

Laboratory work 1:
Study and Empirical Analysis of Algorithms for
Determining
Fibonacci N-th Term

Elaborated:
st. gr. FAF-233

Chirtoaca Liviu

Verified:
asist. univ.

Fiștic Cristofor

TABLE OF CONTENTS

Contents

ALGORITHM ANALYSIS.....	3
Objective	3
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
The Memoization Method	12
Matrix Exponentiation with Modulo Arithmetics	14
Continued Fraction Approximation for Fibonacci Numbers.	17
CONCLUSION.....	19

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.

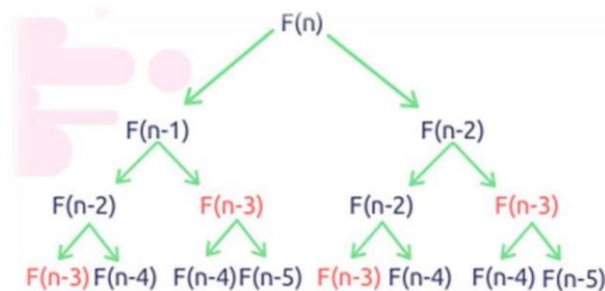


Figure 1 Fibonacci Recursion

Algorithm Description:

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

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

Implementation:

```
def fibonacci(x):
    if x <= 1:
        return x
    else:
        return fibonacci(x-1)+ fibonacci(x-2)
```

Figure 2 Fibonacci recursion in Python

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.

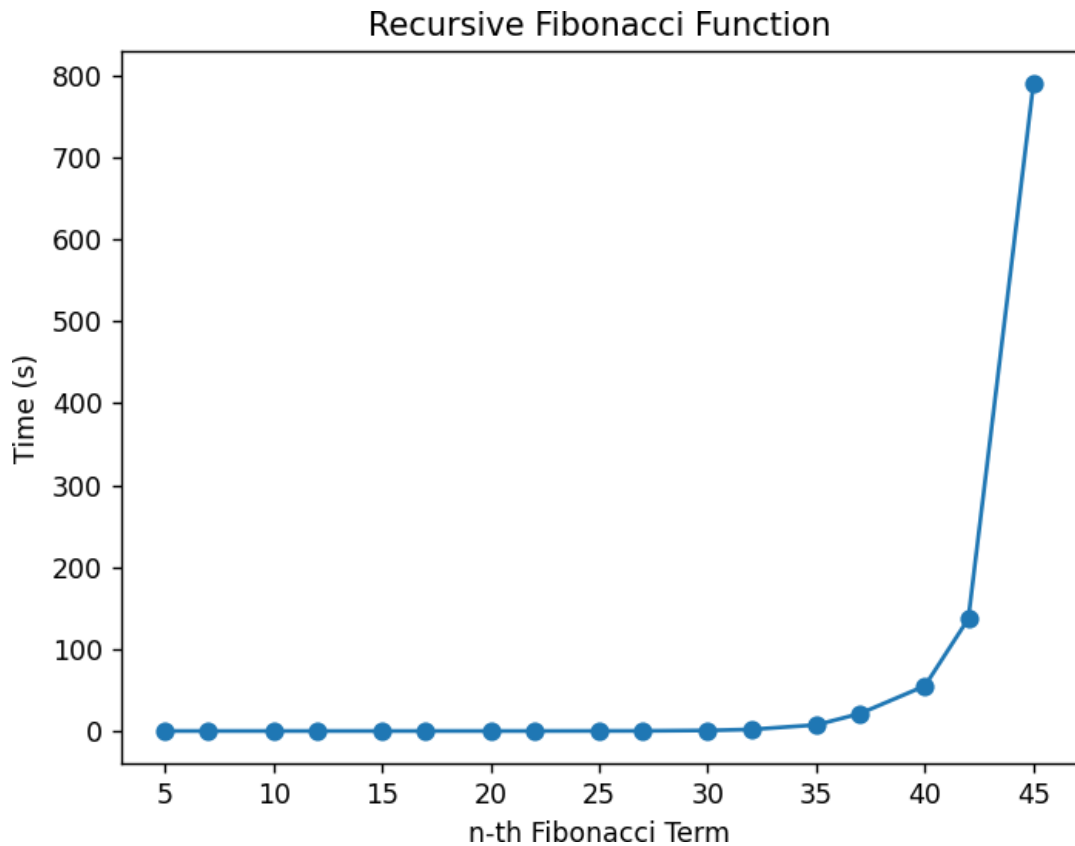


Figure 4 Graph of Recursive Fibonacci Function

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:

```
def f(x):
    l1 = [0, 1]

    for i in range(2, x + 1):
        l1.append(l1[i-1] + l1[i-2])

    return l1[x]
```

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:

	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
0	0.0	0.0	0.000726	0.0	0.000758	0.0007	0.000360	0.003646	0.001101	0.001459	0.001831	0.002550	0.004351	0.005489	0.019695	0.014626
1	0.0	0.0	0.000000	0.0	0.000000	0.0000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001074	0.000000	0.000727
2	0.0	0.0	0.000000	0.0	0.000000	0.0000	0.033937	0.014543	0.013128	0.013862	0.020456	0.009831	0.025155	0.037205	0.076604	0.064923

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),

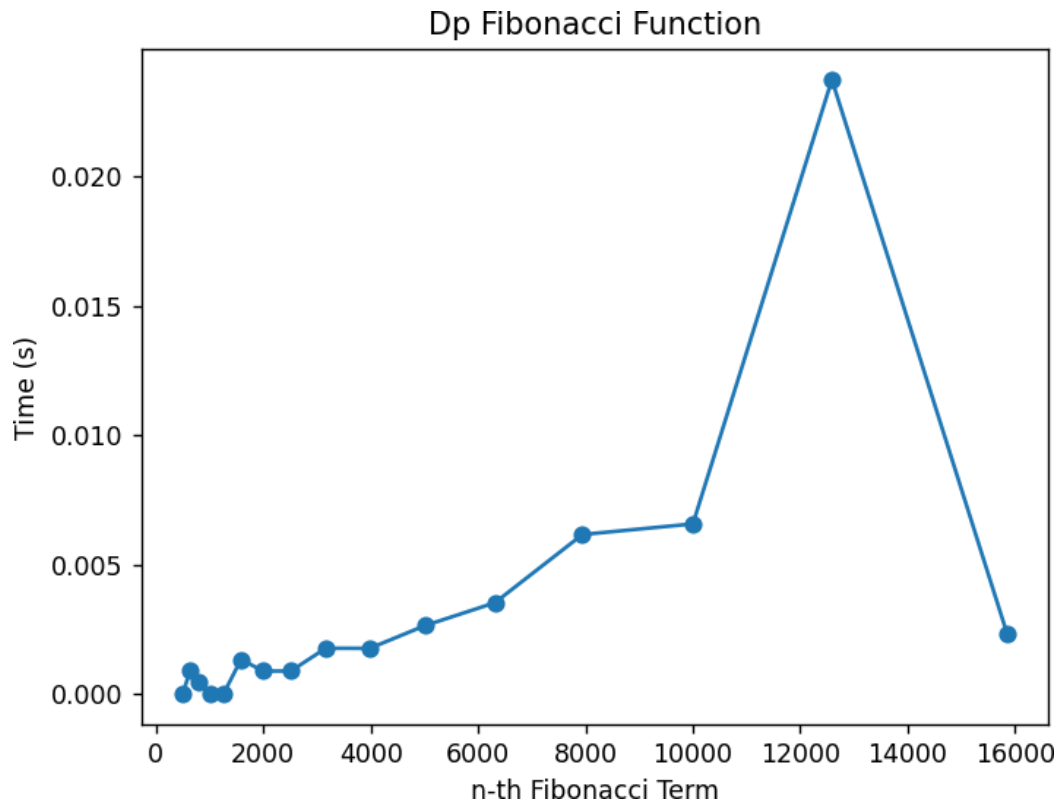


Figure 7 Fibonacci DP Graph

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:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_2 \\ F_3 \end{pmatrix}$$

Which turns into the general:

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

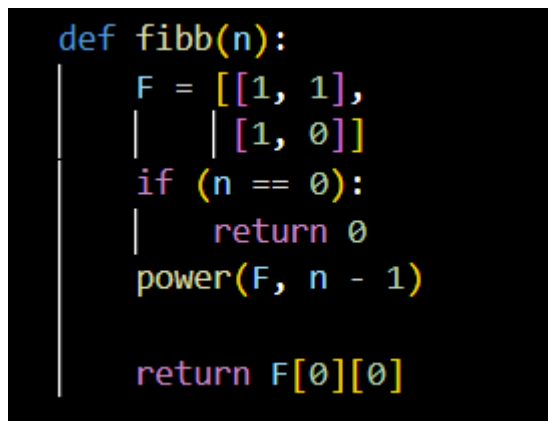
This can be displayed as follow:

Fibonacci(n) :

```
F<- []  
vec <- [[0], [1]]  
Matrix <- [[0, 1],[1, 1]]  
F <-power(Matrix, n)  
F <- F * vec  
Return F[0][0]
```

Implementation:

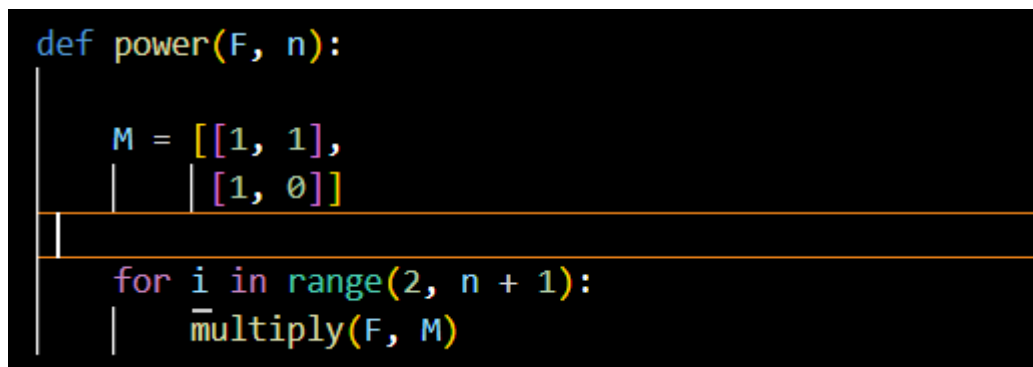
The implementation of the driving function in Python is as follows:

A screenshot of a code editor with a dark background and light-colored text. The code defines a function 'fibb(n)'. Inside the function, a matrix 'F' is defined as a 2x2 list of lists: [[1, 1], [1, 0]]. An 'if' statement checks if 'n' is equal to 0; if true, it returns 0. Otherwise, it calls 'power(F, n - 1)'. Finally, it returns 'F[0][0]'.

```
def fibb(n):  
    F = [[1, 1],  
         [1, 0]]  
    if (n == 0):  
        return 0  
    power(F, n - 1)  
  
    return F[0][0]
```

Figure 8 Fibonacci Matrix Power Method in Python

With additional miscellaneous functions:

A screenshot of a code editor with a dark background and light-colored text. The code defines a function 'power(F, n)'. Inside the function, a matrix 'M' is defined as a 2x2 list of lists: [[1, 1], [1, 0]]. A 'for' loop iterates from 2 to 'n + 1', and in each iteration, it calls 'multiply(F, M)'.

```
def power(F, n):  
    M = [[1, 1],  
         [1, 0]]  
  
    for i in range(2, n + 1):  
        multiply(F, M)
```

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.

```
def multiply(F, M):
    x = (F[0][0] * M[0][0] +
        | F[0][1] * M[1][0])
    y = (F[0][0] * M[0][1] +
        | F[0][1] * M[1][1])
    z = (F[1][0] * M[0][0] +
        | F[1][1] * M[1][0])
    w = (F[1][0] * M[0][1] +
        | F[1][1] * M[1][1])

    F[0][0] = x
    F[0][1] = y
    F[1][0] = z
    F[1][1] = w
```

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:

	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
0	0.0	0.0	0.000726	0.0	0.000758	0.0007	0.000360	0.003646	0.001101	0.001459	0.001831	0.002550	0.004351	0.005489	0.019695	0.014626
1	0.0	0.0	0.000000	0.0	0.000000	0.0000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001074	0.000000	0.000727
2	0.0	0.0	0.000000	0.0	0.000000	0.0000	0.033937	0.014543	0.013128	0.013862	0.020456	0.009831	0.025155	0.037205	0.076604	0.064923

Figure 11 Matrix Method Fibonacci Results

With the naïve Matrix method (indicated in last row, row[2]), although being slower than the Binet and Dynamic Programming one, still performing pretty well, with the form of the graph indicating a pretty solid $T(n)$ time complexity.

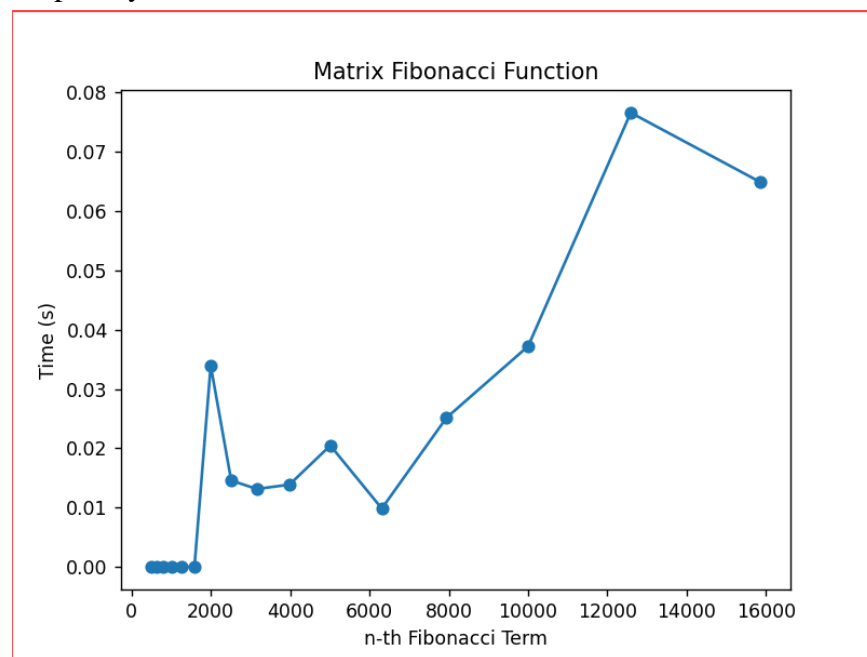


Figure 12 Matrix Method Fibonacci graph

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:

```
def fib(x):
    ctx = Context(prec=60, rounding=ROUND_HALF_EVEN)
    phi = Decimal((1 + Decimal(5**(1/2))))
    phi2 = Decimal((1 - Decimal(5**(1/2))))

    return int((ctx.power(phi, Decimal(x)) - ctx.power(phi2, Decimal(x))) / (2**x * Decimal(5**(1/2))))
```

Figure 13 Fibonacci Binet Formula Method in Python

Results:

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

	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
0	0.0	0.0	0.000726	0.0	0.000758	0.0007	0.000360	0.003646	0.001101	0.001459	0.001831	0.002550	0.004351	0.005489	0.019695	0.014626
1	0.0	0.0	0.000000	0.0	0.000000	0.0000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001074	0.000000	0.000727
2	0.0	0.0	0.000000	0.0	0.000000	0.0000	0.033937	0.014543	0.013128	0.013862	0.020456	0.009831	0.025155	0.037205	0.076604	0.064923

Figure 14 Fibonacci Binet Formula Method results

And as shown in its performance graph,

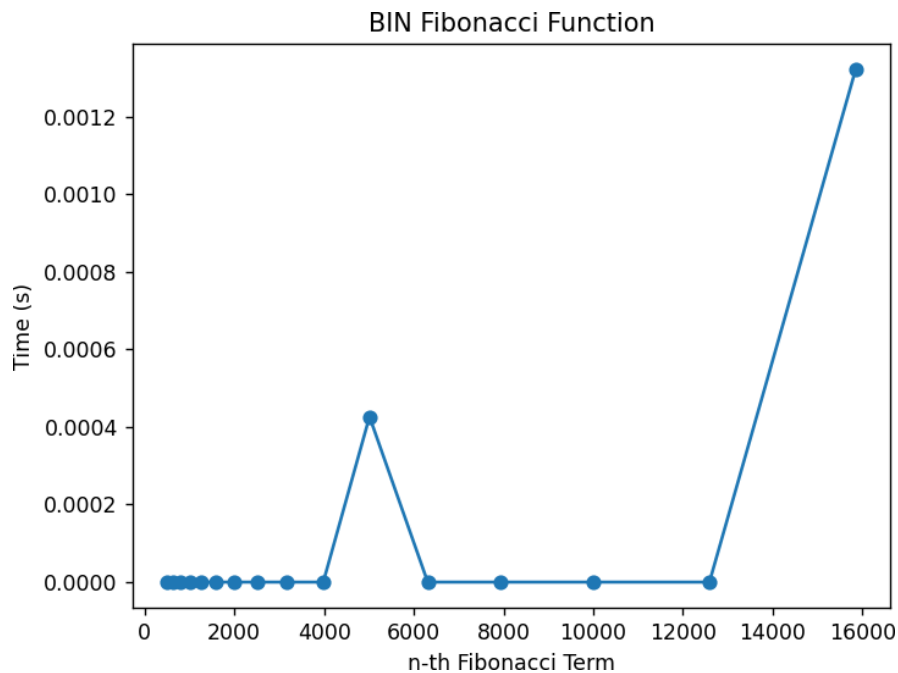


Figure 15 Fibonacci Binet formula Method

The Binet Formula Function is not accurate enough to be considered within the analysed limits and is recommended to be used for Fibonacci terms up to 80. At least in its naïve form in python, as further modification and change of language may extend its usability further.

The Memoization Method

Memoization is an optimization technique that stores the results of expensive function calls and reuses them when the same inputs occur again. In Fibonacci calculation, it uses recursion while caching previously computed Fibonacci numbers to avoid redundant calculations. This reduces the time complexity from $O(n^2)$ to $O(n)$, significantly improving performance. However, it requires additional memory to store the cached results, making the space complexity $O(n)$.

Algorithm Description:

```
if (n <= 1) return n;
    if (memo.ContainsKey(n)) return memo[n]; // If the value is already computed, return it

    // Compute and store in the cache
    memo[n] = FibonacciMemoization(n - 1) + FibonacciMemoization(n - 2);
    return memo[n];
```

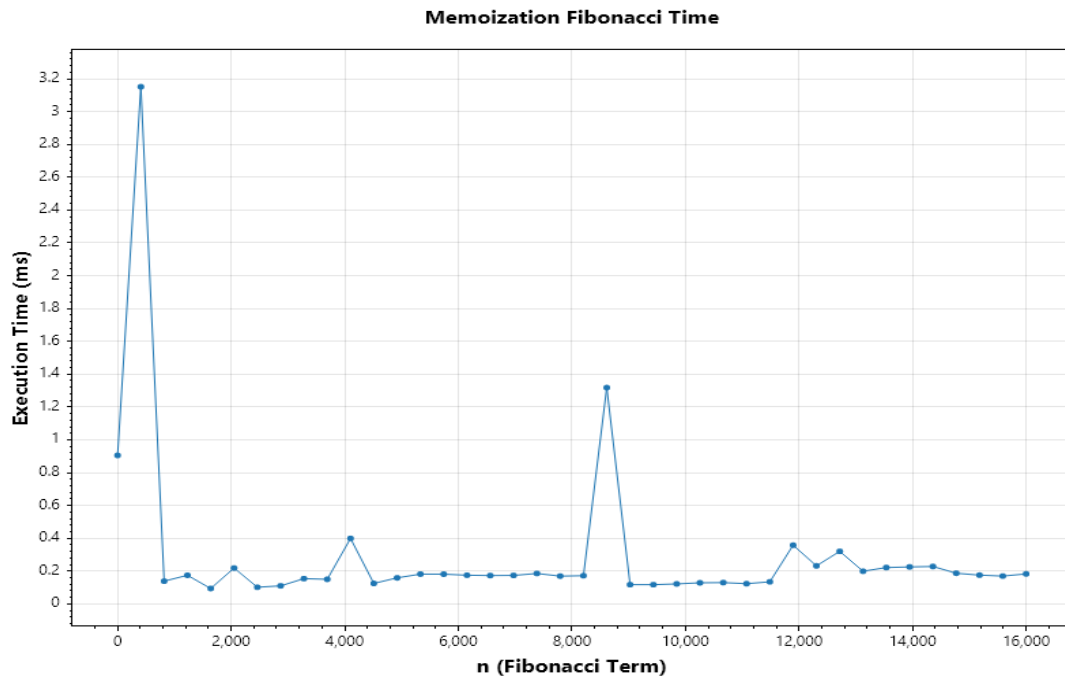
Implementation:

```
if (n <= 1) return n;  
if (memo.ContainsKey(n)) return memo[n];  
  
memo[n] = FibonacciMemoization(n - 1) + FibonacciMemoization(n - 2);  
return memo[n];
```

Results:

n	Memoization (ms)				
1	0.7655	6154	0.1381	10667	0.1627
411	3.7686	6564	0.1252	11077	0.1546
821	0.2481	6974	0.1501	11487	0.1428
1231	0.2911	7385	0.1494	11897	0.1726
1641	0.1441	7795	0.1391	12307	0.2609
2052	0.3004	8205	0.1453	12718	0.1743
2462	0.2091	8615	1.3496	13128	0.1782
2872	0.2181	9026	0.1692	13538	0.1726
3282	0.1452	9436	0.1606	13948	0.1826
3693	0.1696	9846	0.1601	14359	0.1752
4103	0.3753	10256	0.1582	14769	0.1917
4513	0.1480	10667	0.1627	15179	0.2147
4923	0.1367	11077	0.1546	15589	0.2680
		11487	0.1428	16000	0.1427
		11897	0.1726		

And the graph looks like this:



Matrix Exponentiation with Modulo Arithmetics

This method uses matrix exponentiation to compute Fibonacci numbers efficiently, reducing the time complexity to $O(\log n)$. It works by representing the Fibonacci sequence as a matrix equation and repeatedly squaring the matrix to obtain the n -th Fibonacci number. Modular arithmetic is used to avoid overflow when dealing with large numbers, keeping all calculations within a predefined modulus (e.g., $10^9 + 7$). This approach is especially useful for computing large Fibonacci numbers while keeping the result within manageable bounds.

Algorithm Description:

```
const long MOD = 1000000007; // Prevents overflow for large numbers
```

```
if (n == 0) return 0;
```

```
long[,] F = { { 1, 1 }, { 1, 0 } };
```

```
PowerMod(F, n - 1, MOD);
```

```
return F[0, 0] % MOD;
```

```
if (n <= 1) return;
```

```
long[,] M = { { 1, 1 }, { 1, 0 } };
```

```
PowerMod(F, n / 2, mod);
```

```
MultiplyMod(F, F, mod);
```

```
if (n % 2 != 0) MultiplyMod(F, M, mod);
```

```

long x = (F[0, 0] * M[0, 0] + F[0, 1] * M[1, 0]) % mod;
long y = (F[0, 0] * M[0, 1] + F[0, 1] * M[1, 1]) % mod;
long z = (F[1, 0] * M[0, 0] + F[1, 1] * M[1, 0]) % mod;
long w = (F[1, 0] * M[0, 1] + F[1, 1] * M[1, 1]) % mod;

F[0, 0] = x;
F[0, 1] = y;
F[1, 0] = z;
F[1, 1] = w;
}

```

Implementation:

```

static long FibonacciModulo(int n)
{
    const long MOD = 1000000007;
    if (n == 0) return 0;
    long[,] F = { { 1, 1 }, { 1, 0 } };
    PowerMod(F, n - 1, MOD);
    return F[0, 0] % MOD;
}

2 references
static void PowerMod(long[,] F, int n, long mod)
{
    if (n <= 1) return;
    long[,] M = { { 1, 1 }, { 1, 0 } };
    PowerMod(F, n / 2, mod);
    MultiplyMod(F, F, mod);
    if (n % 2 != 0) MultiplyMod(F, M, mod);
}

2 references
static void MultiplyMod(long[,] F, long[,] M, long mod)
{
    long x = (F[0, 0] * M[0, 0] + F[0, 1] * M[1, 0]) % mod;
    long y = (F[0, 0] * M[0, 1] + F[0, 1] * M[1, 1]) % mod;
    long z = (F[1, 0] * M[0, 0] + F[1, 1] * M[1, 0]) % mod;
    long w = (F[1, 0] * M[0, 1] + F[1, 1] * M[1, 1]) % mod;

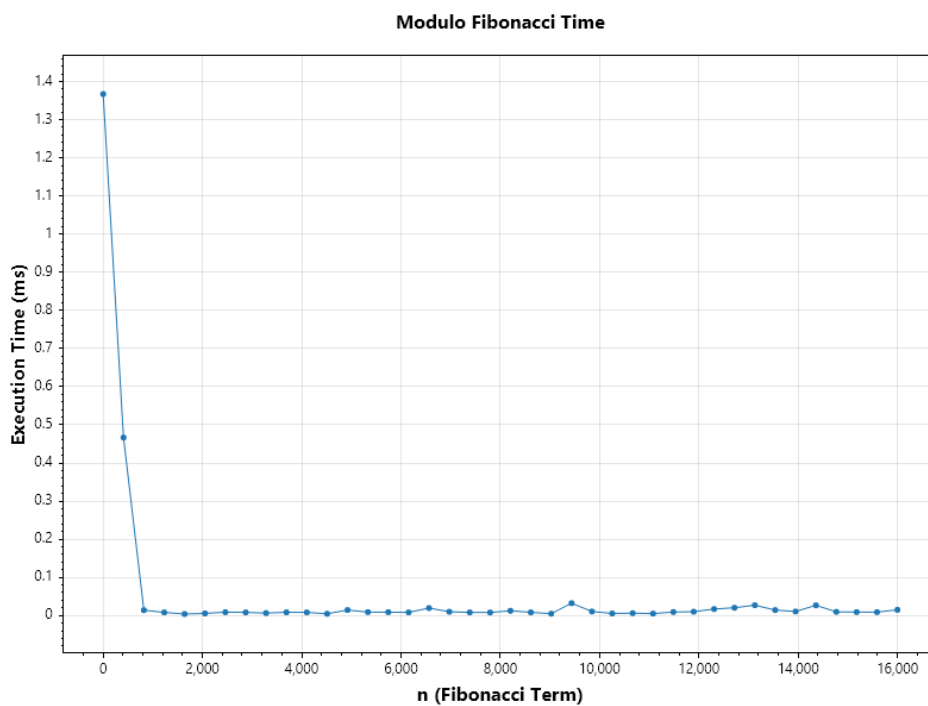
    F[0, 0] = x;
    F[0, 1] = y;
    F[1, 0] = z;
    F[1, 1] = w;
}

```

Results:

)	Modulo (
0.7601	0.0067	
0.3548	0.0071	
0.0144	0.0165	
0.0174	0.0130	
0.0156	0.0090	
0.0095	0.0137	
0.0138	0.0115	
0.0165	0.0061	
0.0128	0.0064	
0.0082	0.0179	
0.0106	0.0097	
0.0048	0.0081	
0.0090	0.0359	
0.0113	0.0073	
0.0121	0.0127	
0.0047	0.0163	
0.0047	0.0188	
0.0050	0.0116	
0.0164	0.0212	
0.0110	0.0178	
0.0067		
0.0071		
0.0165		
0.0130		
0.0090		
0.0137		
0.0115		
0.0061		
0.0064		
0.0179		
0.0097		
0.0081		
0.0359		

And the graph looks like this:



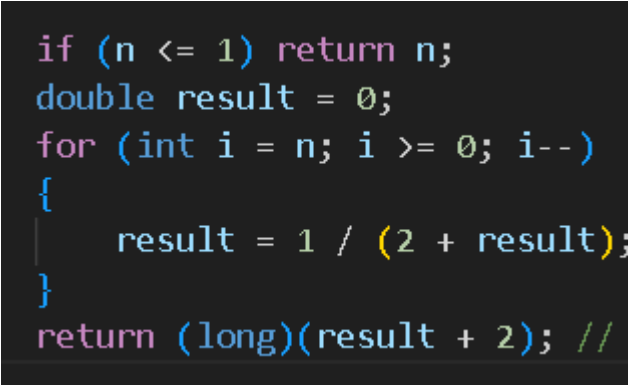
Continued Fraction Approximation for Fibonacci Numbers.

This method approximates Fibonacci numbers using a continued fraction representation. It involves iterating through a fraction sequence where each term is derived from the previous one, starting with a base value. Although it is not as precise as other methods, it provides a simple and fast approximation for Fibonacci numbers, especially for smaller values of n . The result is refined through successive iterations to approximate the n -th Fibonacci number.

Algorithm Description:

```
if (n <= 1) return n;
double result = 0;
for (int i = n; i >= 0; i--)
{
    result = 1 / (2 + result);
}
return (long)(result + 2);
```

Implementation:

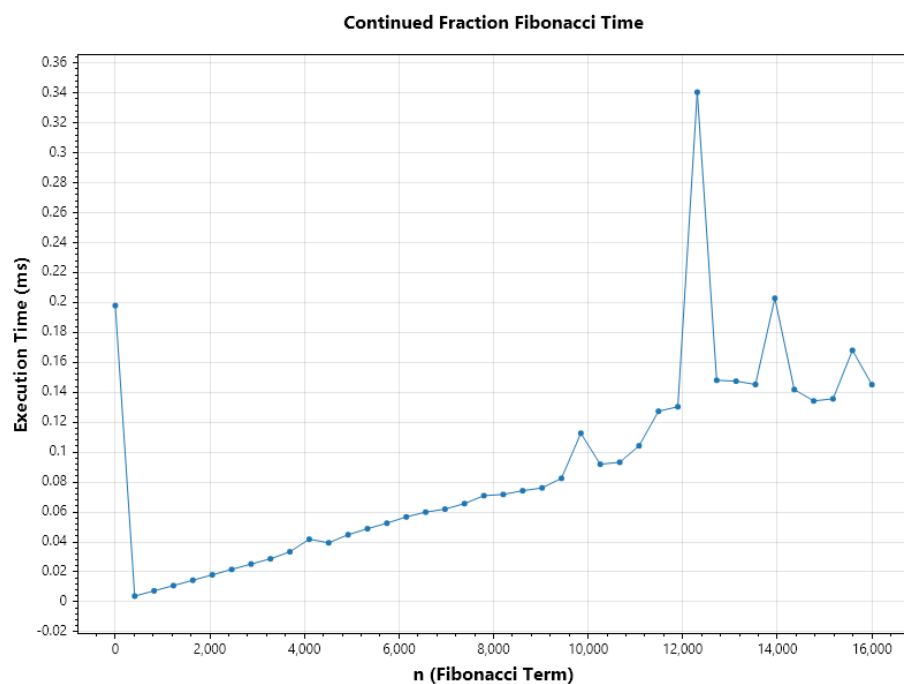


```
if (n <= 1) return n;
double result = 0;
for (int i = n; i >= 0; i--)
{
    result = 1 / (2 + result);
}
return (long)(result + 2); //
```

Results:

0.1102	0.0968
0.0031	0.1024
0.0082	0.1039
0.0160	0.0899
0.0185	0.0968
0.0223	0.1024
0.0244	0.1039
0.0287	0.1341
0.0307	0.1124
0.0336	0.0899
0.0374	0.0968
0.0395	0.1024
0.0448	0.0899
0.0661	0.0899
0.0521	0.0968
0.0558	0.1024
0.0573	0.0899
0.0609	0.0968
0.0650	0.1024
0.0755	0.0899
0.0717	0.0968
0.0927	0.0899
0.0828	0.0968
0.0890	0.1024
0.0862	0.1039
0.0936	0.1341
0.0899	0.1124
0.0968	0.1147
0.1024	0.1182
	0.1582

This is how the graph looks like:



CONCLUSION

Through Empirical Analysis, within this paper, four classes of methods have been tested in their efficiency at both their providing of accurate results, as well as at the time complexity required for their execution, to delimit the scopes within which each could be used, as well as possible improvements that could be further done to make them more feasible.

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

The Binet method, the easiest to execute with an almost constant time complexity, could be used when computing numbers of order up to 80, after the recursive method becomes unfeasible. However, its results are recommended to be verified depending on the language used, as there could be 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 than 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 optimizations, reduced to logarithmic.

For the Memoization method, its major advantage lies in reducing redundant calculations compared to the naive recursive method, making it more efficient for larger Fibonacci numbers. However, it still exhibits linear time complexity, requiring additional memory to store intermediate results.

The Modulo Fibonacci method, utilizing matrix exponentiation with modular arithmetic, performs very efficiently and can compute Fibonacci numbers in logarithmic time, making it ideal for extremely large numbers. This method ensures that values stay manageable by applying a modulus to avoid overflow, and it's particularly suitable for applications involving large Fibonacci numbers with constraints on memory and time

Lastly, the Continued Fraction method, though not as precise as other methods, provides a fast approximation of Fibonacci numbers, making it suitable for scenarios where performance is prioritized over absolute accuracy, especially for smaller Fibonacci terms. While the approximation may not be exact, it still offers a viable alternative when quick estimates are needed.

Each of these methods presents distinct trade-offs, and the choice of algorithm should depend on the specific requirements of the problem at hand, such as the desired accuracy and available computational resources.

Git link: https://github.com/D3adeYe69/AA_labs