



**MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII  
AL REPUBLICII MOLDOVA Universitatea Tehnică a  
Moldovei Facultatea Calculatoare, Informatică și  
Microelectronică Departamentul Inginerie Software și  
Automatică**

Copta Adrian | FAF-223

# **Report**

*Laboratory work 1:*

## ***Study and Empirical Analysis of Algorithms for Determining Fibonacci N-th Term***

Checked by:

**Fiștic Cristof**, *university assistant*

DISA, FCIM, UTM

## TABLE OF CONTENTS:

<b>ALGORITHM ANALYSIS.....</b>	<b>3</b>
Objective.....	3
Tasks.....	3
Theoretical Notes.....	3
Introduction.....	4
Comparison Metric.....	4
Input Format.....	5
<b>IMPLEMENTATION.....</b>	<b>6</b>
Recursive Method.....	6
Dynamic Programming Method.....	9
Tabular Method.....	12
Iterative Method.....	15
Fibonacci sequence properties.....	18
Eigenvalue Method.....	21
Matrix Power Method.....	25
Binet Formula Method.....	29
<b>CONCLUSION.....</b>	<b>33</b>

# 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, (1, 1, 2, 4, 4, 4, 5, 6, 7, 10, 10, 10, 13, 17, 19, 20, 20, 21, 21, 23, 28, 36, 36, 36, 39, 41, 41, 41, 43, 44), to accommodate the recursive method, while the second series will have a bigger scope to be able to compare the other algorithms between themselves (1690, 7722, 7748, 11368, 11882, 11896, 13187,

13253, 16463, 16612, 17059, 19320, 20387, 21368, 21829, 22497, 22552, 23433, 23862, 27466, 28146, 29305, 29633, 30559, 30738, 31959, 33083, 33759, 33777, 34125, 34154, 35999, 37750, 39186, 41368, 42446, 44679, 44766, 47612).

## 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:

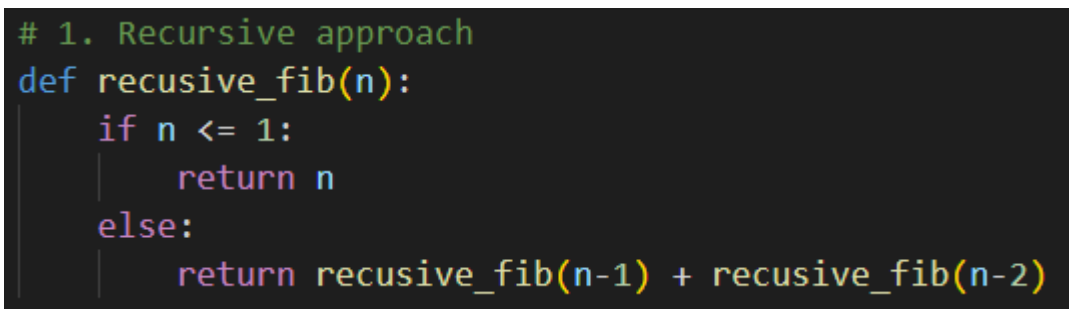
The recursive method, also considered the most inefficient method, follows a straightforward approach of computing the n-th term by computing its predecessors first, and then adding them. However, the method does it by calling upon itself a number of times and repeating the same operation, for the same term, at least twice, occupying additional memory and, in theory, doubling its execution time.

#### *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:*



```
# 1. Recursive approach  
def recursive_fib(n):  
    if n <= 1:  
        return n  
    else:  
        return recursive_fib(n-1) + recursive_fib(n-2)
```

*Figure 1 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:

## Time result of recursive\_fib function

n-th Fibonacci termen	Time (s)
1.0	0.0
1.0	0.0
2.0	0.0
4.0	0.0
4.0	0.0
4.0	0.0
5.0	0.0
6.0	0.0
7.0	0.0
10.0	0.0
10.0	0.0
10.0	0.0
13.0	0.0
17.0	0.0
19.0	0.0
20.0	0.000998
20.0	0.001002
21.0	0.000997
21.0	0.001
23.0	0.003
28.0	0.035001
36.0	1.652231
36.0	1.644615
36.0	1.636634
39.0	6.95748
41.0	18.207441
41.0	18.33649
41.0	18.225968
43.0	47.972083
44.0	77.841995

Figure 2 Results for first set of inputs

In Figure 2 is represented the table of results for the first set of inputs. The first column is the n-th Fibonacci term from our input, the second column represents the execution time of each n-th term. Starting from the 16-th row, we observe a growth in execution time. We may notice that the only function whose time was growing for this few n terms was the Recursive Method Fibonacci function.

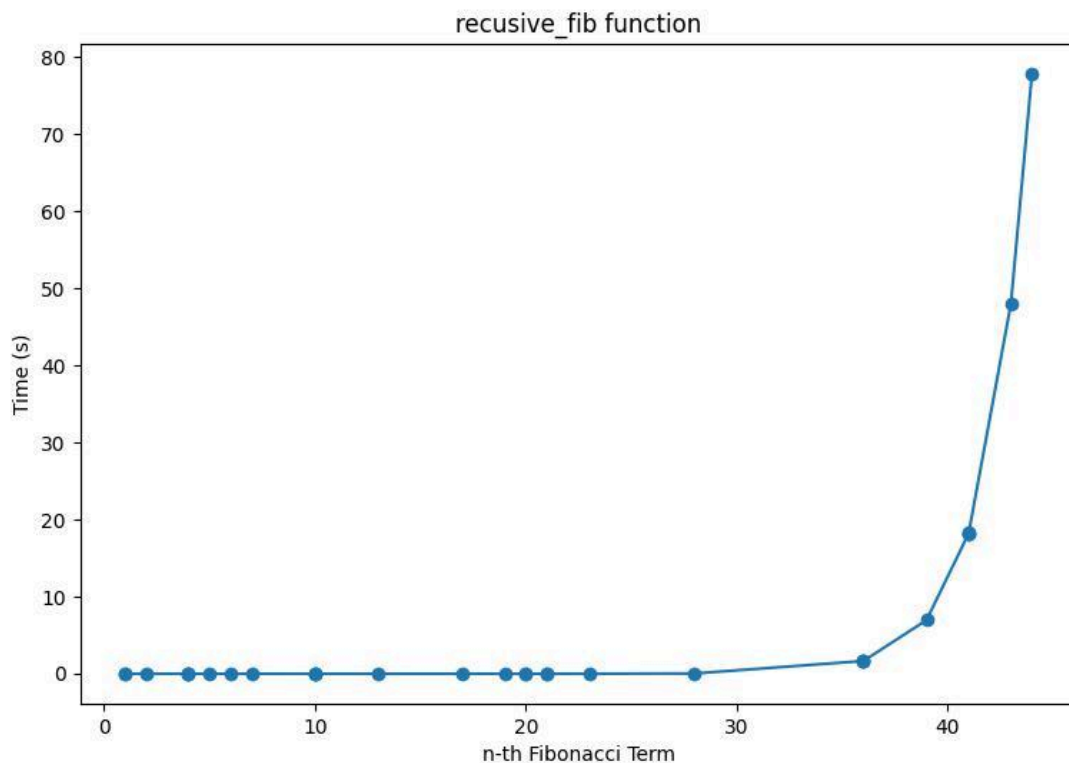


Figure 3 Graph of Recursive Fibonacci Function

Not only that, but also in the graph in Figure 3 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.  $T(2n)$ .

### Dynamic method (with memoization):

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 method follows the algorithm as shown in the next pseudocode:

DynamicFibonacci(n, storage=None):



```

if storage is None:
    initialize storage as an empty dictionary
if n <= 1:
    return n
if n is already in storage:
    return the value of storage[n]
Set storage[0] to 0
Set storage[1] to 1

for i from 2 to n:
    storage[i] = storage[i - 1] + storage[i - 2]
return the value of storage[n]

```

*Implementation:*

```

# 2. Dynamic Programming approach (with memoization)
def dynamic_fib(n, storage=None):
    if storage is None:
        storage = {}
    if n in storage:
        return storage[n]

    storage[0], storage[1] = 0, 1
    for i in range(2, n + 1):
        storage[i] = storage[i - 1] + storage[i - 2]
    return storage[n]

```

*Figure 4 Fibonacci dynamic in Python*

*Results:*

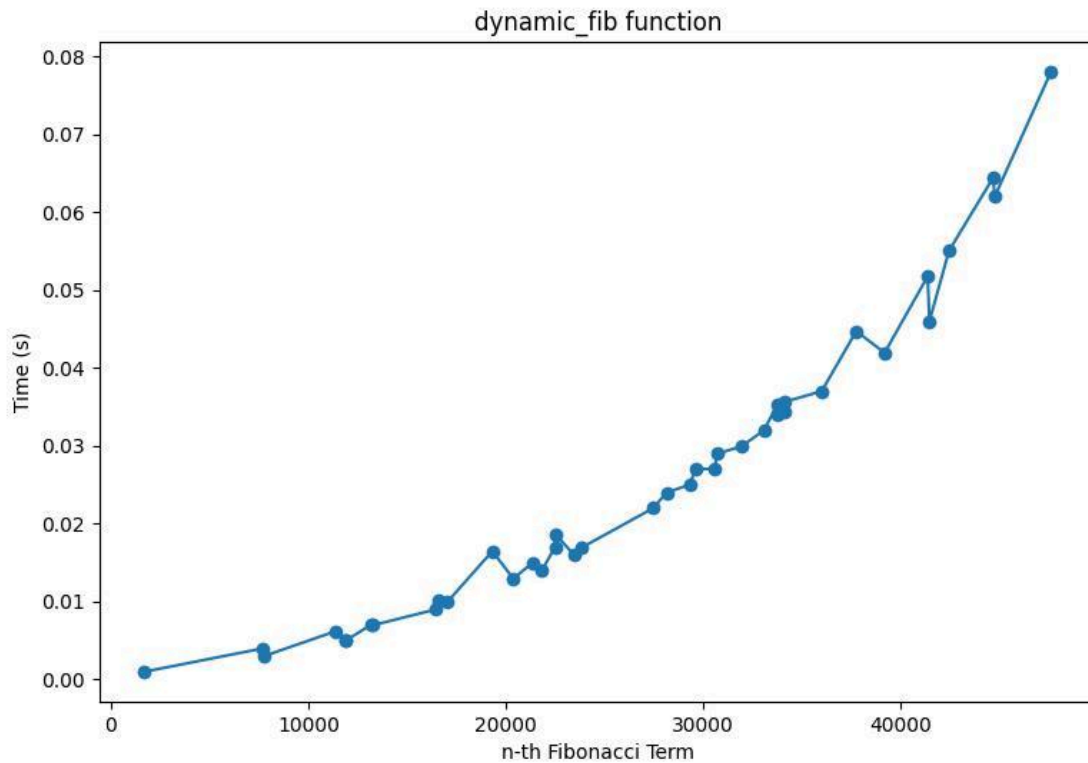
After running the function for each n Fibonacci term proposed in the list from the second Input Format and saving the time for each n, we obtained the following results:

Time result of dynamic\_fib function

n-th Fibonacci termen	Time (s)
1690.0	0.000998
7722.0	0.004002
7748.0	0.002999
11368.0	0.006185
11882.0	0.005
11896.0	0.004999
13187.0	0.00704
13253.0	0.00696
16463.0	0.009
16612.0	0.01011
17059.0	0.01001
19320.0	0.016467
20387.0	0.01296
21368.0	0.015002
21829.0	0.014
22497.0	0.016999
22552.0	0.018545
23443.0	0.016042
23862.0	0.016957
27466.0	0.021995
28146.0	0.024
29305.0	0.025
29633.0	0.027066
30559.0	0.026999
30738.0	0.029
31959.0	0.029953
33083.0	0.032001
33759.0	0.035364
33777.0	0.033996
34125.0	0.034326
34154.0	0.035664
35999.0	0.037
37750.0	0.044719
39186.0	0.041931
41368.0	0.051748
41457.0	0.046
42446.0	0.055014
44679.0	0.064398
44766.0	0.062001
47612.0	0.078002

Figure 5 Results for second set of inputs

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



*Figure 6 Graph of Dynamic Fibonacci Function*

### **Tabulation method:**

Tabulation is actually a dynamic programming technique. It is very simple to describe but can be difficult to implement correctly. From my perspective, these are not intuitive connections you immediately see, but as you dig down into problems, you might start to see how a tabulation solution could be generated. Personally, I would solve the problem a different way first to ensure that you completely understand it before jumping into a tabulated solution. All tabulation means is solving a problem by building out some sort of table. Once the table is built out, somewhere in the table, based on the problem, we will find the solution to our problem and we can then return it.

#### *Algorithm Description:*

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

TableFibonacci(n):

```

if n < 2:
    return n

```

```

// Initialize a table with -1 values
table = create array of size (n + 1) filled with -1

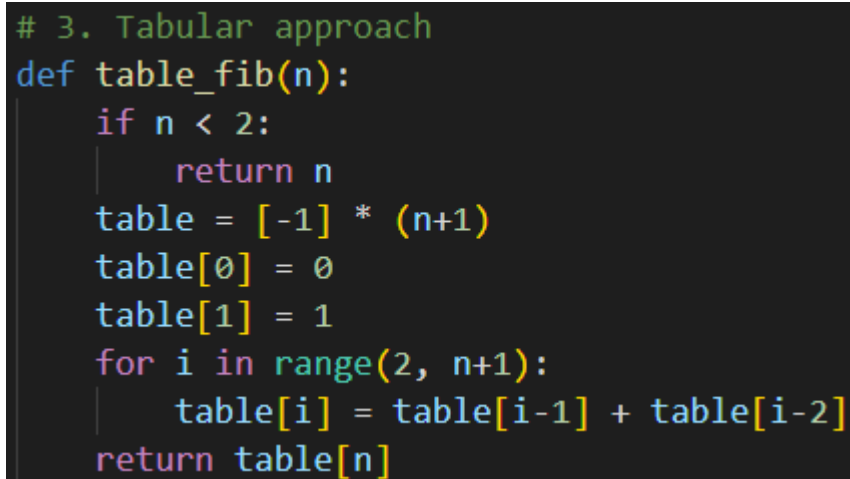
// Base cases
table[0] = 0
table[1] = 1

// Fill the table using dynamic programming
for i from 2 to n:
    table[i] = table[i - 1] + table[i - 2]

return the value of table[n]

```

*Implementation:*



```

# 3. Tabular approach
def table_fib(n):
    if n < 2:
        return n
    table = [-1] * (n+1)
    table[0] = 0
    table[1] = 1
    for i in range(2, n+1):
        table[i] = table[i-1] + table[i-2]
    return table[n]

```

*Figure 7 Fibonacci tabulation 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:

Time result of table\_fib function

n-th Fibonacci termen	Time (s)
1690.0	0.0
7722.0	0.002003
7748.0	0.001999
11368.0	0.004998
11882.0	0.004002
11896.0	0.002999
13187.0	0.005097
13253.0	0.002999
16463.0	0.007999
16612.0	0.006001
17059.0	0.008001
19320.0	0.008999
20387.0	0.010002
21368.0	0.009999
21829.0	0.013001
22497.0	0.012999
22552.0	0.013
23443.0	0.015002
23862.0	0.014999
27466.0	0.019999
28146.0	0.021001
29305.0	0.022383
29633.0	0.022275
30559.0	0.024
30738.0	0.024001
31959.0	0.027001
33083.0	0.029999
33759.0	0.029999
33777.0	0.031447
34125.0	0.031433
34154.0	0.030998
35999.0	0.035894
37750.0	0.038155
39186.0	0.04
41368.0	0.047331
41457.0	0.046614
42446.0	0.049064
44679.0	0.052026
44766.0	0.04632
47612.0	0.055481

Figure 8 Results for second set of inputs

With the Tabular Method showing excellent results with a time complexity denoted in a corresponding graph of  $T(n)$ ,

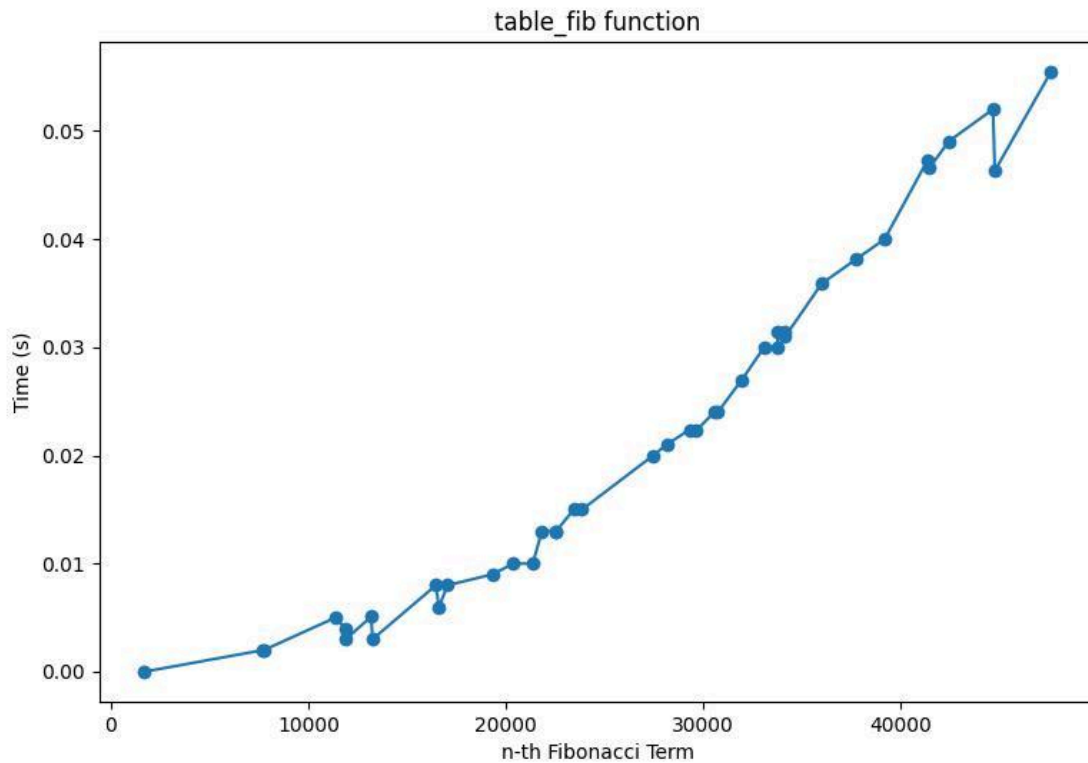


Figure 9 Graph of Tabular Fibonacci Function

### Iterative method:

Observing that we only ever have to use the two most recent Fibonacci numbers, the tabular solution can easily be made iterative, resulting in a large space savings:

*Algorithm Description:*

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

IterativeFibonacci(n):

    set previous to 0

    set current to 1

    for i from 2 to n:

        set temporary to current

        set current to previous + current

        set previous to temporary

return the value of current

*Implementation:*

```
# 4. Iterative approach
def iterative_fib(n):
    previous, current = (0, 1)
    for i in range(2, n+1):
        previous, current = (current, previous + current)
    return current
```

*Figure 10 Fibonacci iterative approach 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:

Time result of iterative\_fib function

n-th Fibonacci termen	Time (s)
1690.0	0.0
7722.0	0.002445
7748.0	0.001009
11368.0	0.002001
11882.0	0.001997
11896.0	0.002403
13187.0	0.002007
13253.0	0.000999
16463.0	0.003001
16612.0	0.002
17059.0	0.003999
19320.0	0.003998
20387.0	0.003
21368.0	0.004001
21829.0	0.004001
22497.0	0.003998
22552.0	0.004
23443.0	0.004
23862.0	0.005008
27466.0	0.005038
28146.0	0.005996
29305.0	0.007
29633.0	0.005959
30559.0	0.007
30738.0	0.007
31959.0	0.008006
33083.0	0.007993
33759.0	0.009009
33777.0	0.007991
34125.0	0.009
34154.0	0.008999
35999.0	0.009
37750.0	0.009044
39186.0	0.010956
41368.0	0.011004
41457.0	0.011996
42446.0	0.012
44679.0	0.014005
44766.0	0.014996
47612.0	0.014999

Figure 11 Results for second set of inputs



With the Iterative Method showing excellent results with a time complexity denoted in a corresponding graph of  $T(n)$ ,

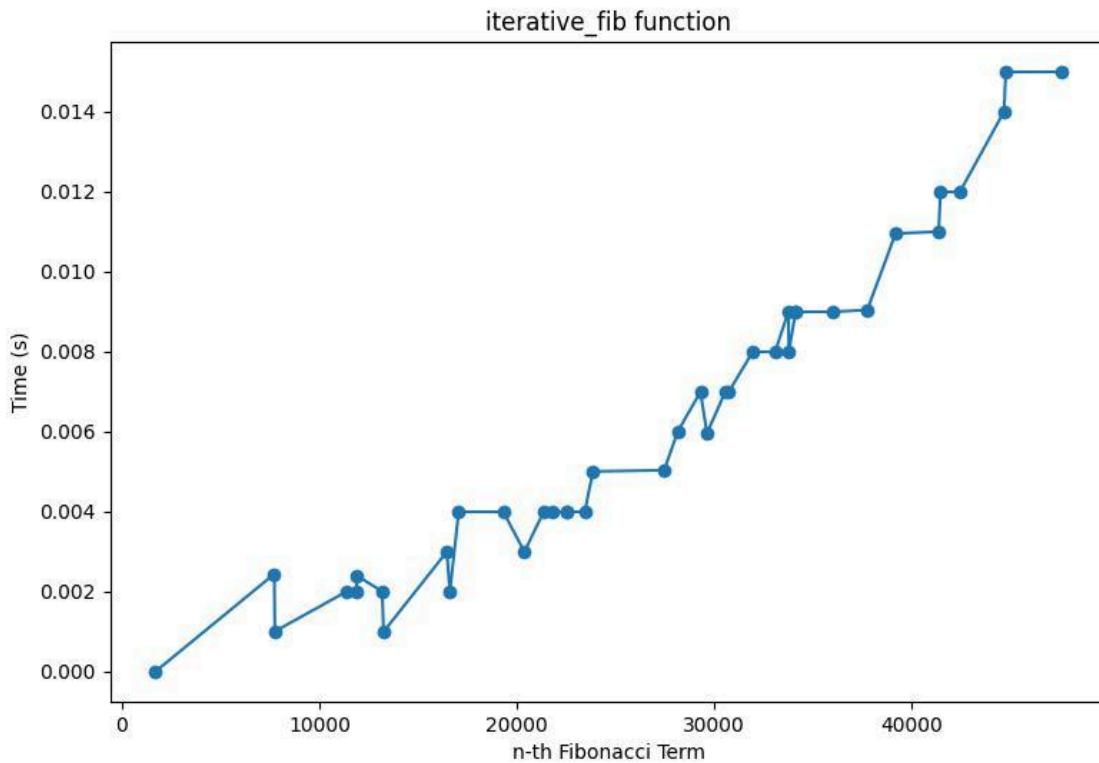


Figure 12 Graph of Iterative Fibonacci Function

### Fibonacci sequence properties method:

The code defines a function `fibonacci(n)` that calculates the  $n$ th Fibonacci number using dynamic programming. It initializes a list `FibArray` with the first two Fibonacci numbers (0 and 1). The function checks if the Fibonacci number for  $n$  is already present in `FibArray` and returns it. Otherwise, it calculates the Fibonacci number recursively, stores it in `FibArray` for future use, and returns the calculated value.

#### Algorithm Description:

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

PropertiesFibonacci( $n$ ):

```

if  $n \leq 1$ :
    return  $n$ 
else:
    fib = create array [0, 1]

```

```
for i from 2 to n:  
    append (fib[-1] + fib[-2]) to fib
```

```
return the value of fib[-1]
```

*Implementation:*

```
# 5. Fibonacci sequence properties approach  
def properties_fib(n):  
    if n <= 1:  
        return n  
    else:  
        fib = [0, 1]  
        for i in range(2, n+1):  
            fib.append(fib[-1] + fib[-2])  
        return fib[-1]
```

*Figure 13 Fibonacci array approach 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:

Time result of properties\_fib function

n-th Fibonacci termen	Time (s)
1690.0	0.0
7722.0	0.0024
7748.0	0.002009
11368.0	0.005999
11882.0	0.003059
11896.0	0.004001
13187.0	0.002998
13253.0	0.002999
16463.0	0.007
16612.0	0.005
17059.0	0.004
19320.0	0.007001
20387.0	0.006999
21368.0	0.010038
21829.0	0.006961
22497.0	0.010999
22552.0	0.008
23443.0	0.012
23862.0	0.008002
27466.0	0.016001
28146.0	0.016478
29305.0	0.011997
29633.0	0.014
30559.0	0.022003
30738.0	0.024
31959.0	0.024999
33083.0	0.016001
33759.0	0.016997
33777.0	0.028
34125.0	0.016
34154.0	0.026004
35999.0	0.019999
37750.0	0.034998
39186.0	0.022002
41368.0	0.042298
41457.0	0.042577
42446.0	0.025997
44679.0	0.049003
44766.0	0.025084
47612.0	0.037001

Figure 14 Results for second set of inputs

The Array Method showing excellent results with a time complexity denoted in a corresponding graph of T(n),

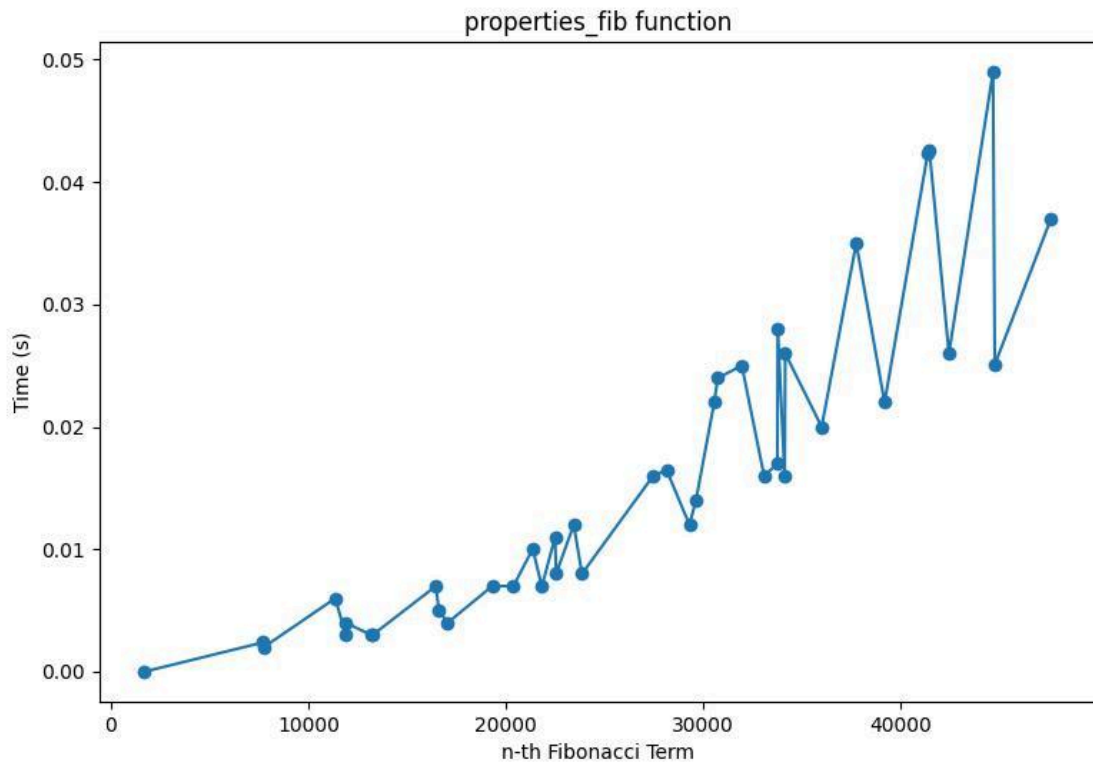


Figure 15 Graph of Sequence Proprieties Fibonacci Function

### Eigenvalue method:

*Algorithm Description:*

The eigenvalue decomposition allows us to diagonalize F1 like so:

$$\mathbf{F}_1 = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T = \mathbf{Q} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \mathbf{Q}^T$$

Writing F1 in this form makes it easy to square it:

$$\begin{aligned} \mathbf{F}_1^2 &= \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T \\ &= \mathbf{Q}\mathbf{\Lambda}^2\mathbf{Q}^T \\ &= \mathbf{Q} \begin{bmatrix} \lambda_1^2 & 0 \\ 0 & \lambda_2^2 \end{bmatrix} \mathbf{Q}^T \end{aligned}$$

or to raise it to an arbitrary power:

$$\mathbf{F}_n = \mathbf{F}_1^n = \mathbf{Q}\mathbf{\Lambda}^n\mathbf{Q}^T = \mathbf{Q} \begin{bmatrix} \lambda_1^n & 0 \\ 0 & \lambda_2^n \end{bmatrix} \mathbf{Q}^T$$

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:

$$\lambda_1 = \frac{1 + \sqrt{5}}{2}$$

$$\lambda_2 = \frac{1 - \sqrt{5}}{2}$$

Where the largest eigenvalue is in fact  $\phi$ , the golden ratio. The matrix formulation is an easy way to see famous connection between the Fibonacci numbers and  $\phi$ . To calculate  $F_n$  for large values of  $n$ , it suffices to calculate  $\phi^n$  and then do some constant time  $O(1)$  bookkeeping, like so:

EigenFibonacci( $n$ , mod= $10^9 + 7$ ):

```
function mod_pow(base, exp, mod):
```

```
    result = 1
```

```
    base = base % mod
```

```
    while exp > 0:
```

```
        if exp is odd:
```

```
            result = (result * base) % mod
```

```
        exp = exp // 2
```

```
        base = (base * base) % mod
```

```
    return result
```

```
F1 = [[1, 1], [1, 0]]
```

```
eigenvalues, eigenvectors = calculate_eigen(F1)
```

```
# Use modular exponentiation to prevent overflow
```

```
Fn = matrix_multiply(eigenvectors, diagonal_matrix([mod_pow(eig, n, mod) for eig in
eigenvalues]), transpose(eigenvectors))
```

```
return round(Fn[0, 1]) % mod
```

*Implementation:*

```
# 6. Eigenvalue approach
def eigen_fib(n, mod=10**9 + 7):
    def mod_pow(base, exp, mod):
        result = 1
        base = base % mod
        while exp > 0:
            if exp % 2 == 1:
                result = (result * base) % mod
            exp = exp // 2
            base = (base * base) % mod
        return result

    F1 = np.array([[1, 1], [1, 0]])
    eigenvalues, eigenvectors = np.linalg.eig(F1)

    # Use modular exponentiation to prevent overflow
    Fn = eigenvectors @ np.diag([mod_pow(eig, n, mod) for eig in eigenvalues]) @ eigenvectors.T

    return int(np rint(Fn[0, 1])) % mod
```

*Figure 16 Fibonacci eigenvalue approach 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:

Time result of eigen\_fib function

n-th Fibonacci termen	Time (s)
1690.0	0.0
7722.0	0.0
7748.0	0.001
11368.0	0.0
11882.0	0.0
11896.0	0.0
13187.0	0.0
13253.0	0.0
16463.0	0.0
16612.0	0.0
17059.0	0.0
19320.0	0.0
20387.0	0.0
21368.0	0.0
21829.0	0.0
22497.0	0.0
22552.0	0.0
23443.0	0.001
23862.0	0.0
27466.0	0.0
28146.0	0.0
29305.0	0.0
29633.0	0.0
30559.0	0.0
30738.0	0.0
31959.0	0.0
33083.0	0.0
33759.0	0.0
33777.0	0.0
34125.0	0.0
34154.0	0.0
35999.0	0.0
37750.0	0.0
39186.0	0.0
41368.0	0.0
41457.0	0.0
42446.0	0.0
44679.0	0.0
44766.0	0.001
47612.0	0.0

Figure 17 Results for second set of inputs

So there you have it – an  $O(1)$  algorithm for any Fibonacci number. There’s just one tiny little problem with it:  $\phi$ , 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 F93 before we no longer have enough precision to exactly represent it. Past F93, our clever little “exact” eigenvalue algorithm is good for nothing but a rough approximation! Now, we could use high precision rational numbers, but that approach turns out to always require strictly more space and time than just sticking to integers. So, abandoning the eigenvalue approach on the garbage heap of ivory tower theory, let’s turn our attention to simply calculating the powers of an integer matrix.

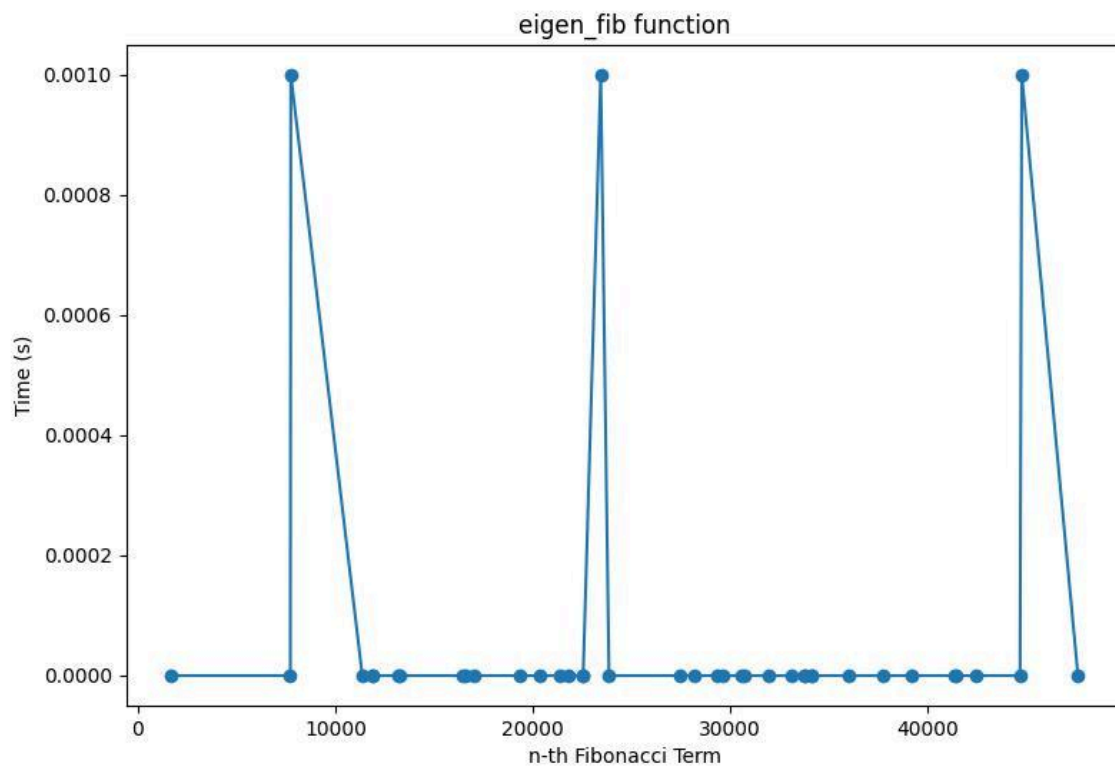


Figure 18 Graph of Eigenvalue Function

### Matrix method:

Luckily, because all Fibonacci matrices are of a special form, we really only need to keep track of two elements in the right-hand column of the matrix. I call this the “implicit matrix form.” Here is a Fibonacci matrix described with just two numbers,  $a$  and  $b$  :



$$\mathbf{F}_n = \begin{bmatrix} a+b & a \\ a & b \end{bmatrix}$$

We can easily work out closed forms for multiplying and squaring matrices in this form. While the full expressions are a little complex - we never actually need to explicitly calculate the left-hand column, a fact I will indicate by graying those columns out:

$$\begin{bmatrix} a+b & a \\ a & b \end{bmatrix} \begin{bmatrix} x+y & x \\ x & y \end{bmatrix} = \begin{bmatrix} a(2x+y) + b(x+y) & a(x+y) + bx \\ a(x+y) + bx & ax + by \end{bmatrix}$$

$$\begin{bmatrix} a+b & a \\ a & b \end{bmatrix}^2 = \begin{bmatrix} 2a^2 + 2ab + b^2 & a^2 + 2ab \\ a^2 + 2ab & a^2 + b^2 \end{bmatrix}$$

Using the implicit matrix form, we can multiply two different Fibonacci matrices with just four multiplications, and we can square a matrix with only three! It's only a constant time speed-up but every little bit helps.

*Algorithm Description:*

function multiply(a, b, x, y):

    return x \* (a + b) + a \* y, a \* x + b \* y

function square(a, b):

    a2 = a \* a

    b2 = b \* b

    ab = a \* b

    return a2 + (ab << 1), a2 + b2

function power(a, b, m):

    if m == 0:

        return 0, 1

    elif m == 1:

        return a, b

    else:

        x, y = a, b

        n = 2

        while n <= m:

```

# repeated square until  $n = 2^q > m$ 
x, y = square(x, y)
n = n * 2
# add on the remainder
a, b = power(a, b, m - n // 2)
return multiply(x, y, a, b)

```

function matrix\_fibonacci(n):

```

a, b = power(1, 0, n)
return a

```

*Implementation:*

```

# 7. Matrix approach
def multiply(a, b, x, y):
    return x*(a+b) + a*y, a*x + b*y

def square(a, b):
    a2 = a * a
    b2 = b * b
    ab = a * b
    return a2 + (ab << 1), a2 + b2

def power(a, b, m):
    if m == 0:
        return (0, 1)
    elif m == 1:
        return (a, b)
    else:
        x, y = a, b
        n = 2
        while n <= m:
            # repeated square until  $n = 2^q > m$ 
            x, y = square(x, y)
            n = n*2
        # add on the remainder
        a, b = power(a, b, m-n//2)
        return multiply(x, y, a, b)

def matrix_fib(n):
    a, b = power(1, 0, n)
    return a

```

Figure 19 Fibonacci matrix approach 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:

Time result of matrix_fib function	
n-th Fibonacci termen	Time (s)
1690.0	0.0
7722.0	0.0
7748.0	0.0
11368.0	0.001001
11882.0	0.0
11896.0	0.0
13187.0	0.000998
13253.0	0.0
16463.0	0.001
16612.0	0.0
17059.0	0.001131
19320.0	0.001009
20387.0	0.0
21368.0	0.000999
21829.0	0.0
22497.0	0.0
22552.0	0.000999
23443.0	0.000999
23862.0	0.0
27466.0	0.000999
28146.0	0.0
29305.0	0.001003
29633.0	0.000997
30559.0	0.0
30738.0	0.001001
31959.0	0.000998
33083.0	0.001001
33759.0	0.001001
33777.0	0.0
34125.0	0.001001
34154.0	0.001004
35999.0	0.0
37750.0	0.001
39186.0	0.001997
41368.0	0.001
41457.0	0.000999
42446.0	0.000999
44679.0	0.000998
44766.0	0.001
47612.0	0.002

*Figure 20 Results for second set of inputs*

It would of course be possible to derive these relationships without ever introducing the Fibonacci matrices, but I think they provide a valuable foundation for intuition. Without that foundation, the above program seems a little arbitrary. You may be wondering why I square numbers as `a*a` instead of `a**2` or `pow(a, 2)`, and why I use `ab<<1` instead of `2*ab` or `ab+ab` to double them. The answer is simple - I benchmarked the various forms and found these expressions to be very slightly faster.

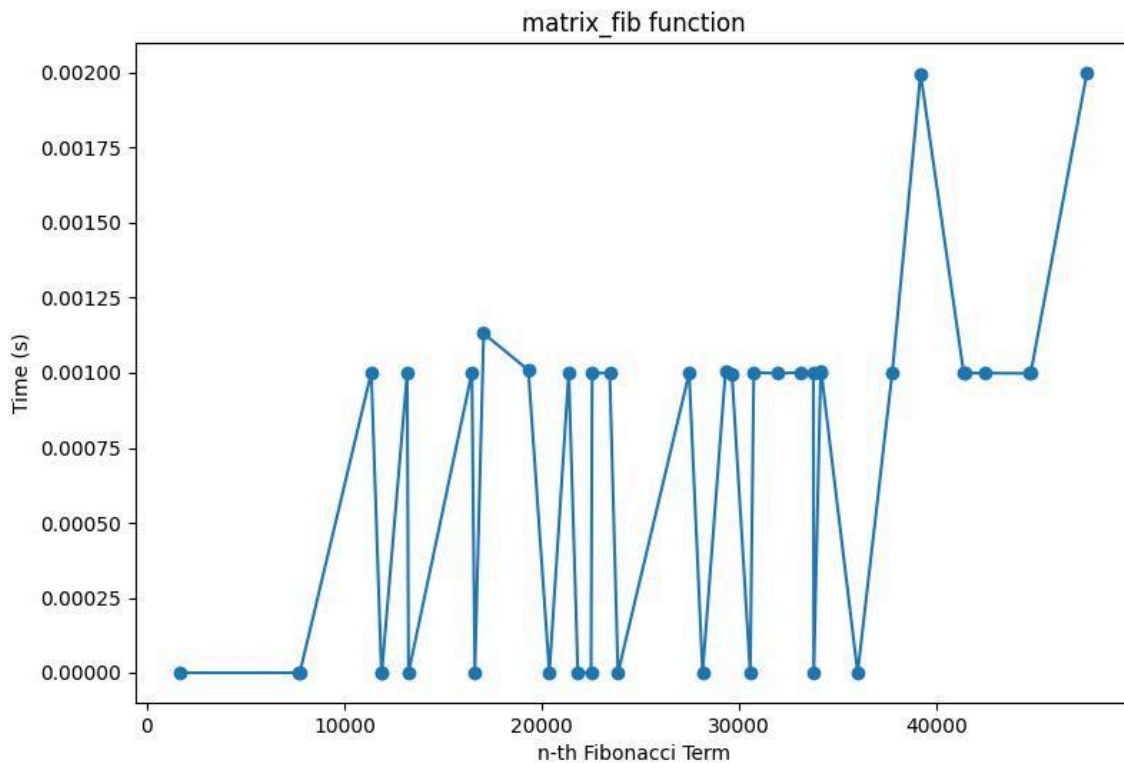


Figure 21 Graph of Matrix Function

### Binet's formula method:

Python Program for n-th Fibonacci number Using Direct Formula The formula for finding the n-th Fibonacci number is as follows:

$$F_n = \frac{1}{\sqrt{5}} \left( \left( \frac{1 + \sqrt{5}}{2} \right)^n - \left( \frac{1 - \sqrt{5}}{2} \right)^n \right)$$

*Algorithm Description:*

function binet\_fibonacci(n):

    set precision for decimal arithmetic to  $2 * n + 1$

```
sqrt_5 = square root of 5  
phi = (1 + sqrt_5) / 2  
fib_n = (phi * n - (-1 / phi) * n) / sqrt_5
```

```
return round(fib_n)
```

*Implementation:*

```
# 8. Binet's Formula approach  
def binet_fib(n):  
    decimal.getcontext().prec = 2 * n + 1 # Set precision to handle large numbers  
  
    sqrt_5 = decimal.Decimal(math.sqrt(5))  
    phi = (1 + sqrt_5) / 2  
    fib_n = ( phi * n - (-1 / phi) * n) / sqrt_5  
    return round(fib_n)
```

*Figure 22 Fibonacci Binet Approach 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:

Time result of binet\_fib function

n-th Fibonacci termen	Time (s)
1690.0	0.0
7722.0	0.001002
7748.0	0.0
11368.0	0.0
11882.0	0.0
11896.0	0.001462
13187.0	0.0
13253.0	0.0
16463.0	0.0
16612.0	0.001429
17059.0	0.0
19320.0	0.0
20387.0	0.001454
21368.0	0.000506
21829.0	0.0
22497.0	0.001007
22552.0	0.0
23443.0	0.001
23862.0	0.0
27466.0	0.001
28146.0	0.0
29305.0	0.001
29633.0	0.0
30559.0	0.000998
30738.0	0.001
31959.0	0.0
33083.0	0.001002
33759.0	0.001
33777.0	0.000999
34125.0	0.0
34154.0	0.001001
35999.0	0.001
37750.0	0.001001
39186.0	0.001
41368.0	0.0
41457.0	0.000998
42446.0	0.001
44679.0	0.001
44766.0	0.0
47612.0	0.001

Figure 23 Results for second set of inputs

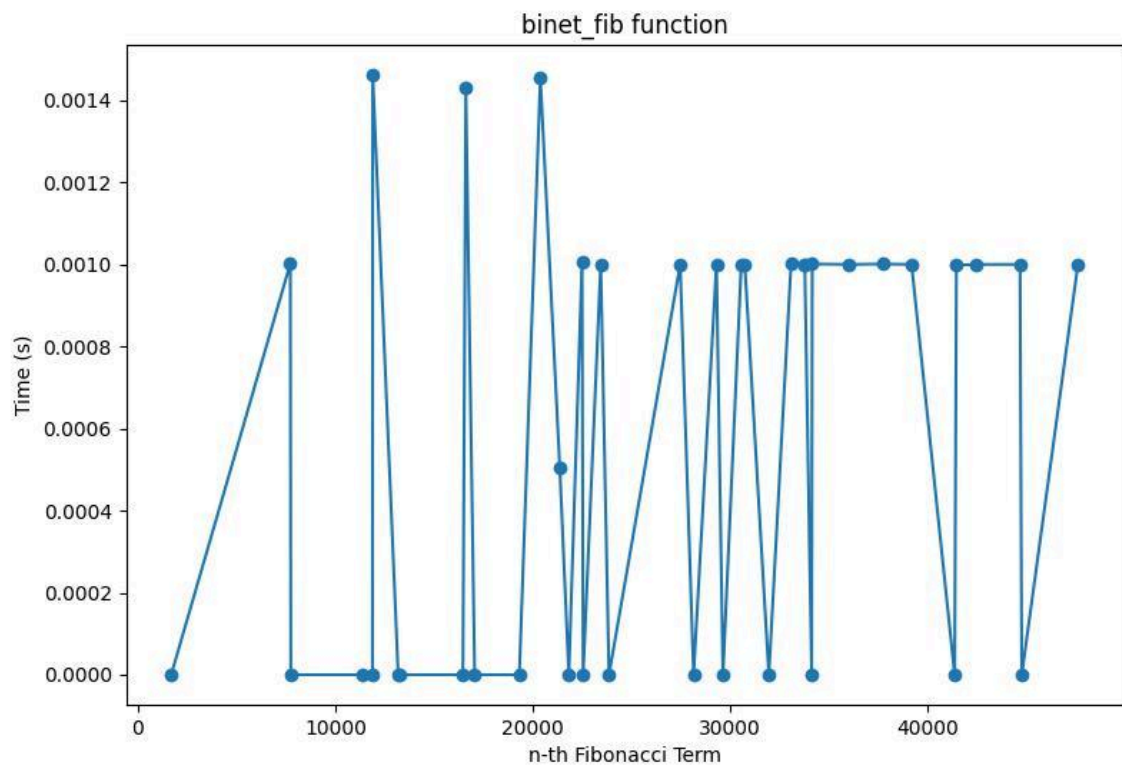


Figure 24 Graph of Binet Function

## CONCLUSION:

In conclusion, this laboratory work focused on conducting an empirical analysis of various Fibonacci methods, namely `recursive_fib`, `dynamic_fib`, `table_fib`, `iterative_fib`, `properties_fib`, `eigen_fib`, `matrix_fib`, and `binet_fib`, each designed to obtain the  $n$ -th Fibonacci term. Through rigorous benchmarking, we systematically evaluated and compared the performance of these methods.

Our findings revealed that among the tested methods, the `eigen_fib` method, leveraging eigenvalues, and the `matrix_fib` method emerged as the most efficient in terms of computational speed and resource utilization.

These results suggest that the eigenvalue approach and the matrix manipulation technique significantly outperformed other methods in producing Fibonacci terms, demonstrating their effectiveness in handling Fibonacci sequence computations. Moreover, the `binet_fib` method, which applies Binet's formula, also exhibited noteworthy performance, providing a viable alternative for Fibonacci term calculation. While not surpassing the eigenvalue and matrix methods, it showcased competitive results in our benchmarking analysis. In summary, our empirical analysis highlights the significance of choosing an appropriate Fibonacci method based on the specific requirements and constraints of the task at hand.

The eigenvalue method, matrix manipulation, and Binet's formula demonstrated superior performance in our benchmark, offering valuable insights for selecting efficient algorithms in Fibonacci term computations.