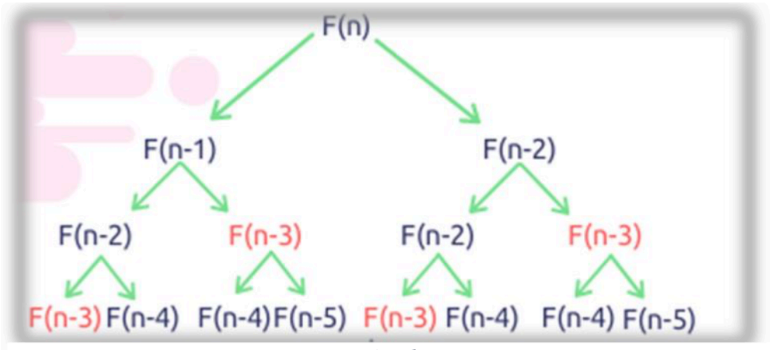


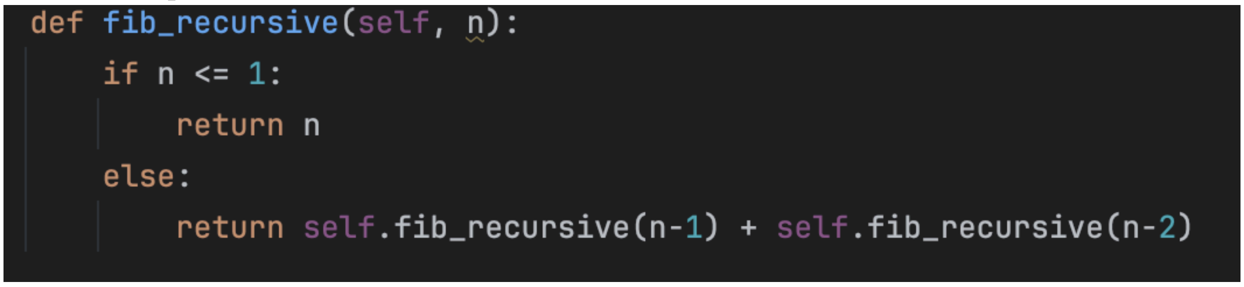
Figure 1 Fibonacci Recursion

Algorithm Description:

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

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

Implementation:

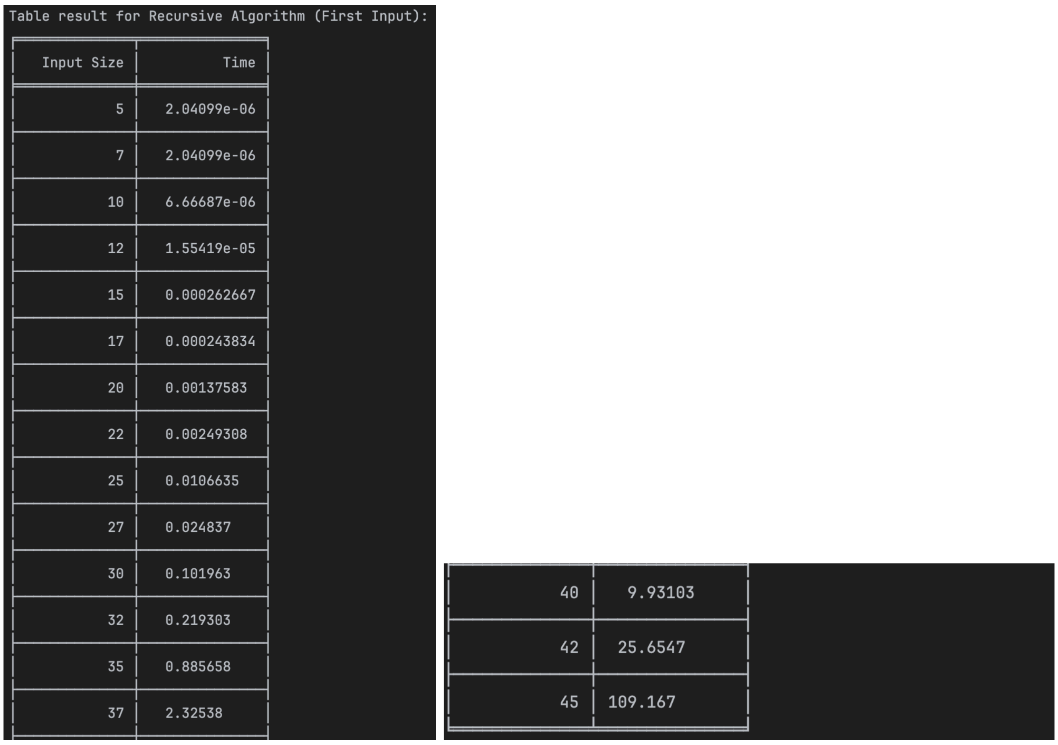


```python
def fib_recursive(self, n):
    if n <= 1:
        return n
    else:
        return self.fib_recursive(n-1) + self.fib_recursive(n-2)
```

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:



```
Table result for Recursive Algorithm (First Input):

| Input Size |        Time |
|          5 | 2.04099e-06 |
|          7 | 2.04099e-06 |
|         10 | 6.66687e-06 |
|         12 | 1.55419e-05 |
|         15 | 0.000262667 |
|         17 | 0.000243834 |
|         20 | 0.00137583  |
|         22 | 0.00249308  |
|         25 | 0.0106635   |
|         27 | 0.024837    |
|         30 | 0.101963    |
|         32 | 0.219303    |
|         35 | 0.885658    |
|         37 | 2.32538     |

|   40 |   9.93103 |
|   42 |  25.6547  |
|   45 | 109.167   |
```

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 rows) denotes the Fibonacci n-th term for which the functions were run. Starting from the second column, 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.

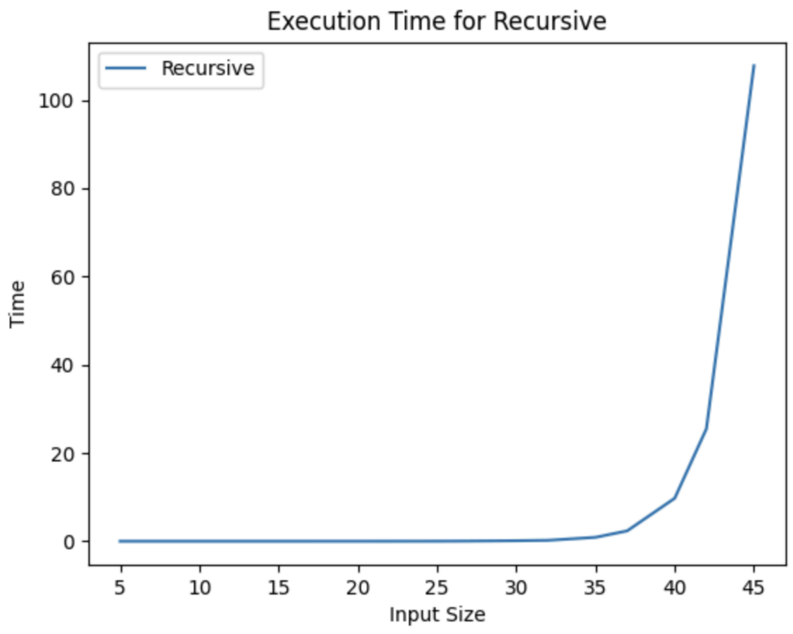


Figure 4 Graph of Recursive Fibonacci Function

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).

Iterative Method:

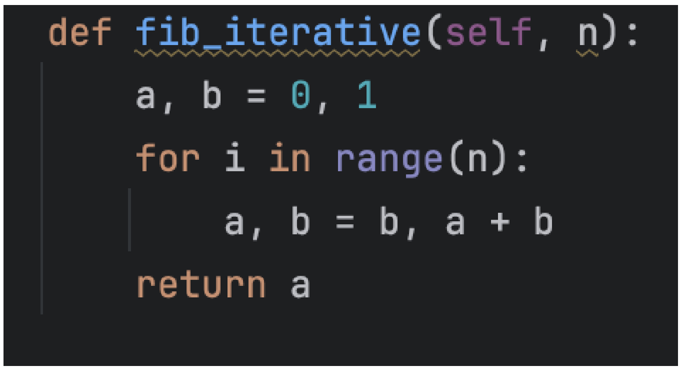
The iterative method provides a more efficient approach compared to the recursive method. It iteratively calculates each Fibonacci number starting from the base cases up to the desired term n. This method doesn't rely on function calls and doesn't suffer from the overhead associated with recursion.

Algorithm Description:

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

```
function fib_iterative(n):
    Initialize variables a = 0 and b = 1

    Repeat n - 1 times:
        next_fib = a + b
        a = b
        b = next_fib

    Return b
```

Implementation:

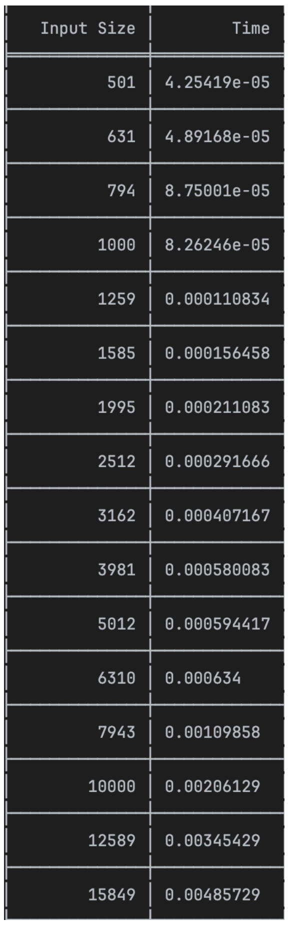


```python
def fib_iterative(self, n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

Figure 5 Fibonacci iterative in Python

Results:

After running the function for each n Fibonacci term proposed in the list from the second Input Format and saving the time for each n, we obtained the following results:



Input Size	Time
501	4.25419e-05
631	4.89168e-05
794	8.75001e-05
1000	8.26246e-05
1259	0.000110834
1585	0.000156458
1995	0.000211083
2512	0.000291666
3162	0.000407167
3981	0.000580083
5012	0.000594417
6310	0.000634
7943	0.00109858
10000	0.00206129
12589	0.00345429
15849	0.00485729

Figure 6 Results for second set of inputs

In Figure 6 is represented the table of results for the second set of inputs.With the Iterative Method (2nd column) showing excellent results with a time complexity denoted in a corresponding graph of T(n)

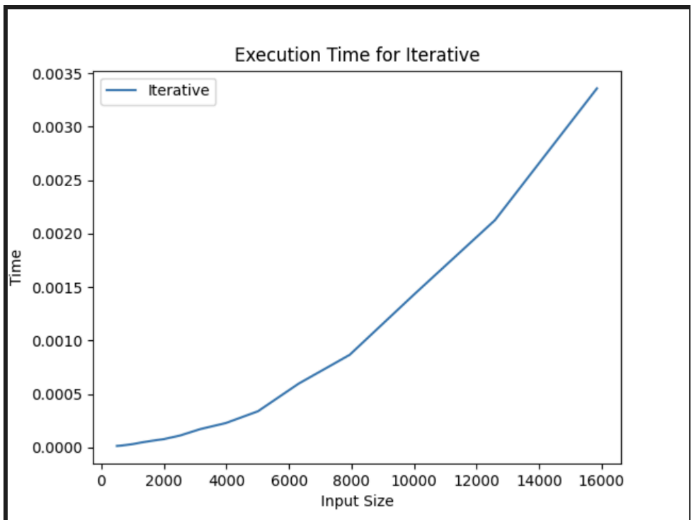


Figure 7 Graph of Iterative Fibonacci Function

Dynamic Programming Method:

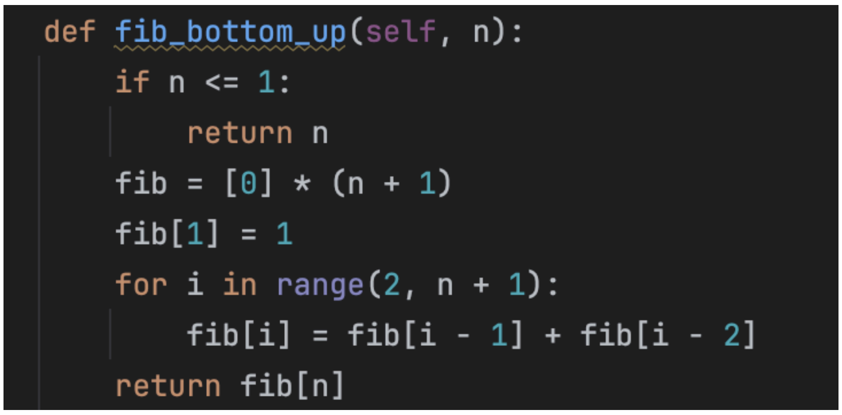
The Dynamic Programming method, similar to the recursive method, takes the straightforward approach of calculating the n-th term. However, instead of calling the function upon itself, from top down it operates based on an array data structure that holds the previously computed terms, eliminating the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

Algorithm Description:

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

```
function fib_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:

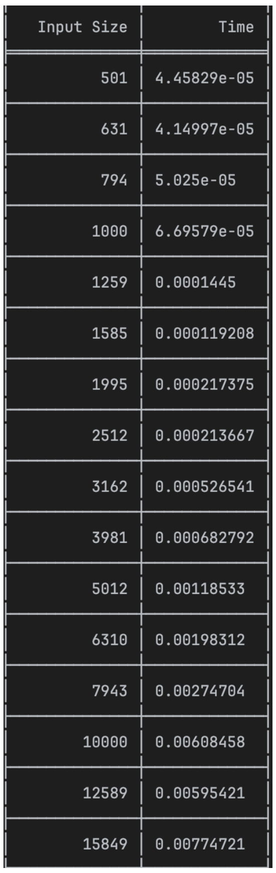


```python
def fib_bottom_up(self, n):
    if n <= 1:
        return n
    fib = [0] * (n + 1)
    fib[1] = 1
    for i in range(2, n + 1):
        fib[i] = fib[i - 1] + fib[i - 2]
    return fib[n]
```

Figure 8 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:



```
|   Input Size | Time        |
|==============|=============|
|          501 | 4.45829e-05 |
|--------------|-------------|
|          631 | 4.14997e-05 |
|--------------|-------------|
|          794 | 5.025e-05   |
|--------------|-------------|
|         1000 | 6.69579e-05 |
|--------------|-------------|
|         1259 | 0.0001445   |
|--------------|-------------|
|         1585 | 0.000119208 |
|--------------|-------------|
|         1995 | 0.000217375 |
|--------------|-------------|
|         2512 | 0.000213667 |
|--------------|-------------|
|         3162 | 0.000526541 |
|--------------|-------------|
|         3981 | 0.000682792 |
|--------------|-------------|
|         5012 | 0.00118533  |
|--------------|-------------|
|         6310 | 0.00198312  |
|--------------|-------------|
|         7943 | 0.00274704  |
|--------------|-------------|
|        10000 | 0.00608458  |
|--------------|-------------|
|        12589 | 0.00595421  |
|--------------|-------------|
|        15849 | 0.00774721  |
```

Figure 9 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),

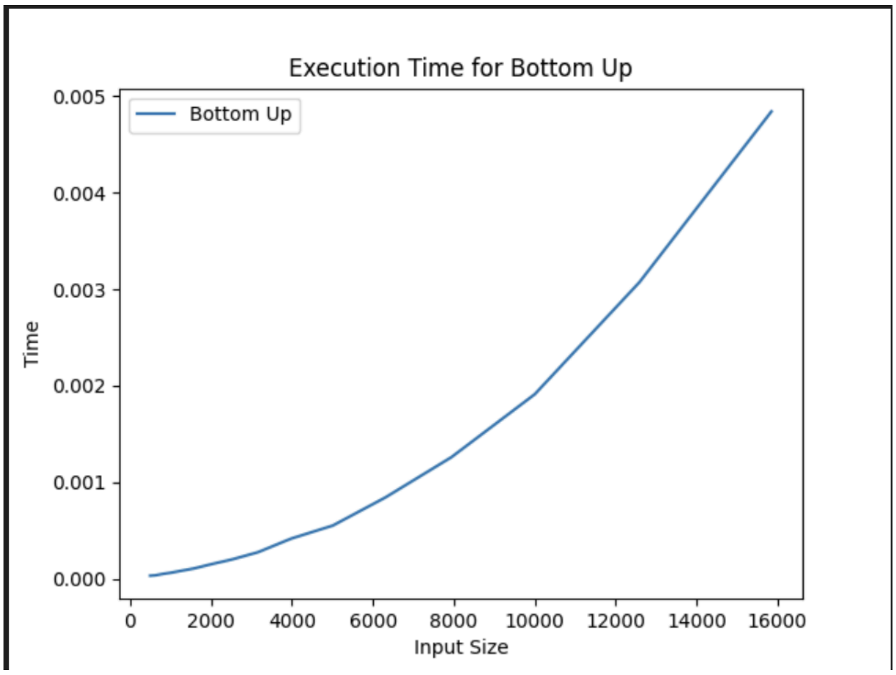


Figure 10 Fibonacci DP Graph

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:

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:

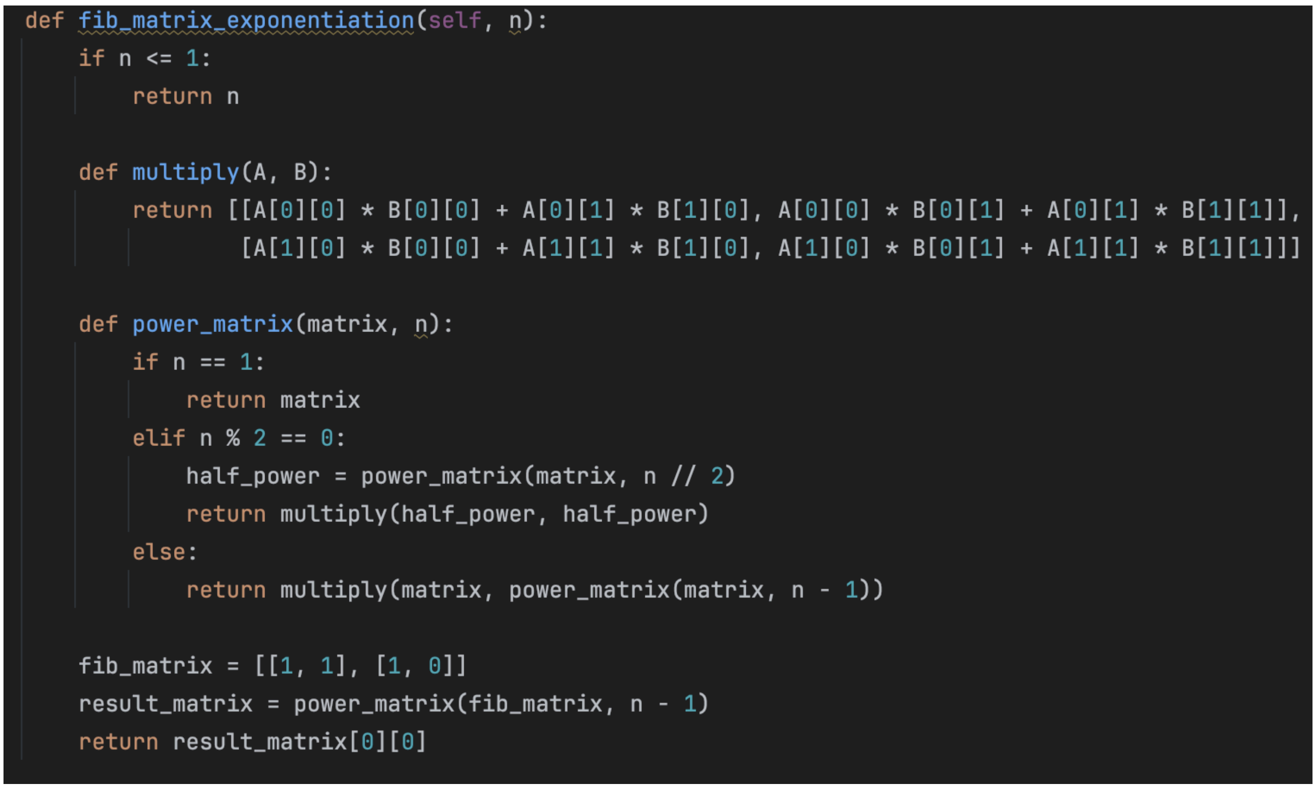
$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n\begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

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:

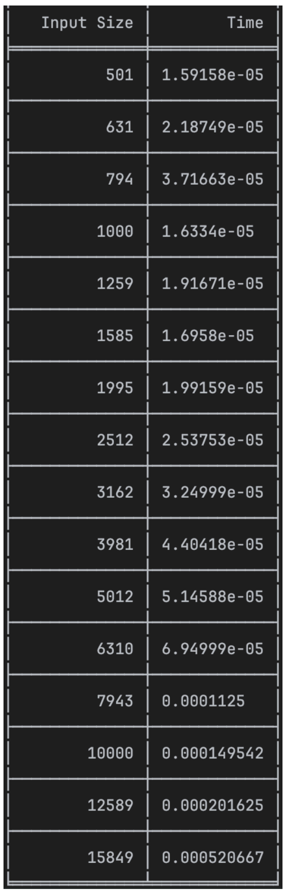


```python
def fib_matrix_exponentiation(self, n):
    if n <= 1:
        return n

    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_matrix(matrix, n):
        if n == 1:
            return matrix
        elif n % 2 == 0:
            half_power = power_matrix(matrix, n // 2)
            return multiply(half_power, half_power)
        else:
            return multiply(matrix, power_matrix(matrix, n - 1))

    fib_matrix = [[1, 1], [1, 0]]
    result_matrix = power_matrix(fib_matrix, n - 1)
    return result_matrix[0][0]
```

Figure 11 Power Function 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:



Input Size	Time
501	1.59158e-05
631	2.18749e-05
794	3.71663e-05
1000	1.6334e-05
1259	1.91671e-05
1585	1.6958e-05
1995	1.99159e-05
2512	2.53753e-05
3162	3.24999e-05
3981	4.40418e-05
5012	5.14588e-05
6310	6.94999e-05
7943	0.0001125
10000	0.000149542
12589	0.000201625
15849	0.000520667

Figure 12 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.

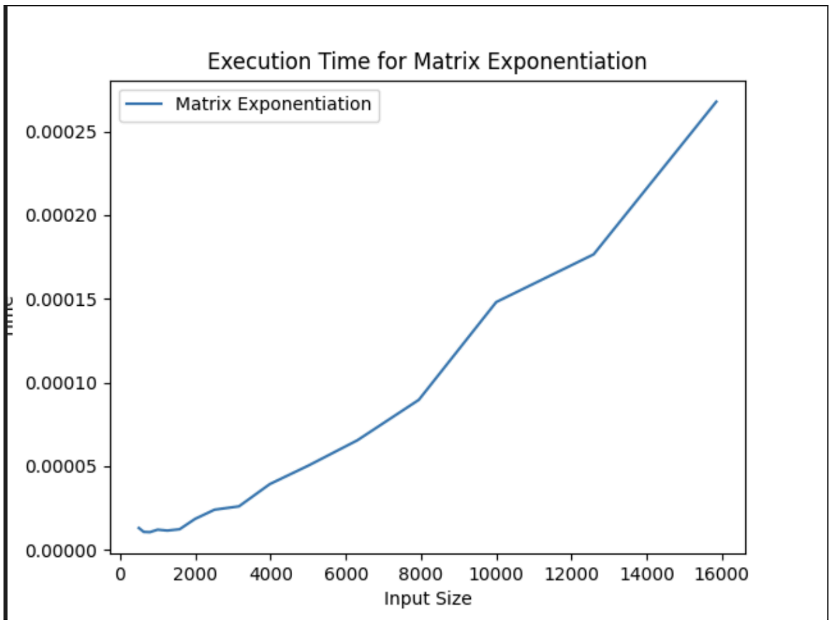


Figure 13 Matrix Method Fibonacci graph

Binet Formula Method:

The Binet Formula Method is another unconventional way of calculating the n-th term of the Fibonacci series, as it operates using the Golden Ratio formula, or phi. However, due to its nature of requiring the usage of decimal numbers, at some point, the rounding error of python that accumulates, begins affecting the results significantly. The observation of error starting with around 70-th number making it unusable in practice, despite its speed.

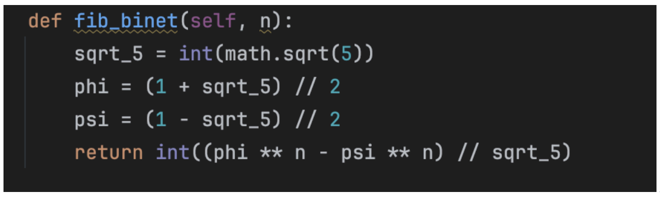
Algorithm Description:

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

```
Fibonacci(n):
    phi <- (1 + sqrt(5))
    phi1 <-(1 - sqrt(5))
    return pow(phi, n)- pow(phi1, n)/(pow(2, n)*sqrt(5))
```

Implementation:

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

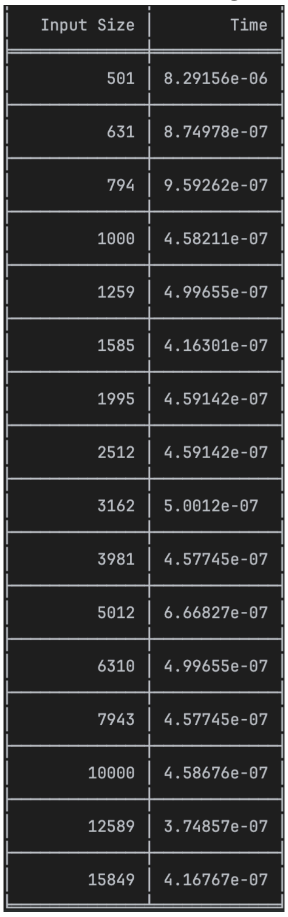


```
def fib_binet(self, n):
    sqrt_5 = int(math.sqrt(5))
    phi = (1 + sqrt_5) // 2
    psi = (1 - sqrt_5) // 2
    return int((phi ** n - psi ** n) // sqrt_5)
```

Figure 14 Fibonacci Binet Formula Method in Python

Results:

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



Input Size	Time
501	8.29156e-06
631	8.74978e-07
794	9.59262e-07
1000	4.58211e-07
1259	4.99655e-07
1585	4.16301e-07
1995	4.59142e-07
2512	4.59142e-07
3162	5.0012e-07
3981	4.57745e-07
5012	6.66827e-07
6310	4.99655e-07
7943	4.57745e-07
10000	4.58676e-07
12589	3.74857e-07
15849	4.16767e-07

Figure 15 Fibonacci Binet Formula Method results

And as shown in its performance graph,

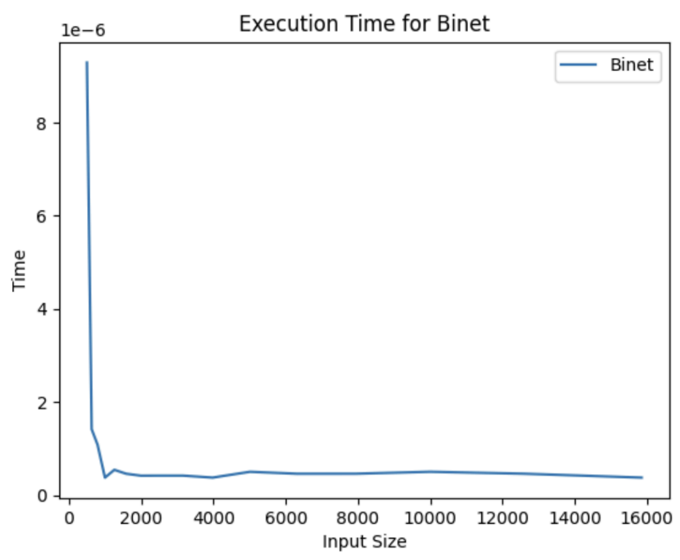


Figure 16 Fibonacci Binet formula Method

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:

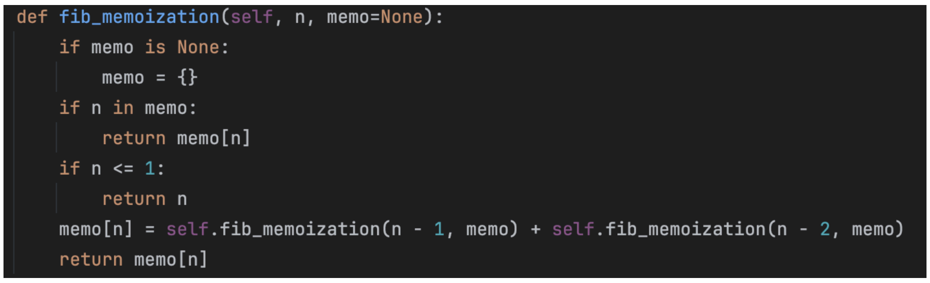
The Memoization Method for calculating Fibonacci numbers involves storing the results of already computed Fibonacci numbers in memory to avoid redundant calculations. When computing a Fibonacci number, it first checks if the result is already stored in memory. If yes, it returns the stored value; otherwise, it computes the Fibonacci number recursively and stores the result in memory for future use.

Algorithm Description:

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

```
Fibonacci(n):
    if memo contains key n:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = Fibonacci(n - 1) + Fibonacci(n - 2)
    return memo[n]
```

Implementation:

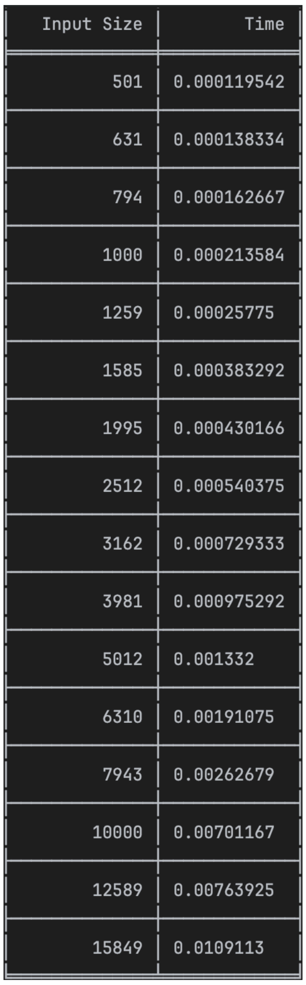


```python
def fib_memoization(self, n, memo=None):
    if memo is None:
        memo = {}
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = self.fib_memoization(n - 1, memo) + self.fib_memoization(n - 2, memo)
    return memo[n]
```

Figure 17 Fibonacci Memoization Method in Python

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	Time
501	0.000119542
631	0.000138334
794	0.000162667
1000	0.000213584
1259	0.00025775
1585	0.000383292
1995	0.000430166
2512	0.000540375
3162	0.000729333
3981	0.000975292
5012	0.001332
6310	0.00191075
7943	0.00262679
10000	0.00701167
12589	0.00763925
15849	0.0109113

Figure 18 Fibonacci Memoization Method results

And as shown in its performance graph,

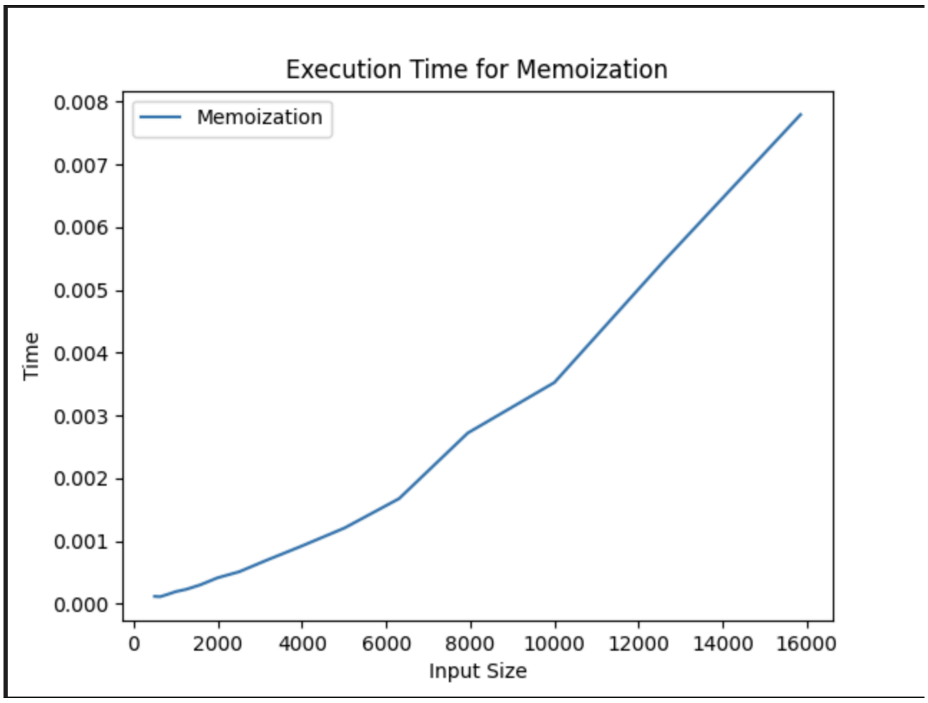


Figure 19 Fibonacci Memoization Method

CONCLUSION

Through the empirical analysis conducted in this study, six distinct algorithms for determining the Fibonacci n-th term have been evaluated in terms of their efficiency and applicability.

Recursive Method: Despite its simplicity, the Recursive method demonstrates exponential time complexity, making it suitable only for small input sizes. Beyond a certain threshold, its execution time grows rapidly, rendering it impractical for larger Fibonacci terms.

Iterative Method: The Iterative method offers a more efficient alternative to the Recursive approach, with linear time complexity. Its iterative nature allows for precise calculations without the overhead of function calls, making it suitable for a wide range of Fibonacci numbers.

Dynamic Programming Method: Utilizing a bottom-up approach and storing previously computed results, the Dynamic Programming method achieves linear time complexity, providing accurate Fibonacci numbers efficiently. This method proves to be highly effective for large Fibonacci terms, outperforming the Recursive approach significantly.

Matrix Power Method: By leveraging matrix multiplication, the Matrix Power method offers another efficient solution for computing Fibonacci numbers. Despite being slightly slower than the Dynamic Programming method, it still exhibits linear time complexity and provides accurate results for a wide range of input sizes.

Binet Formula Method: The Binet Formula method, based on the Golden Ratio formula, demonstrates nearly constant time complexity. While it offers fast computation for small Fibonacci terms, its practicality is limited by rounding errors with larger numbers, particularly in implementations using floating-point arithmetic.

Memoization Method: The Memoization method, employing recursive calls with memory storage, achieves performance similar to the Iterative method. By avoiding redundant calculations through memoization, it provides accurate Fibonacci numbers efficiently, making it a viable alternative for various input sizes.

In conclusion, each algorithm offers distinct advantages and limitations in terms of time complexity and accuracy. The choice of algorithm should be tailored to the specific requirements of the Fibonacci computation task, considering factors such as input size, precision, and computational resources available. This study provides valuable insights into the performance characteristics of each algorithm, laying the groundwork for informed decision-making in Fibonacci number computation.