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-222 Martiniuc Artiom

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2023

TABLE OF CONTENTS

Tasks: 3
Theoretical Notes: 3
Introduction: 4
Comparison Metric: 4
Input Format: 4
IMPLEMENTATION 5
Recursive Method: 5
Dynamic Programming Method: 6
Matrix Power Method: 8
Binet Formula Method: 11
Iterative method: 12
Memoization Method: 14
CONCLUSION 16

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 native 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.

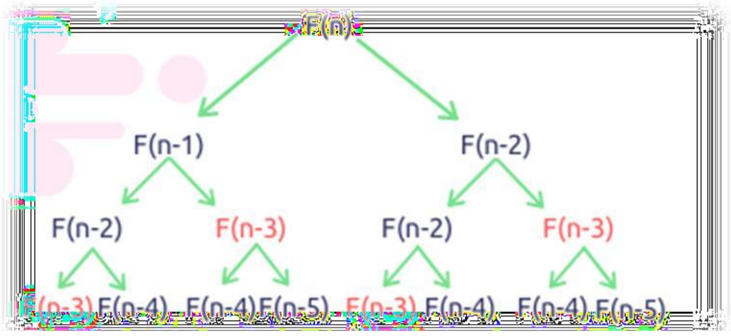


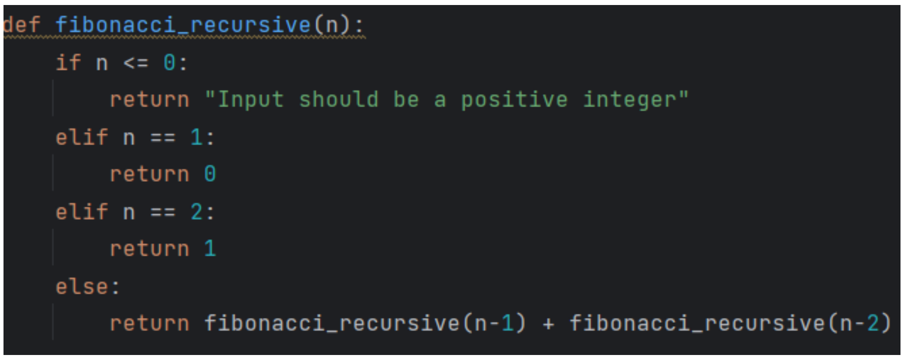
Figure 1 Fibonacci Recursion

Algorithm Description:

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

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

Implementation:

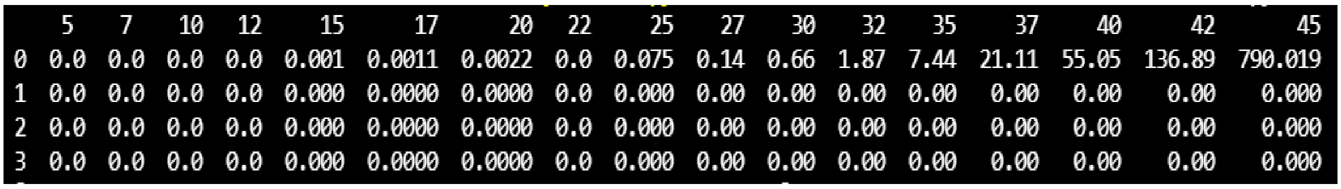


```python
def fibonacci_recursive(n):
    if n <= 0:
        return "Input should be a positive integer"
    elif n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
```

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:



```
    5    7   10   12     15      17      20   22     25    27    30    32    35     37     40      42       45
0 0.0  0.0  0.0  0.0  0.001  0.0011  0.0022  0.0  0.075  0.14  0.66  1.87  7.44  21.11  55.05  136.89  790.019
1 0.0  0.0  0.0  0.0  0.000  0.0000  0.0000  0.0  0.000  0.00  0.00  0.00  0.00   0.00   0.00    0.00    0.000
2 0.0  0.0  0.0  0.0  0.000  0.0000  0.0000  0.0  0.000  0.00  0.00  0.00  0.00   0.00   0.00    0.00    0.000
3 0.0  0.0  0.0  0.0  0.000  0.0000  0.0000  0.0  0.000  0.00  0.00  0.00  0.00   0.00   0.00    0.00    0.000
```

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.

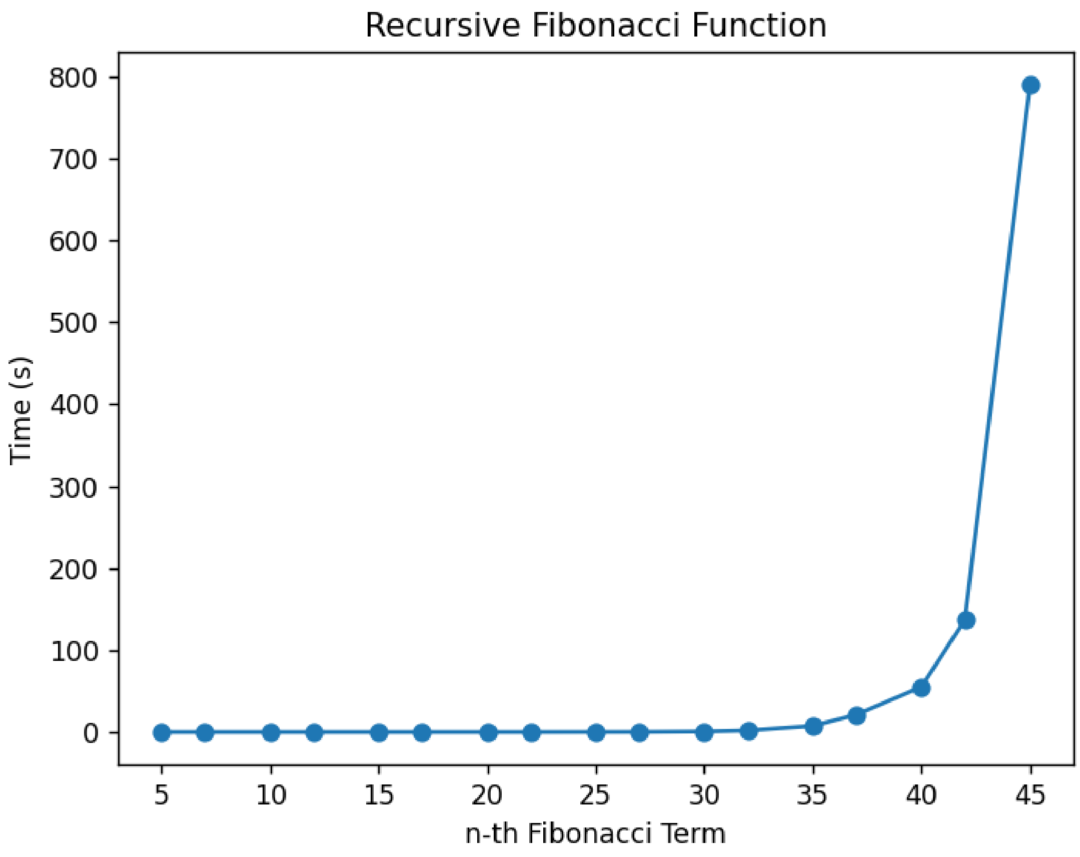


Figure 4 Graph of Recursive Fibonacci Function

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:

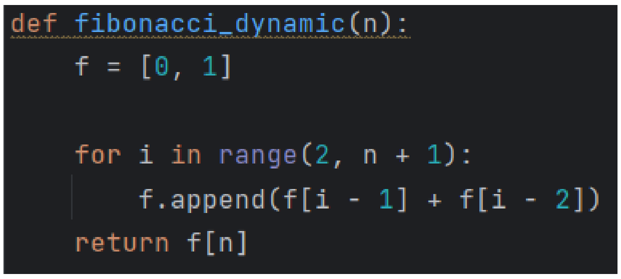
The Dynamic Programming method, similar to the recursive method, takes the straightforward approach of calculating the n-th term. However, instead of calling the function upon itself, from top down it operates based on an array data structure that holds the previously computed terms, eliminating the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

Algorithm Description:

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

```
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:



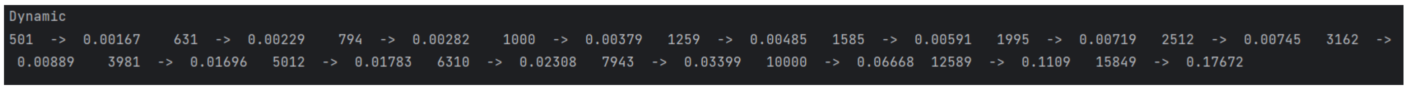
```python
def fibonacci_dynamic(n):
    f = [0, 1]

    for i in range(2, n + 1):
        f.append(f[i - 1] + f[i - 2])
    return f[n]
```

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:



```
Dynamic
501  ->  0.00167   631  ->  0.00229   794  ->  0.00282   1000  ->  0.00379   1259  ->  0.00485   1585  ->  0.00591   1995  ->  0.00719   2512  ->  0.00745   3162  ->
 0.00889   3981  ->  0.01696   5012  ->  0.01783   6310  ->  0.02308   7943  ->  0.03399   10000  ->  0.06668  12589  ->  0.1109   15849  ->  0.17672
```

Figure 6 Fibonacci DP Results

With the Dynamic Programming Method showing excellent results with a time complexity denoted in a corresponding graph of T(n)

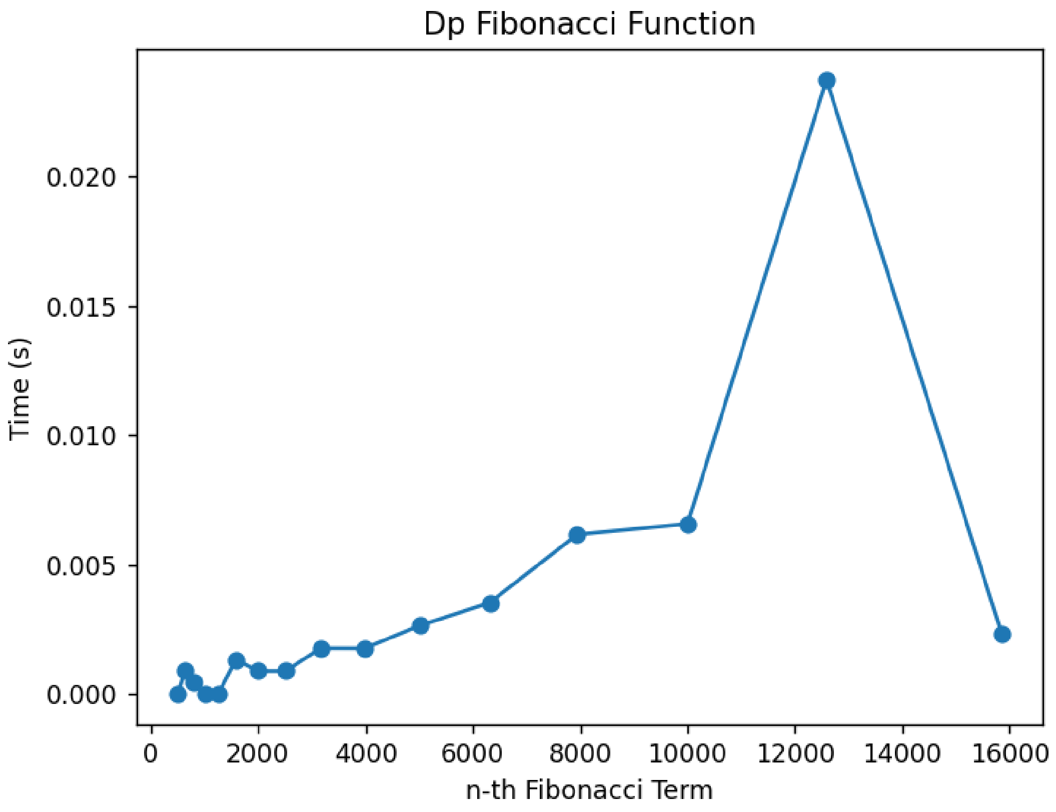


Figure 7 Fibonacci DP Graph

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:

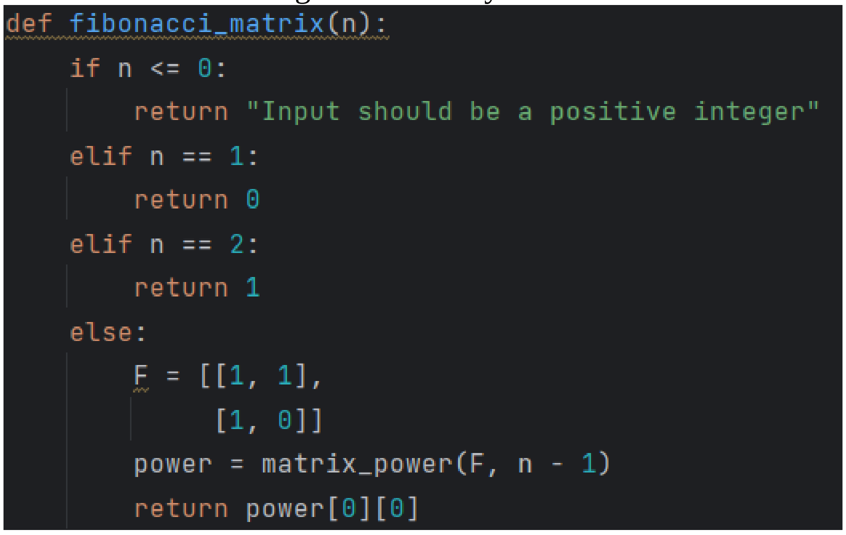
$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n\begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

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:

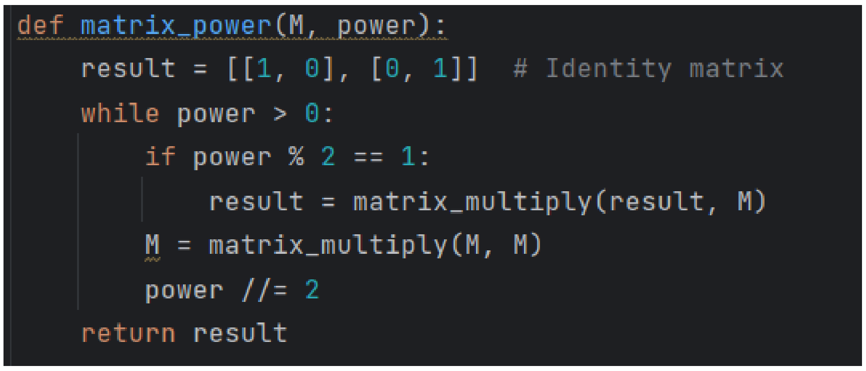
The implementation of the driving function in Python is as follows:



```python
def fibonacci_matrix(n):
    if n <= 0:
        return "Input should be a positive integer"
    elif n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        F = [[1, 1],
             [1, 0]]
        power = matrix_power(F, n - 1)
        return power[0][0]
```

Figure 8 Fibonacci Matrix Power Method in Python

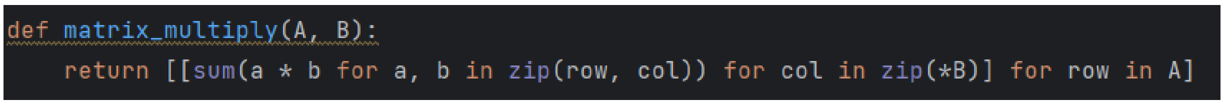
With additional miscellaneous functions:



```python
def matrix_power(M, power):
    result = [[1, 0], [0, 1]]  # Identity matrix
    while power > 0:
        if power % 2 == 1:
            result = matrix_multiply(result, M)
        M = matrix_multiply(M, M)
        power //= 2
    return result
```

Figure 9 Power Function Python

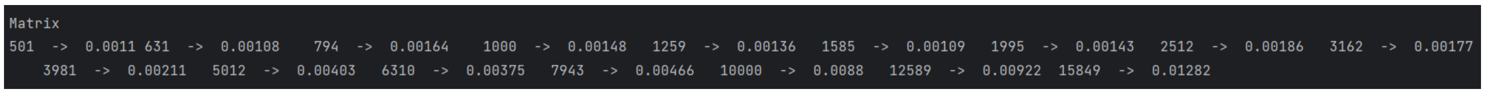
Where the power function (Figure 8) handles the part of raising the Matrix to the power n, while the multiplying function (Figure 9) handles the matrix multiplication with itself.



```python
def matrix_multiply(A, B):
    return [[sum(a * b for a, b in zip(row, col)) for col in zip(*B)] for row in A]
```

Figure 10 Multiply 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:



```
Matrix
501   ->  0.0011 631   ->  0.00108    794  ->  0.00164    1000  ->  0.00148   1259  ->  0.00136   1585  ->  0.00109   1995  ->  0.00143   2512  ->  0.00186   3162  ->  0.00177
    3981  ->  0.00211   5012  ->  0.00403   6310  ->  0.00375   7943  ->  0.00466   10000  ->  0.0088   12589  ->  0.00922  15849  ->  0.01282
```

Figure 11 Matrix Method Fibonacci Results

With the native Matrix method still performing pretty well, with the form f the graph indicating a pretty solid T(n) time complexity.

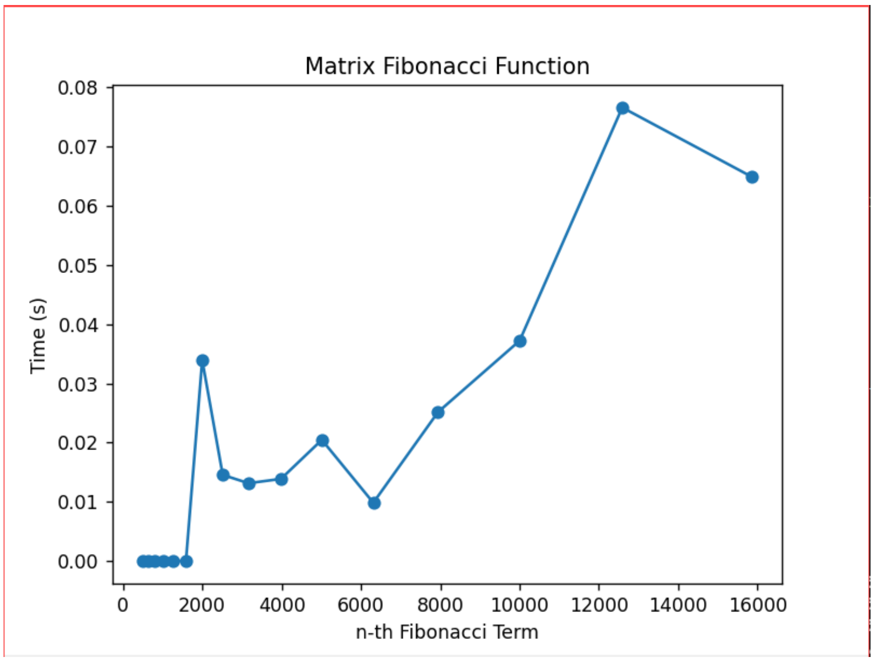


Figure 12 Matrix Method Fibonacci graph

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.

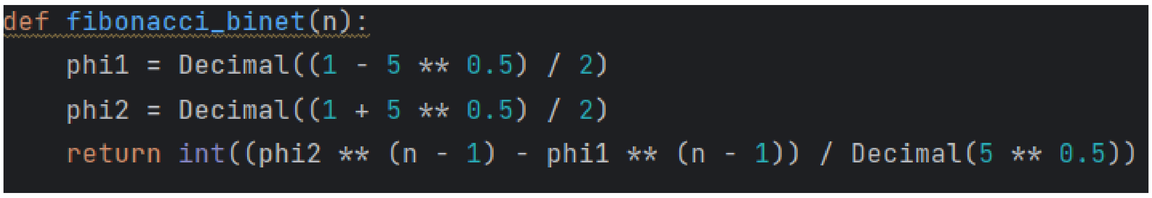
Algorithm Description:

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

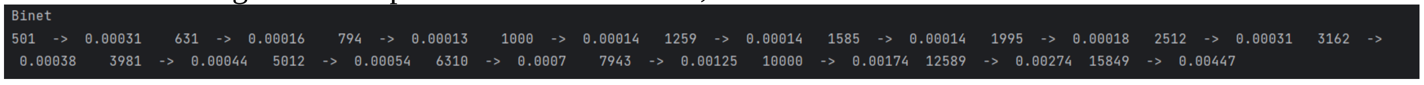


```python
def fibonacci_binet(n):
    phi1 = Decimal((1 - 5 ** 0.5) / 2)
    phi2 = Decimal((1 + 5 ** 0.5) / 2)
    return int((phi2 ** (n - 1) - phi1 ** (n - 1)) / Decimal(5 ** 0.5))
```

Figure 13 Fibonacci Binet Formula Method in Python

Results:

Although the most performant with its time, as shown below



```
Binet
501  ->  0.00031    631  ->  0.00016    794  ->  0.00013    1000  ->  0.00014   1259  ->  0.00014   1585  ->  0.00014   1995  ->  0.00018   2512  ->  0.00031   3162  ->
 0.00038    3981  ->  0.00044    5012  ->  0.00054    6310  ->  0.0007    7943  ->  0.00125    10000  ->  0.00174   12589  ->  0.00274   15849  ->  0.00447
```

Figure 14 Fibonacci Binet Formula Method results

And as shown in its performance graph,

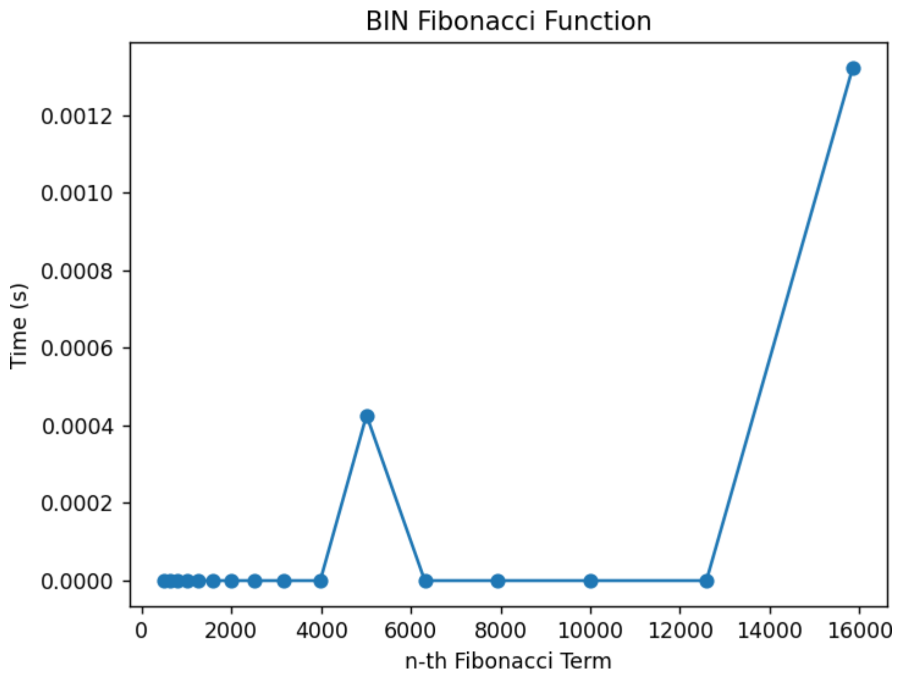


Figure 15 Fibonacci Binet formula Method

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

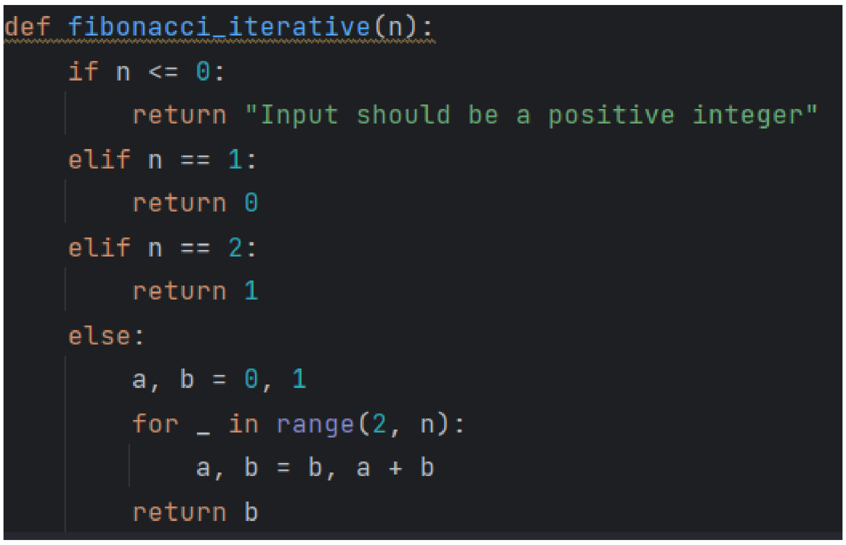
Iterative method:

This method uses a loop to calculate the Fibonacci sequence, which is more efficient than the recursive method. It keeps track of the current and next numbers in the sequence, updating these at each step of the loop.

```
fibonacci_iterative(n):
        SET a = 0
        SET b = 1
        FOR i FROM 2 TO n-1
            SET temp = b
            SET b = a + b
            SET a = temp
        END FOR
        RETURN b
```

Implementation:

This is implementation of function in Python.

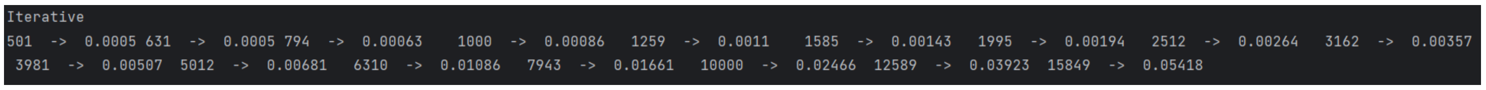


```python
def fibonacci_iterative(n):
    if n <= 0:
        return "Input should be a positive integer"
    elif n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        a, b = 0, 1
        for _ in range(2, n):
            a, b = b, a + b
        return b
```

Figure 16 Fibonacci Iterative Method in Python

Results:

Below we can see the results of time complexity of this method.



```
Iterative
501  ->  0.0005 631  ->  0.0005 794  ->  0.00063    1000  ->  0.00086   1259  ->  0.0011    1585  ->  0.00143   1995  ->  0.00194   2512  ->  0.00264   3162  ->  0.00357
 3981  ->  0.00507  5012  ->  0.00681   6310  ->  0.01086   7943  ->  0.01661   10000  ->  0.02466  12589  ->  0.03923  15849  ->  0.05418
```

Figure 17 Fibonacci Iterative Method results

And also, a graph:

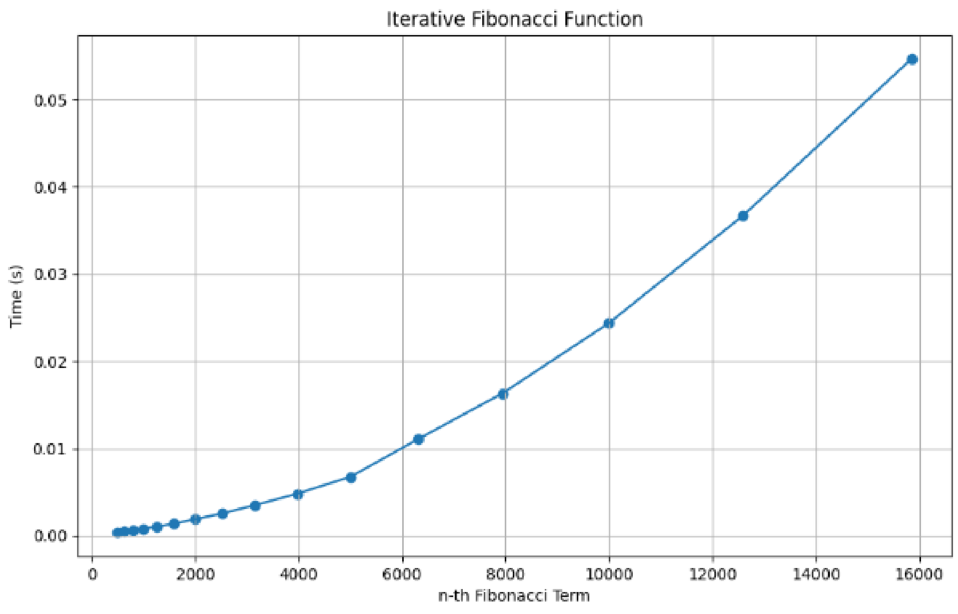


Figure 18 Fibonacci Iterative Method

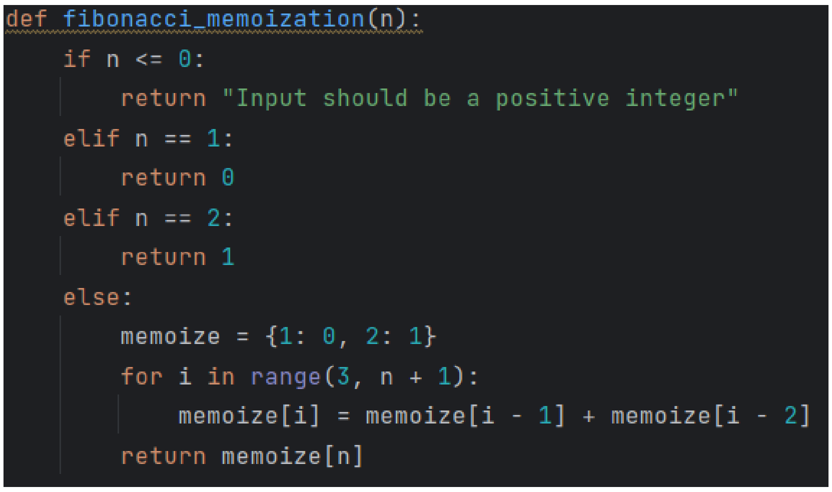
h

Memoization Method:

This method also uses recursion but with a technique called memoization to avoid repeated calculations. It stores the Fibonacci numbers calculated so far in a dictionary and uses them directly instead of recalculating them.

```
fibonacci_memoization(n):
        SET memoize = {1: 0, 2: 1}
        FOR i FROM 3 TO n
            SET memoize[i] = memoize[i - 1] + memoize[i - 2]
        END FOR
        RETURN memoize[n]
```

Implementation:

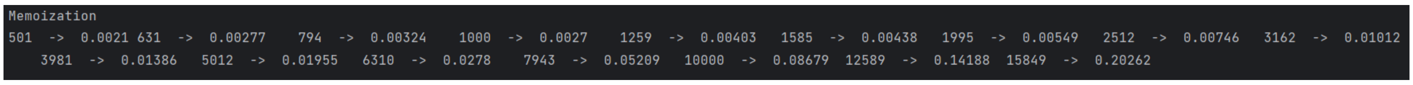


```python
def fibonacci_memoization(n):
    if n <= 0:
        return "Input should be a positive integer"
    elif n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        memoize = {1: 0, 2: 1}
        for i in range(3, n + 1):
            memoize[i] = memoize[i - 1] + memoize[i - 2]
        return memoize[n]
```

Figure 19 Fibonacci Memoization Method in Python

Results:

Below is shown the time complexity for memoization method.



```
Memoization
501   ->  0.0021 631   ->  0.00277    794  ->  0.00324    1000  ->  0.0027    1259  ->  0.00403    1585  ->  0.00438    1995  ->  0.00549    2512  ->  0.00746    3162  ->  0.01012
    3981  ->  0.01386    5012  ->  0.01955    6310  ->  0.0278    7943  ->  0.05209    10000  ->  0.08679   12589  ->  0.14188   15849  ->  0.20262
```

Figure 20 Fibonacci Memoization Method results

And graphically represented:

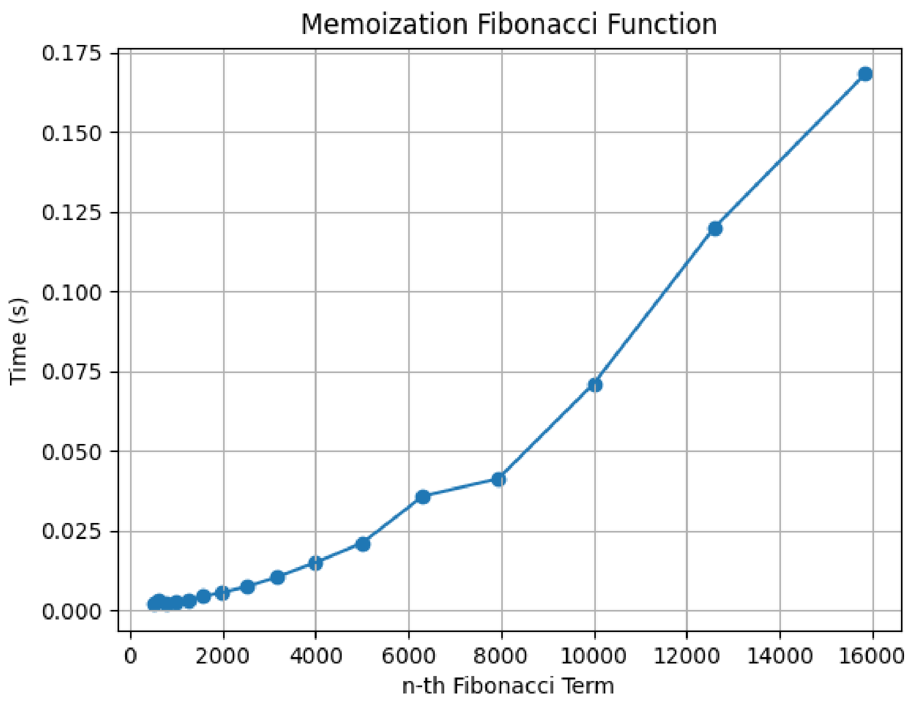


Figure 21 Fibonacci Memoization Method

CONCLUSION

Through Empirical Analysis, within this paper, six 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.

Iterative Method: This method uses a loop to calculate the Fibonacci sequence, which is more efficient than the recursive method. It keeps track of the current and next numbers in the sequence, updating these at each step of the loop.

The Recursive method, being the easiest to write, but also the most difficult to execute with an exponential time complexity, can be used for smaller order numbers, such as numbers of order up to 30 with no additional strain on the computing machine and no need for testing of patience.

The Binet method, the easiest to execute with an almost constant time complexity, could be used when computing numbers of order up to 80, after the recursive method becomes unfeasible. However, its results are recommended to be verified depending on the language used, as there could rounding errors due to its formula that uses the Golden Ratio.

The Dynamic Programming and Matrix Multiplication Methods can be used to compute Fibonacci numbers further then the ones specified above, both of them presenting exact results and showing a linear complexity in their naivety that could be, with additional tricks and optimisations, reduced to logarithmic.

The Memoization method provides a significant improvement over the Recursive method by storing previously computed values in a cache, thereby avoiding redundant calculations. This approach drastically reduces the time complexity, making it suitable for larger Fibonacci numbers.

The Iterative method offers a straightforward and efficient solution. By keeping track of only the current and next numbers in the sequence, it avoids the overhead associated with function calls in the Recursive method.