

**Ministry of Education, Culture and Research of the Republic of
Moldova Technical University of Moldova
Department of Software and Automation Engineering**

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

Elaborated:
st. gr. FAF-223

Tofan Liviu

Verified:
asist. univ.

Fiștic Cristofor

Chișinău – 2024

TABLE OF CONTENTS

ALGORITHM ANALYSIS.....	3
Objective	3
Tasks:	3
Theoretical Notes:	3
Introduction:	4
Comparison Metric:	4
Input Format:.....	4
IMPLEMENTATION.....	5
Recursive Method:	5
Dynamic Programming Method:	7
Iterative Programming Method:	9
Matrix Power Method:	11
Binet Formula Method:	14
Matrix Exponentiation using Eigenvalues Method:.....	16
CONCLUSION.....	18

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 ($O(n)$)

Input Format:

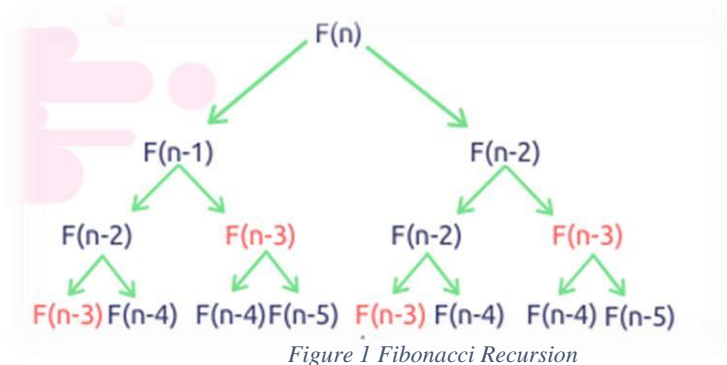
As input, each algorithm will receive two series of numbers that will contain the order of the Fibonacci terms being looked up. The first series will have a more limited scope, (5, 7, 10, 12, 15, 17, 20, 22, 25, 27, 30, 35, 37, 40, 42, 45), to accommodate the recursive method, while the second series will have a bigger scope to be able to compare the other algorithms between themselves (500, 650, 800, 1000, 1250, 1500, 2000, 2550, 3150, 4000, 5000, 6350, 8000, 10000, 12500, 16000) .

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. This follows the mathematical definition very closely but its performance is terrible: roughly $O(2^n)$. 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.



Algorithm Description:

The naive 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 Fibonacci1(n):
    if n <= 1:
        return n
    else:
        return Fibonacci1(n - 1) + Fibonacci1(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:

n	time(s)
5	0.000000
7	0.000000
10	0.000000
12	0.000000
15	0.000000
17	0.001033
20	0.000995
22	0.003059
25	0.013017
27	0.034211
30	0.144028
35	1.593820
37	4.170472
40	17.655913
42	46.257112
45	197.033727

Figure 3 Results for first set of inputs

In Figure 3 is represented the table of results for the first set of inputs. Starting from the 17th term of the Fibonacci sequence, the computation time grows rapidly, reaching a significantly slower pace at the last element, 45.

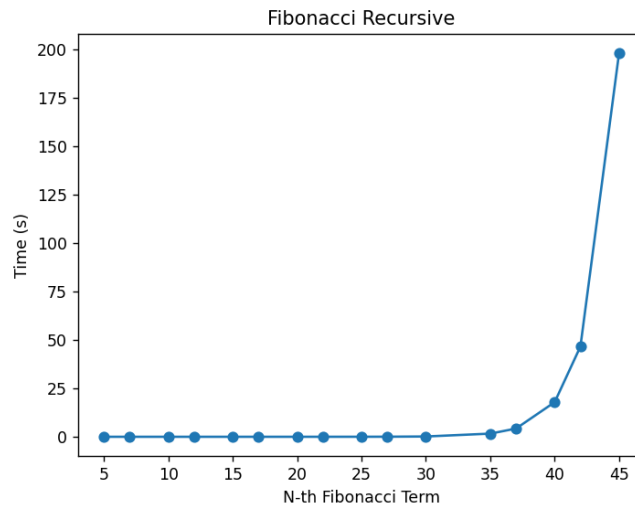


Figure 4 Graph of Recursive Fibonacci Function

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 40nd term, leading us to deduce that the Time Complexity is exponential, $O(2^n)$.

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]
```

Implementation:

```
def Fibonacci2(n):
    num = [0, 1]

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

    return num[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:

n	time(s)
500	0.000000
650	0.000000
800	0.000000
1000	0.000000
1250	0.000000
1500	0.000000
2000	0.000000
2550	0.000523
3150	0.000000
4000	0.001023
5000	0.000992
6350	0.001004
8000	0.002005
10000	0.004000
12500	0.006555
16000	0.009536

Figure 6 Fibonacci DP Results

With the Dynamic Programming Method we can see excellent results with a time complexity denoted in a corresponding graph of $O(n)$, comparing to Recursive Method.

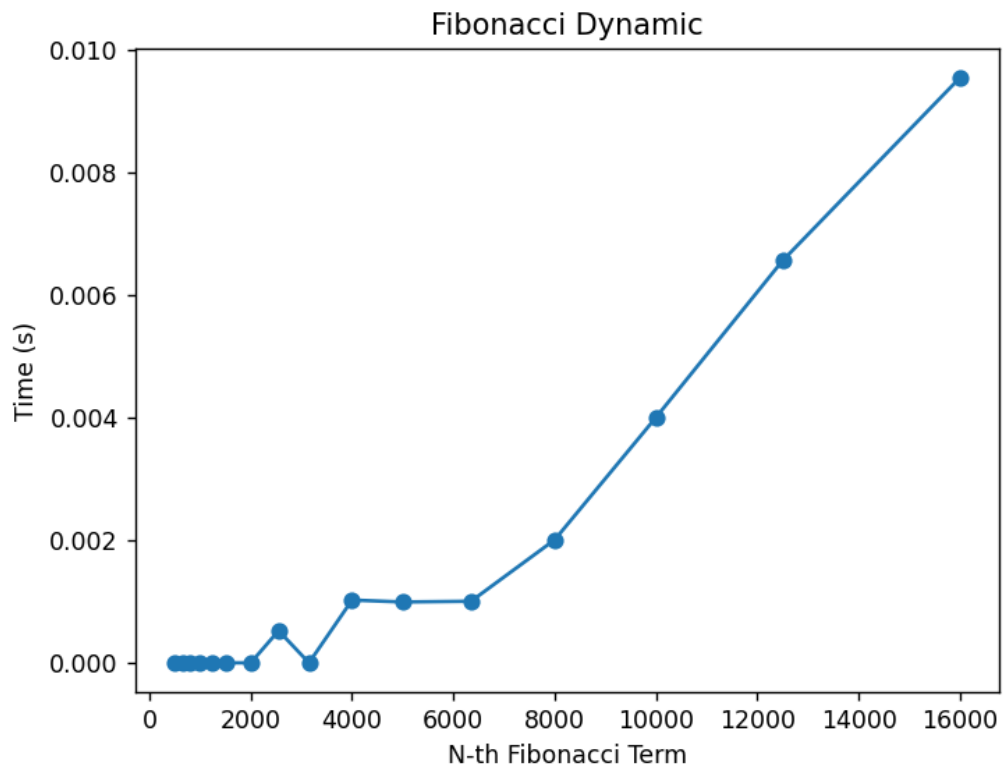


Figure 7 Fibonacci DP Graph

Iterative Programming Method:

The iterative method efficiently computes the n-th term of the Fibonacci sequence by directly calculating each term and using variables to track the current and next Fibonacci numbers, eliminating the need for recursion and optimizing the process.

Algorithm Description:

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

Fibonacci(n) :

 a := 0

 b := 1

 if n <= 1:

 return n

 else:

 for i from 2 to n:

 c := a + b

 a := b

 b := c

 return b

Implementation:

```
def Fibonacci3(n):  
    a = 0  
    b = 1  
  
    if n <= 1:  
        return n  
    else:  
        for i in range(2, n+1):  
            c = a + b  
            a = b  
            b = c  
        return b
```

Figure 8 Fibonacci Iterative 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:

n	time(s)
500	0.000000
650	0.000000
800	0.001024
1000	0.000000
1250	0.000000
1500	0.000000
2000	0.000000
2550	0.000000
3150	0.000000
4000	0.000993
5000	0.000000
6350	0.001005
8000	0.001000
10000	0.001000
12500	0.002000
16000	0.003520

Figure 9 Fibonacci Iterative Results

With the Iterative Programming Method we can see excellent results with a time complexity denoted in a corresponding graph of $O(n)$, same as Dynamic Method. Instead the iterative method directly computes without additional memory overhead.

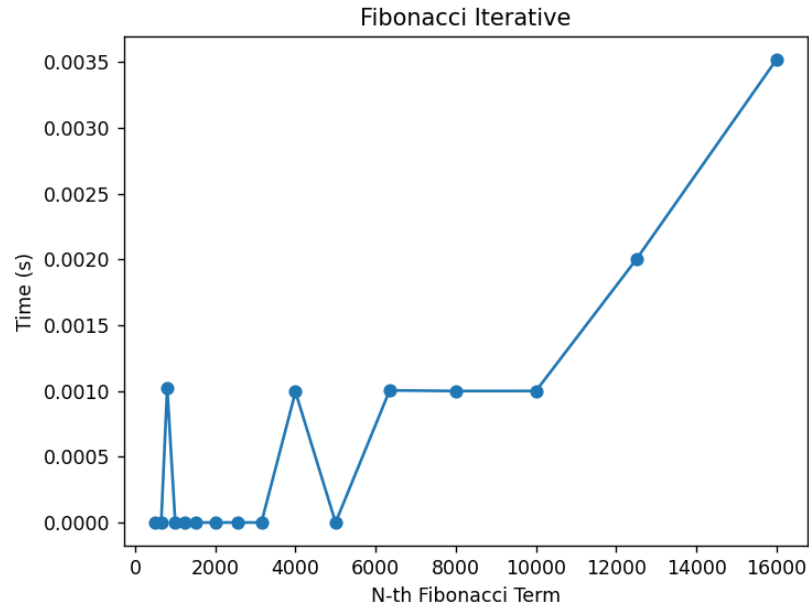


Figure 10 Fibonacci Iterative Graph

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 $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ with itself.

Algorithm Description:

It is known that

$$F_0 = 0$$

$$F_1 = 1$$

$$F_N = F_{N-1} + F_{N-2}$$

This property of Matrix multiplication can be used to represent

$$F_n = F_{n_1} = F^{n-1}_1 F_1 = \begin{pmatrix} F_n & F_{n-1} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} & F_{n-2} \end{pmatrix} \begin{pmatrix} 1 & 0 \end{pmatrix}$$

And similarly:

$$F_n = \begin{pmatrix} F_n + F_{n-1} & F_n \\ F_{n-1} + F_{n-2} & F_{n-1} \end{pmatrix}$$

Which turns into the general:

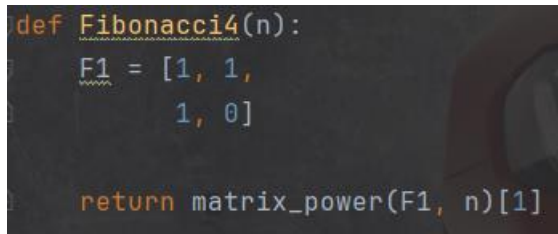
$$F_n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

This set of operation can be described in pseudocode as follows:

```
Fibonacci(n):  
    F = []  
    vec = [[0], [1]]  
    Matrix = [[1, 1], [1, 0]]  
    F = matrix_power(Matrix, n)  
    F = matrix_multiply(F, vec)  
    Return F[0][0]
```

Implementation:

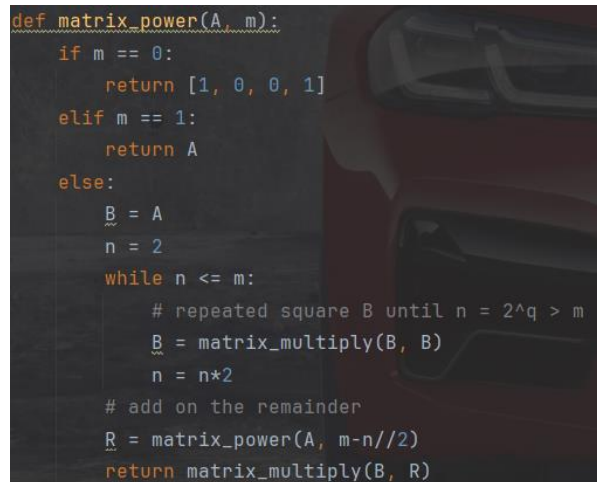
The implementation of the driving function in Python is as follows:



```
def Fibonacci4(n):  
    F1 = [[1, 1],  
          [1, 0]]  
  
    return matrix_power(F1, n)[1]
```

Figure 8 Fibonacci Matrix Power Method in Python

With additional miscellaneous functions:



```
def matrix_power(A, m):  
    if m == 0:  
        return [[1, 0, 0, 1]]  
    elif m == 1:  
        return A  
    else:  
        B = A  
        n = 2  
        while n <= m:  
            # repeated square B until n = 2^q > m  
            B = matrix_multiply(B, B)  
            n = n*2  
        # add on the remainder  
        R = matrix_power(A, m-n//2)  
        return matrix_multiply(B, R)
```

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 matrix_multiply(A, B):
    a, b, c, d = A
    x, y, z, w = B

    return (
        a * x + b * z,
        a * y + b * w,
        c * x + d * z,
        c * y + d * 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:

n	time(s)
500	0.000000
650	0.000000
800	0.000000
1000	0.000000
1250	0.000000
1500	0.000000
2000	0.000000
2550	0.000000
3150	0.000000
4000	0.000000
5000	0.000000
6350	0.000000
8000	0.000000
10000	0.000995
12500	0.000000
16000	0.000000

Figure 11 Matrix Method Fibonacci Results

"We observe that, thus far, the matrix method proves to be a faster approach for finding the n-th term of Fibonacci compared to both the iterative method and the dynamic programming method with a list. The matrix method exhibits a time complexity of $O(\log n)$, surpassing the linear time complexities of both the iterative ($O(n)$) and dynamic programming with list ($O(n)$) approaches.

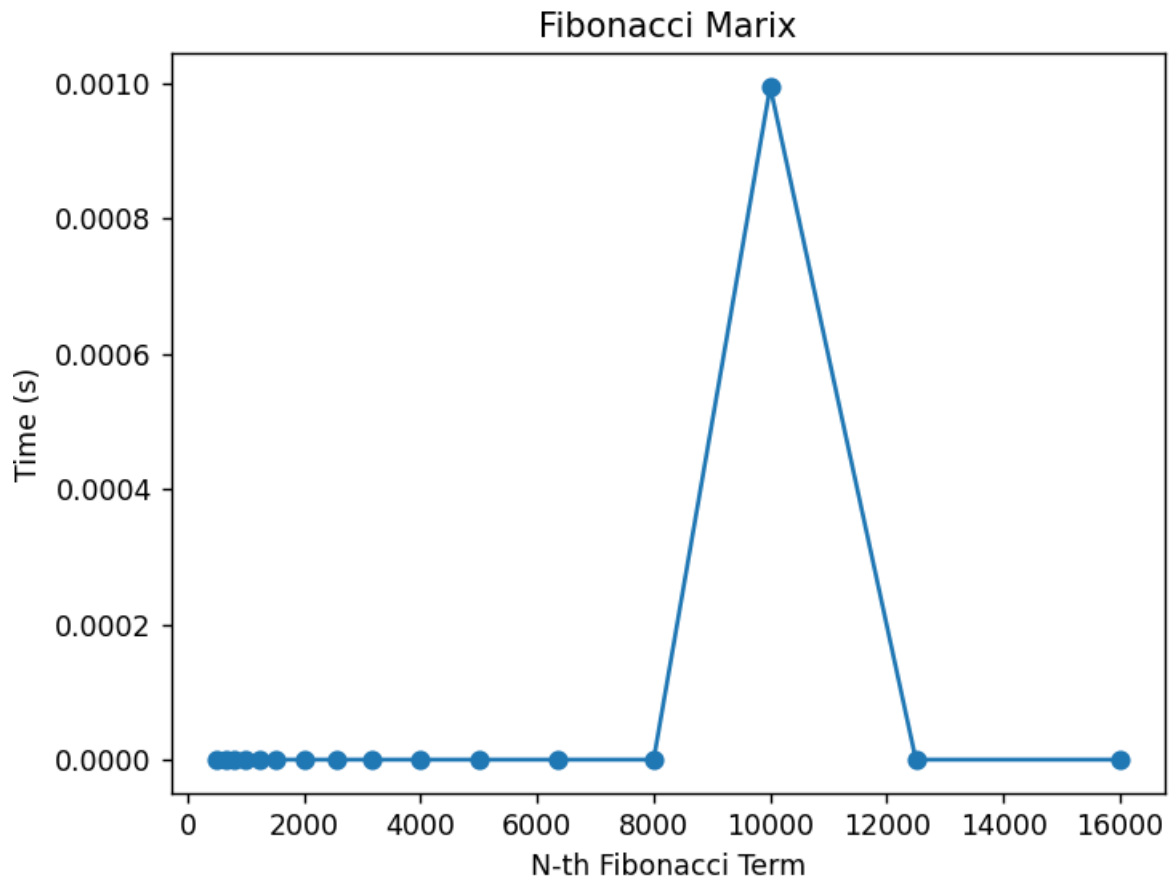


Figure 12 Matrix Method Fibonacci graph

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 72-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:

```
function Fibonacci(n):
    phi = (1 + sqrt(5)) / 2
    ans = ((phi^n) - ((-phi)^(-n))) / (2^n * sqrt(5))
    return round(ans) # Rounding to the nearest integer
```

Implementation:

The implementation of the function in Python is as follows:

```
def Fibonacci5(n):
    ans = (((1+sqrt(5))**n)-((1-sqrt(5))**n)/(2**n*sqrt(5)))
    return int(ans)
```

Figure 13 Fibonacci Binet Formula Method in Python

Results:

Input is the first list with small numbers, although the most performant with its time, as shown in the table of results:

n	time(s)
5	0.000000
7	0.000000
10	0.000000
12	0.000000
15	0.000000
17	0.000000
20	0.000000
22	0.000000
25	0.000000
27	0.000000
30	0.000000
35	0.000000
37	0.000000
40	0.000000
42	0.000000
45	0.000000

Figure 14 Fibonacci Binet Formula Method results

And as shown in its performance graph,

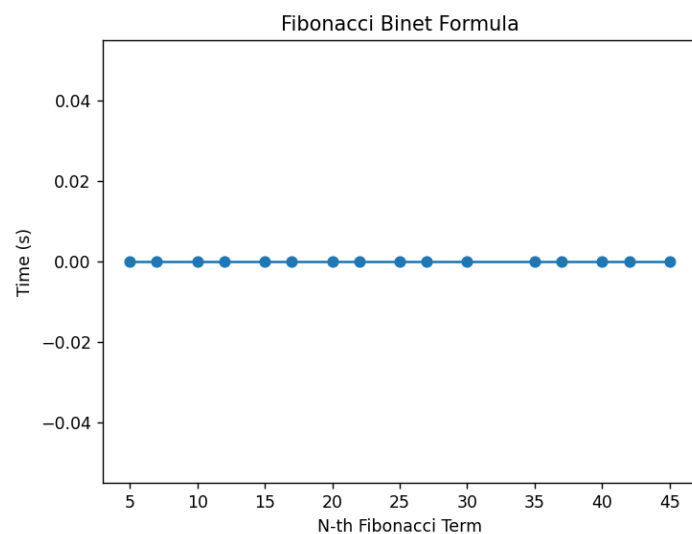


Figure 15 Fibonacci Binet formula Method

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

Matrix Exponentiation using Eigenvalues Method:

Matrix Exponentiation using Eigenvalues is a mathematical method employed in computational algorithms to efficiently calculate the n-th term of sequences, such as the Fibonacci sequence. By diagonalizing the transition matrix associated with the sequence through its eigenvalues and eigenvectors, the method leverages the properties of matrix exponentiation, providing a succinct and computationally advantageous approach for obtaining Fibonacci numbers. In this context, the method allows for precise and rapid calculations by exploiting the inherent mathematical structure encoded in the eigenvalues and eigenvectors of the matrix.

Algorithm Description:

Note that the matrix F_1 is symmetric and real-valued. Therefore it has real eigenvalues which we'll call λ_1 and λ_2 . The eigenvalue decomposition allows us to diagonalize F_1 . We can calculate the two eigenvalues analytically by solving the characteristic equation $(1-\lambda)\lambda-1=0$. Since this is a quadratic polynomial, we can use the quadratic equation to obtain both solutions in closed form:

$$\lambda_1 = (1 + \sqrt{5}) / 2$$

$$\lambda_2 = (1 - \sqrt{5}) / 2$$

The set of operation for the Matrix Exponentiation using Eigenvalues Method can be described in pseudocode as follows:

`Fibonacci(n) :`

```

    F1 = Matrix([[1, 1], [1, 0]]) # Initial Fibonacci matrix
    eigenvalues, eigenvectors = EigenvalueDecomposition(F1)
    Fn = eigenvectors @ DiagonalMatrix(eigenvalues ** n) @
    Transpose(eigenvectors)
    return Round(Fn[0, 1])

```

Implementation:

The implementation of the function in Python is as follows:


```
def Fibonacci6(n):
    F1 = np.array([[1, 1], [1, 0]])
    eigenvalues, eigenvectors = np.linalg.eig(F1)
    Fn = eigenvectors @ np.diag(eigenvalues ** n) @ eigenvectors.T
    return int(np rint(Fn[0, 1]))
```

Figure 11 Fibonacci Binet Formula Method in Python

Results:

I used the input from first list with numbers, because this method after 1475 term doesn't work. Although the most it is very performant with its time, as shown in the table of results:,

n	time(s)
5	0.000000
7	0.000000
10	0.000000
12	0.000000
15	0.000000
17	0.000000
20	0.000000
22	0.000000
25	0.000000
27	0.000000
30	0.000000
35	0.000000
37	0.000000
40	0.000000
42	0.000000
45	0.000000

Figure 12 Fibonacci Binet Formula Method results

And as shown in its performance graph,

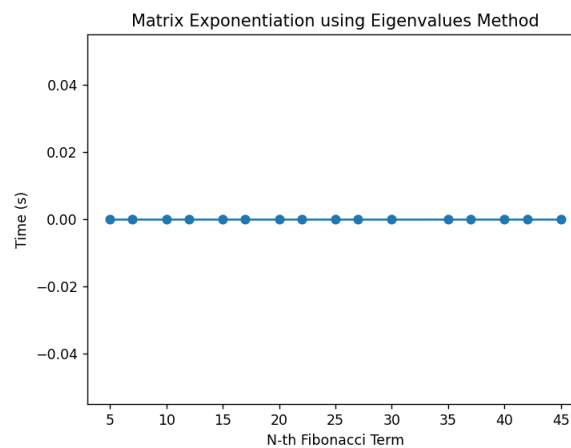


Figure 13 Fibonacci Binet formula Method

So there you have it, a $O(1)$ algorithm for any Fibonacci number. There's just one tiny little problem with it: ϕ , being irrational, is not particularly convenient for numerical analysis. If we run the above Python program, it will use 64-bit floating point arithmetic and will never be able to precisely represent more than 15 decimal digits. That only lets us calculate up to F_{93} before we no longer have enough precision to exactly represent it. Past F_{93} , our clever little "exact" eigenvalue algorithm is good for nothing but a rough approximation!

CONCLUSION

In conclusion, the analysis of various Fibonacci number calculation methods reveals distinct trade-offs in terms of efficiency, accuracy, and practical usability. The recursive method, while conceptually straightforward, exhibits exponential time complexity ($O(2^n)$), making it highly inefficient. Dynamic programming and iterative methods significantly improve efficiency, achieving linear time complexity ($O(n)$) by eliminating redundant calculations. The matrix method, leveraging matrix exponentiation, proves to be the most efficient, with a logarithmic time complexity ($O(\log n)$), surpassing both dynamic programming and iterative approaches. However, the Binet Formula method, relying on the Golden Ratio, faces precision challenges, limiting its practical use beyond the 72nd Fibonacci term. Matrix exponentiation using eigenvalues offers an $O(1)$ algorithm but encounters precision issues due to the irrational nature of the Golden Ratio. In practice, the choice of method depends on the specific requirements of the application, considering factors such as precision needs, speed, and ease of implementation. Up to the 70th Fibonacci term, the Binet Formula method stands out for its exceptional speed. Despite its precision limitations beyond the 72nd term due to rounding errors, it remains a viable option for quick calculations within the specified range. On the other hand, the recursive method is notably slow, especially for larger terms. The matrix method, offering a good balance of precision and speed, proves highly effective for both small and large-scale Fibonacci numbers, demonstrating its versatility. As an efficient algorithm with 100% precision and fast execution for substantial numbers, the matrix method stands as a robust choice, particularly when dealing with Fibonacci calculations in diverse scenarios. Ultimately, the comprehensive understanding of the strengths and limitations of each method empowers practitioners to make informed decisions based on the unique requirements of their computational tasks.