**Ministerul Educaţiei și Cercetării al Republicii Moldova**
**Universitatea Tehnică a Moldovei**
**Facultatea Calculatoare, Informatică și Microelectronică**

# REPORT

Laboratory work no.1
*Study and Empirical Analysis of Algorithms*
*for Determining Fibonacci N-th Term*

Elaborated:
st. gr. FAF-213 Botnari Ciprian

Verified:
asist. univ. Fiștic Cristofor

Chişinău – 2023

## TABLE OF CONTENTS

# ALGORITHM ANALYSIS

**Objective**

Study and analyze different algorithms for determining Fibonacci n-th term.

**Tasks**:

1. Implement at least 3 algorithms for determining Fibonacci n-th term;
2. Decide properties of input format that will be used for algorithm analysis;
3. Decide the comparison metric for the algorithms;
4. Analyze empirically the algorithms;
5. Present the results of the obtained data;
6. Deduce conclusions of the laboratory.

**Theoretical notes:**

An alternative approach to evaluating the complexity of an algorithm is through empirical analysis, which can provide insights on the complexity class of the algorithm, comparison of efficiency between algorithms solving the same problem, comparison of different implementations of the same algorithm, and performance on specific computers.

The process of empirical analysis typically involves:

1. Establishing the purpose of the analysis
2. Choosing the efficiency metric (number of operations or execution time)
3. Defining the properties of the input data
4. Implementing the algorithm in a programming language
5. Generating input data
6. Running the program with each set of data
7. Analyzing the results

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:**

Fibonacci numbers, commonly denoted $F_n$, form a sequence, the Fibonacci sequence, in which each number is the sum of the two preceding ones. The sequence commonly starts from 0 and 1, although some authors start the sequence from 1 and 1 or sometimes (as did Fibonacci) from 1 and 2. Starting from 0 and 1, the first few values in the sequence are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144.

The Fibonacci numbers were first described in Indian mathematics, as early as 200 BC in work by Pingala on enumerating possible patterns of Sanskrit poetry formed from syllables of two lengths. They are named after the Italian mathematician Leonardo of Pisa, later known as Fibonacci, who introduced the sequence to Western European mathematics in his 1202 book Liber Abaci.

Fibonacci numbers appear unexpectedly often in mathematics, so much so that there is an entire journal dedicated to their study, the Fibonacci Quarterly. Applications of Fibonacci numbers include computer algorithms such as the Fibonacci search technique and the Fibonacci heap data structure, and graphs called Fibonacci cubes used for interconnecting parallel and distributed systems. They also appear in biological settings, such as branching in trees, the arrangement of leaves on a stem, the fruit sprouts of a pineapple, the flowering of an artichoke, an uncurling fern, and the arrangement of a pine cone's bracts.

Fibonacci numbers are also strongly related to the golden ratio: Binet's formula expresses the nth Fibonacci number in terms of n and the golden ratio, and implies that the ratio of two consecutive Fibonacci numbers tends to the golden ratio as n increases. Fibonacci numbers are also closely related to Lucas numbers, which obey the same recurrence relation and with the Fibonacci numbers form a complementary pair of Lucas sequences.

Traditionally, the sequence was determined just by adding two predecessors to obtain a new number, however, with the evolution of computer science, several distinct algorithms for determination have been uncovered. These algorithms are:

1. **Recursion**
2. **Dynamic Programming**
3. **Power of the matrix + Optimized version**
4. **Binet’s formula**
5. **Kartik**
6. **Modulo**

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 6 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:
**6, 7, 8, 10, 11, 14, 16, 18, 19, 20, 22, 24, 25, 28, 33, 34, 35, 37, 39, 40**
to accommodate the recursive method, while the second series will have a bigger scope to be able to compare the other algorithms between themselves
**2006, 2026, 2546, 3998, 5568, 8768, 9577, 12409, 14557, 14827, 23497, 24253, 27233, 38982, 39359, 40879, 47018, 56137, 64163, 88847**

## IMPLEMENTATION

All six algorithms will be implemented 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 on memory of the device used. The error margin determined will constitute some seconds as per experimental measurement.

### Recursive

The recursive algorithm for determining the n-th Fibonacci number is one that uses a function that calls itself in order to calculate the desired result. The basic idea behind it is to use the definition of the Fibonacci sequence, which states that each number is the sum of the two preceding ones. This implementation uses a recursive function to calculate the nth Fibonacci number by breaking down the problem into smaller subproblems. If n is 0 or 1, the function returns n directly. Otherwise, it calculates the (n - 1) and (n - 2) Fibonacci numbers using two separate function calls, and then returns their sum as the result.

It's important to note that this implementation can be quite slow for large values of n, as it calculates many terms multiple times, leading to a large amount of redundant work. This can be addressed through the use of memoization, which caches intermediate results to avoid repeating calculations.

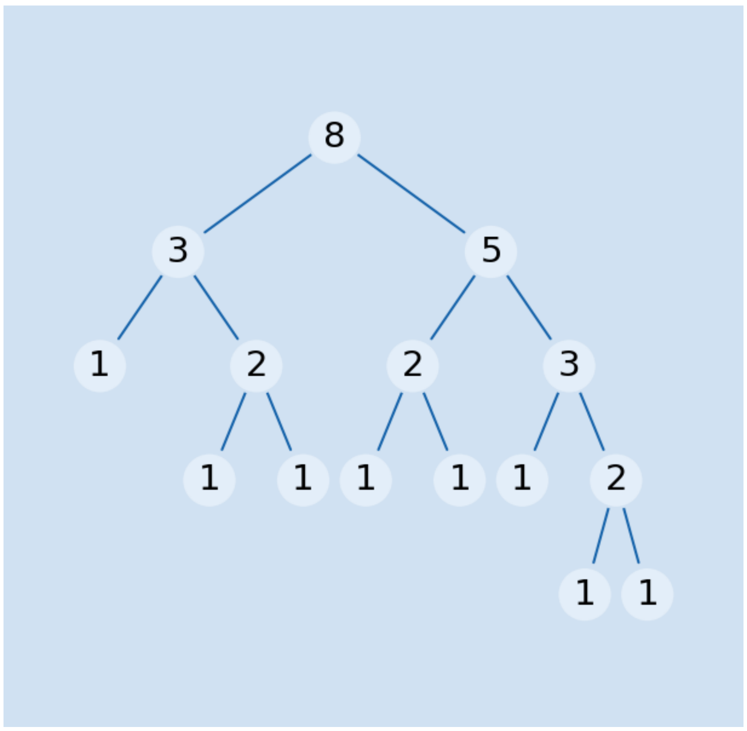


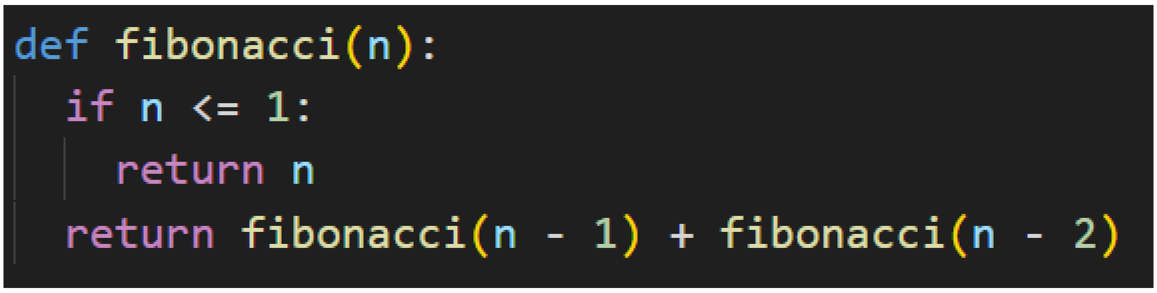
*Figure 1. Fibonacci Recursion*

### *Algorithm Description:*

The 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(n):
  if n <= 1:
    return n
  return fibonacci(n - 1) + fibonacci(n - 2)
```

*Figure 2. Fibonacci Recursion in Python*

***Results:***

| **n-th term, seconds** | **6** | **7** | **8** | **18** | **19** | **20** | **22** | **24** | **25** | **28** | **33** | **34** | **37** | **39** | **40** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ***Recursive*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.01*** | ***0.036*** | ***0.059*** | ***0.252*** | ***2.935*** | ***4.827*** | ***7.619*** | ***54.870*** | ***87.590*** |
| Dynamic | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Matrix | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Matrix optimized | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Binet | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Kartik | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Modulo | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |

The numbers in bold represent n-th Fibonacci term and the values for each Algorithm represent the time in seconds and I measure that using the function **perf_counter** from **time** module. We could notice the spike in time around 37-39 terms which tell a lot about the behavior and setbacks of this algorithm. From the graphs below we could deduce that the time complexity is T($2^n$), which means it is exponential.

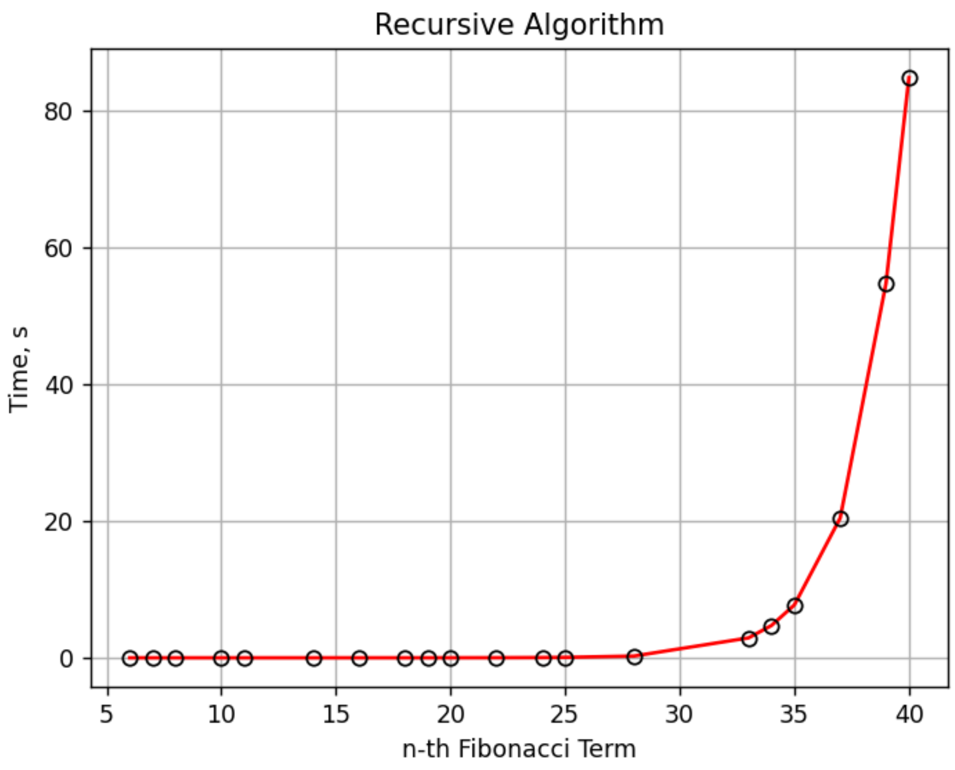


*Figure 3. Graph of Fibonacci Recursion*

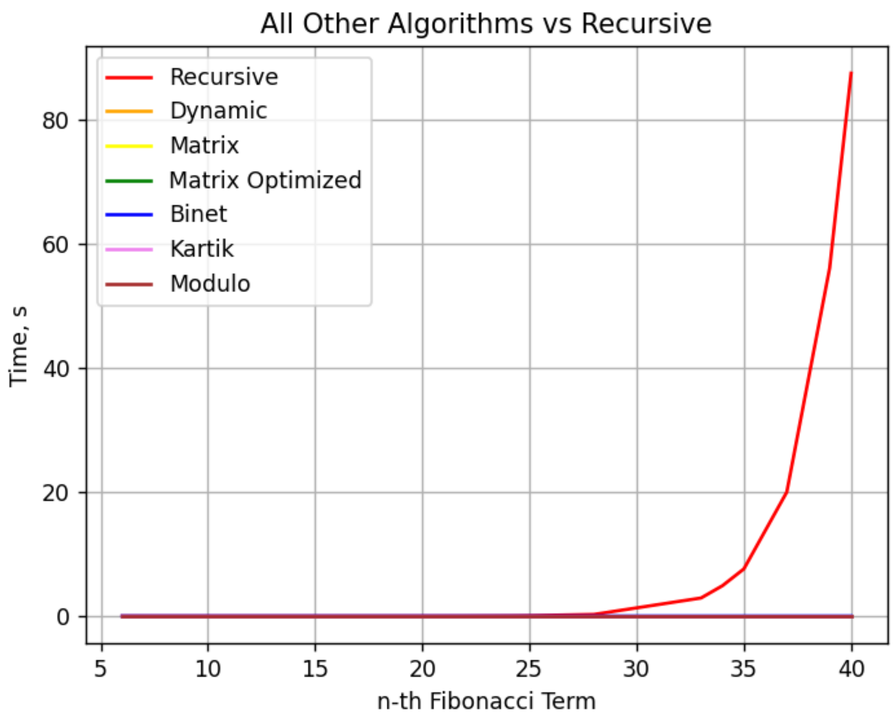


*Figure 4. Graph of Fibonacci Recursion vs other algorithms*

## Dynamic Programming

Dynamic Programming is a technique to solve problems by breaking them down into subproblems and storing their solutions in a cache (memoization) so that they can be reused later, instead of recalculating them.

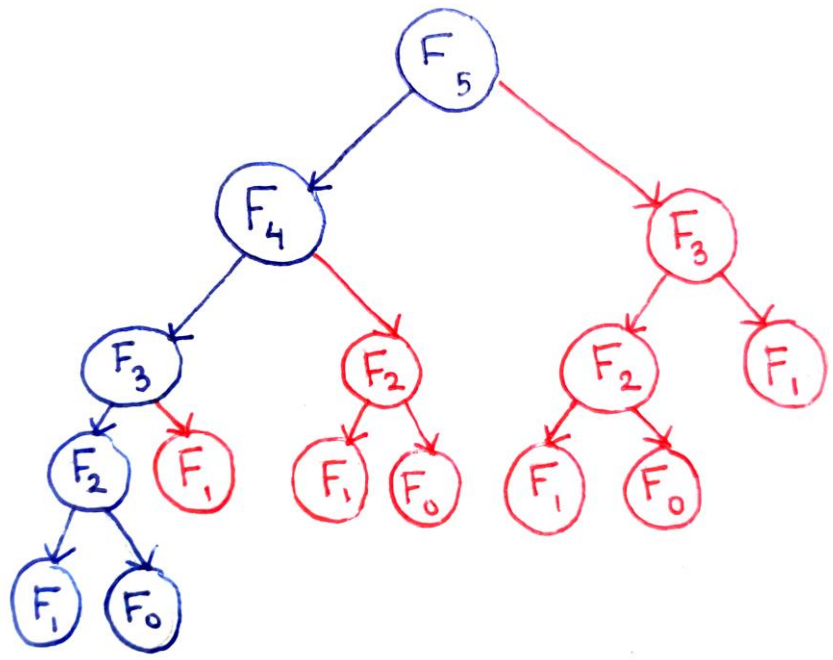


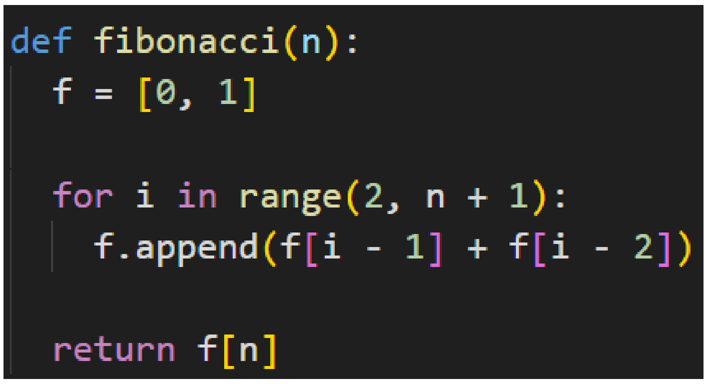
*Figure 5. Dynamic Programming Fibonacci (in blue)*

### *Algorithm Description:*

The dynamic programming Fibonacci method follows the algorithm as shown in the next 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:***



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

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

  return f[n]
```

*Figure 6. Dynamic Programming in Python*

***Results:***

| **n-th term, seconds** | **2006** | **2546** | **8768** | **9577** | **12409** | **14557** | **14827** | **23497** | **27233** | **38982** | **40879** | **47018** | **56137** | **64163** | **88847** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ***Dynamic*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.01*** | ***0.01*** | ***0.02*** | ***0.02*** | ***0.04*** | ***0.06*** | ***0.11*** | ***0.12*** | ***0.16*** | ***0.24*** | ***0.31*** | ***0.52*** |
| Matrix | 0.00 | 0.01 | 0.05 | 0.06 | 0.08 | 0.12 | 0.12 | 0.24 | 0.30 | 0.56 | 0.58 | 0.75 | 1.02 | 1.28 | 2.30 |
| Matrix optimized | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 |
| Binet | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Kartik | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | 0.02 | 0.03 | 0.04 | 0.07 | 0.07 | 0.13 |
| Modulo | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | 0.02 | 0.02 | 0.02 | 0.03 | 0.03 | 0.05 |

The numbers in bold represent n-th Fibonacci term and the values for each Algorithm represent the time in seconds and I measure that using the function **perf_counter** from **time** module. This algorithm has a time complexity of **T(n)** which is more efficient compared to the naive recursive approach which has a time complexity of **T($2^n$)**.

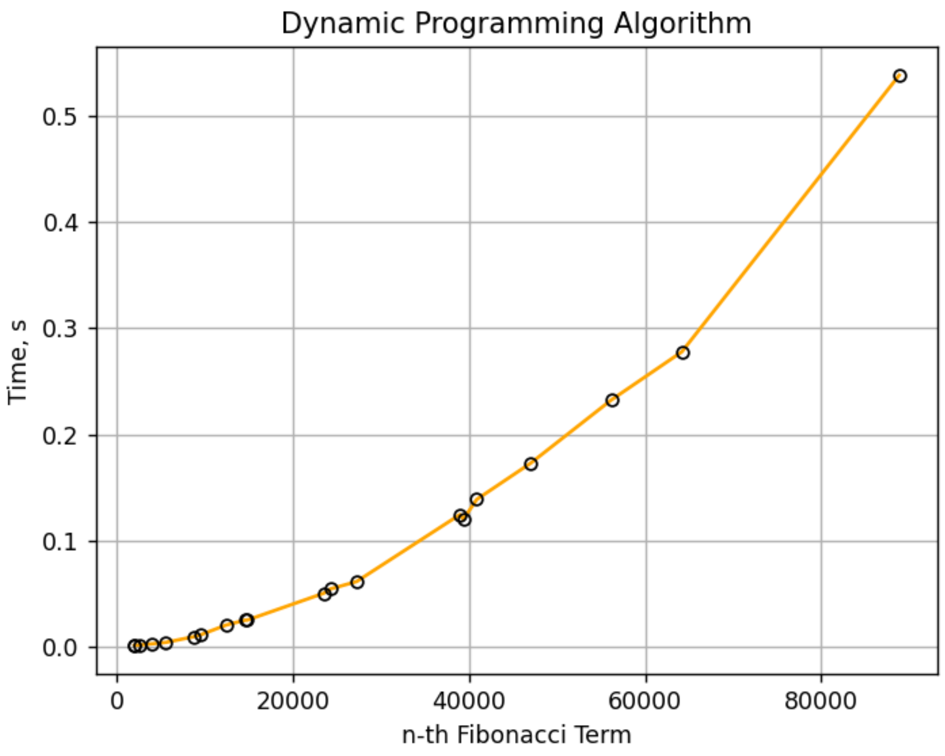


*Figure 7. Graph of Dynamic Programming*

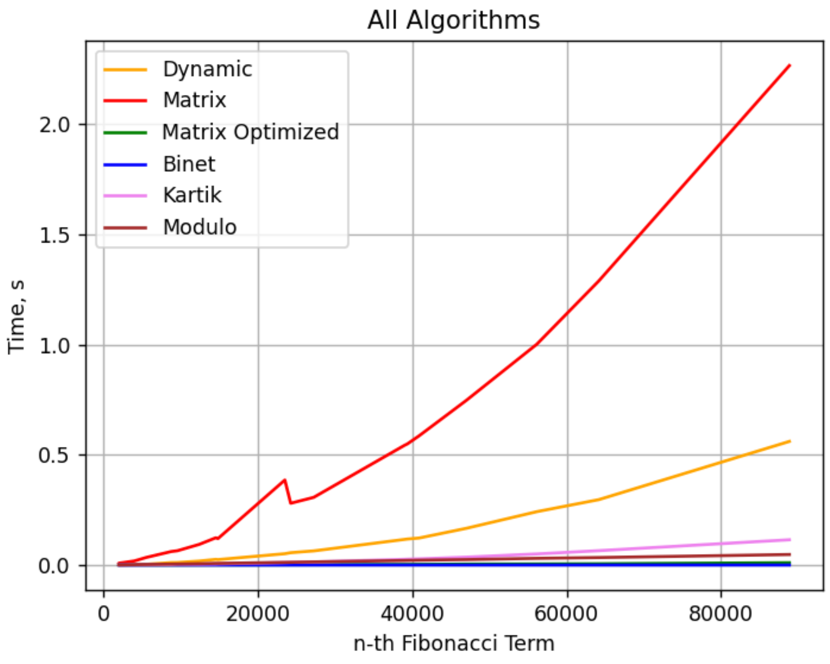


*Figure 8. Graph of Dynamic Programming vs other algorithms*

## Matrix Power

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.

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

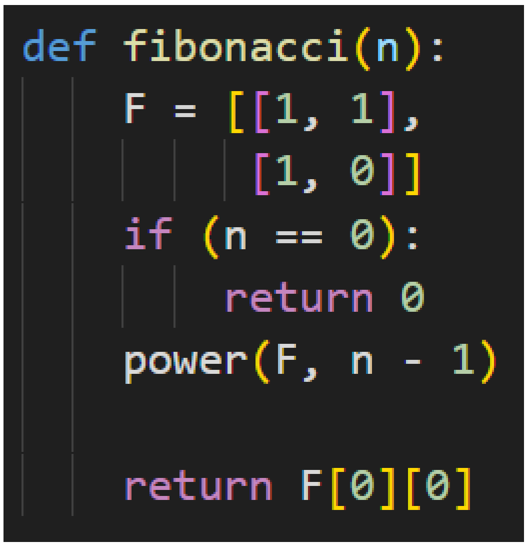
*Figure 9. Matrix Power Fibonacci*

### *Algorithm Description:*

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

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

### *Implementation:*



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

    return F[0][0]
```

*Figure 10. Matrix Power in Python*

### *Results:*

| **n-th term, seconds** | **2006** | **2546** | **8768** | **9577** | **12409** | **14557** | **14827** | **23497** | **27233** | **38982** | **40879** | **47018** | **56137** | **64163** | **88847** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dynamic | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | 0.02 | 0.02 | 0.04 | 0.06 | 0.11 | 0.12 | 0.16 | 0.24 | 0.31 | 0.52 |
| ***Matrix*** | ***0.00*** | ***0.01*** | ***0.05*** | ***0.06*** | ***0.08*** | ***0.12*** | ***0.12*** | ***0.24*** | ***0.30*** | ***0.56*** | ***0.58*** | ***0.75*** | ***1.02*** | ***1.28*** | ***2.30*** |
| Matrix optimized | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 |
| Binet | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Kartik | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | 0.02 | 0.03 | 0.04 | 0.07 | 0.07 | 0.13 |
| Modulo | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | 0.02 | 0.02 | 0.02 | 0.03 | 0.03 | 0.05 |

The numbers in bold represent n-th Fibonacci term and the values for each Algorithm represent the time in seconds and I measure that using the function **perf_counter** from **time** module. This algorithm has a time complexity of **T(n)** which is still better than recursive algorithm, but as we can see it still falls short when we compare it to other algorithms.

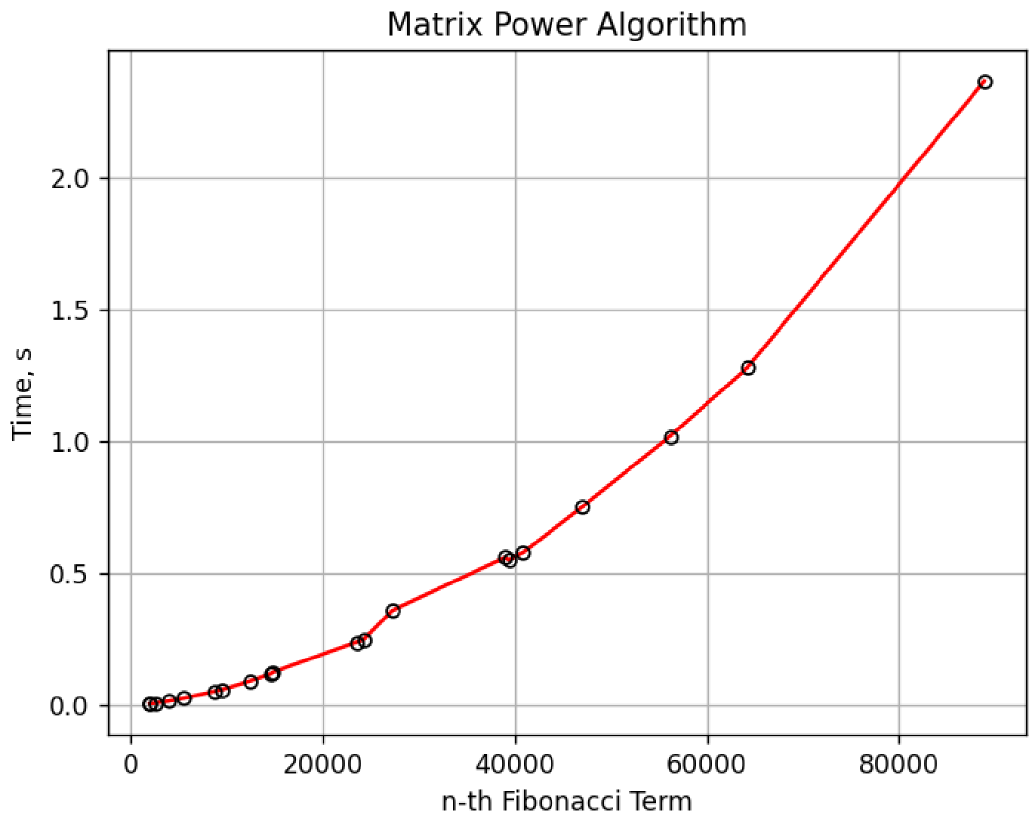


*Figure 11. Graph of Matrix Power*

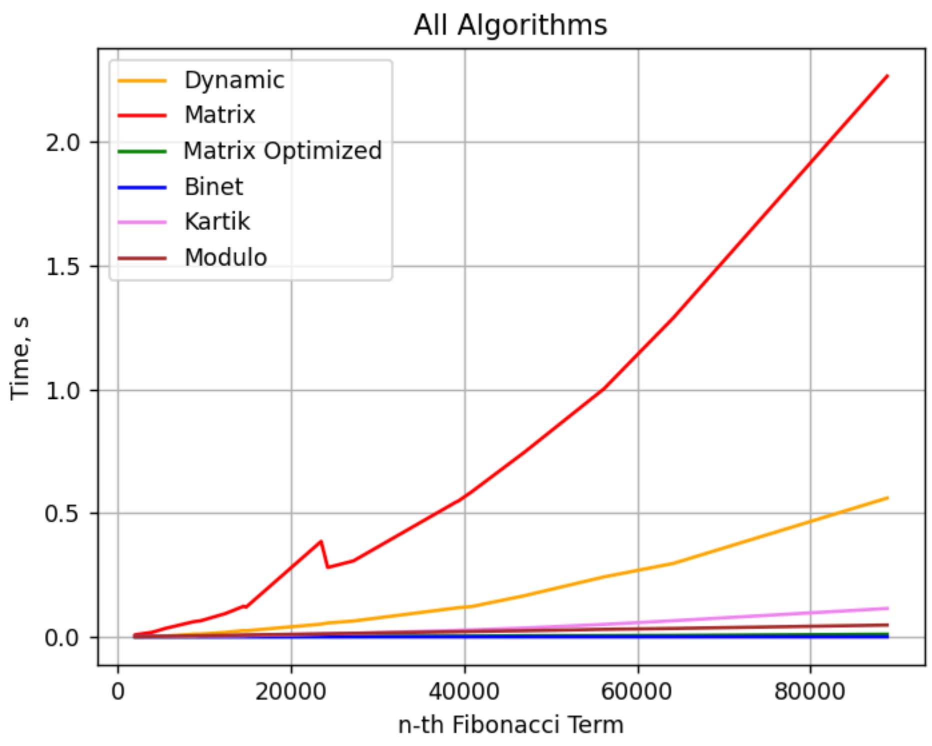


*Figure 12. Graph of Matrix Power vs other algorithms*

## Matrix Power Optimized

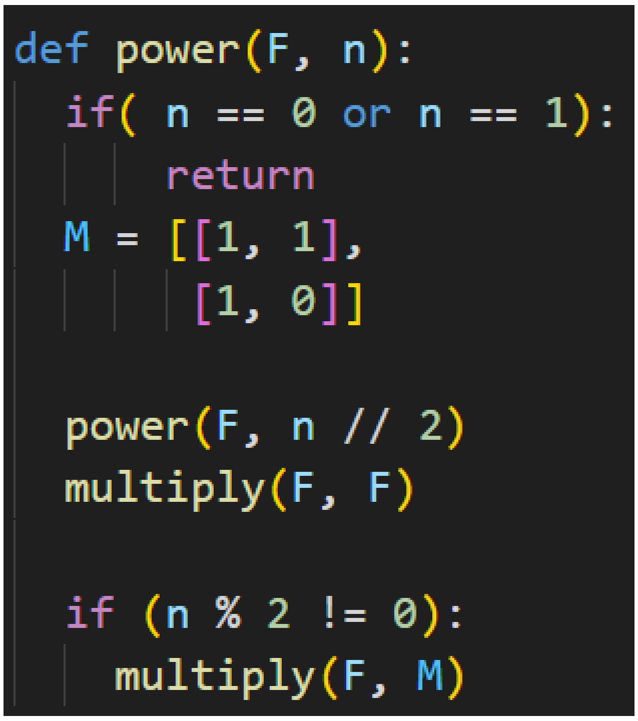
The Matrix Power optimized can be done using recursive multiplication to get power(M, n) in the previous method.

### *Optimization Description:*

The matrix power optimization follows the algorithm as shown in the next pseudocode:

```
Fibonacci(n):

     if n = 0 or n = 1
       Return

     Matrix <- [[1, 1],[1, 0]]

     power(F, n % 2)

     multiply(F, F)

     if (n mod 2 <> 0)
         multiply(F, Matrix)
```

***Optimization:***



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

    power(F, n // 2)
    multiply(F, F)

    if (n % 2 != 0):
        multiply(F, M)
```

*Figure 13. Matrix Power in Python*

***Results:***

| **n-th term, seconds** | **2006** | **2546** | **8768** | **9577** | **12409** | **14557** | **14827** | **23497** | **27233** | **38982** | **40879** | **47018** | **56137** | **64163** | **88847** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dynamic | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | 0.02 | 0.02 | 0.04 | 0.06 | 0.11 | 0.12 | 0.16 | 0.24 | 0.31 | 0.52 |
| Matrix | 0.00 | 0.01 | 0.05 | 0.06 | 0.08 | 0.12 | 0.12 | 0.24 | 0.30 | 0.56 | 0.58 | 0.75 | 1.02 | 1.28 | 2.30 |
| ***Matrix optimized*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.01*** |
| Binet | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Kartik | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | 0.02 | 0.03 | 0.04 | 0.07 | 0.07 | 0.13 |
| Modulo | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | 0.02 | 0.02 | 0.02 | 0.03 | 0.03 | 0.05 |

The numbers in bold represent n-th Fibonacci term and the values for each Algorithm represent the time in seconds and I measure that using the function **perf_counter** from **time** module. This algorithm has a time complexity of **T(log n)** which is still better than the non-optimized version and is one of the best ones out of compared ones.

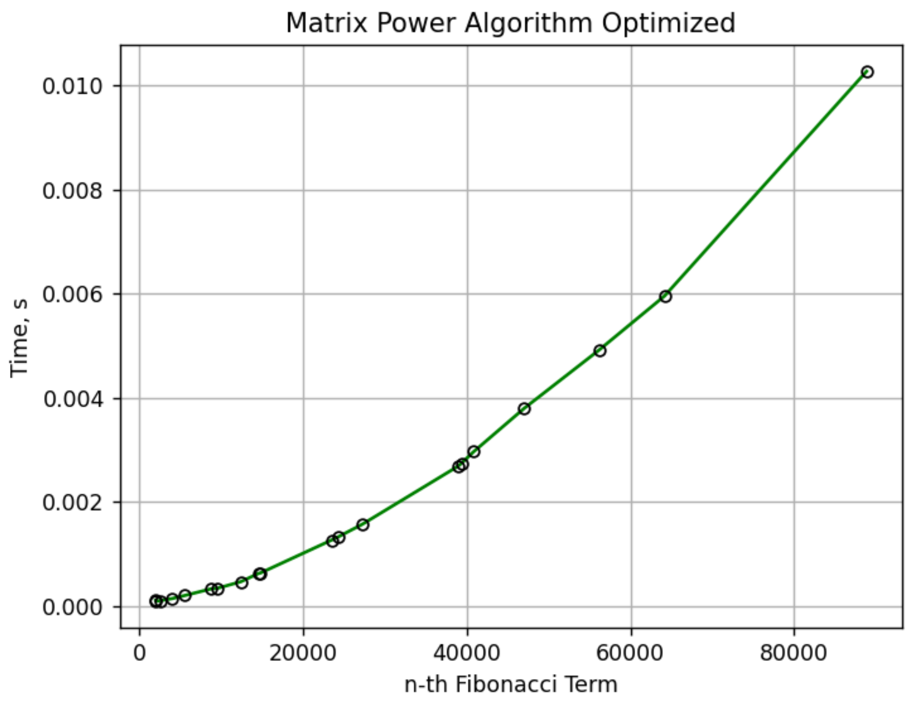


*Figure 14. Graph of Matrix Power Optimized*

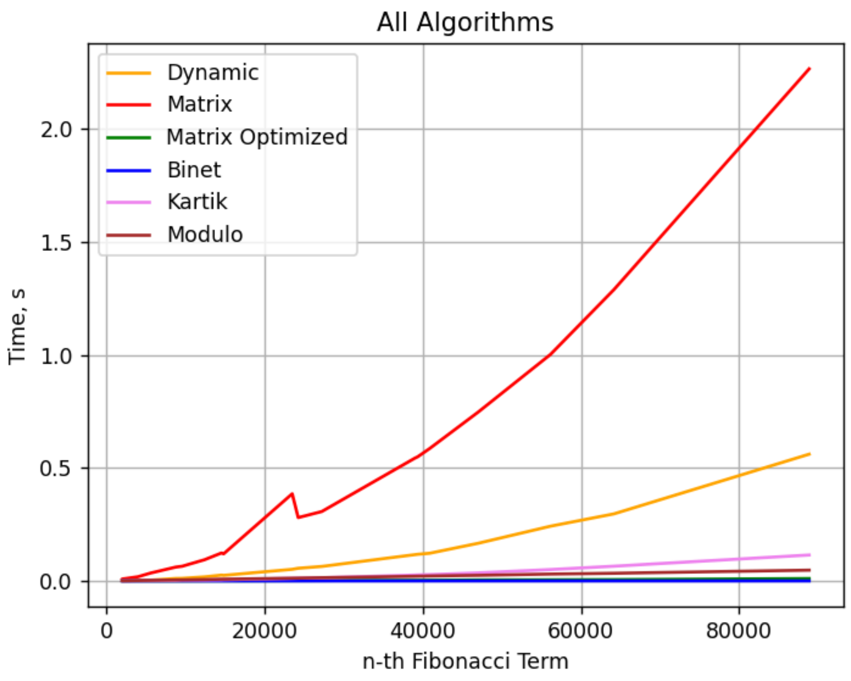


*Figure 15. Graph of Matrix Power Optimized vs other algorithms*

## Binet’s formula

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.

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

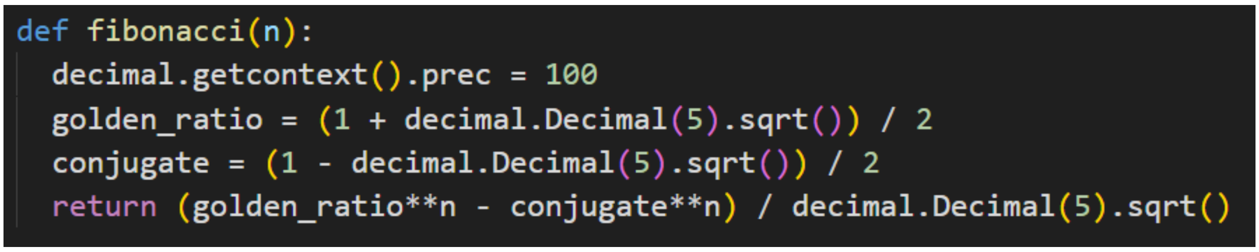
*Figure 16. Binet’s Formula Fibonacci*

### *Algorithm Description:*

The Binet’s formula follows the algorithm as shown in the next pseudocode:

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

### *Implementation:*



```python
def fibonacci(n):
  decimal.getcontext().prec = 100
  golden_ratio = (1 + decimal.Decimal(5).sqrt()) / 2
  conjugate = (1 - decimal.Decimal(5).sqrt()) / 2
  return (golden_ratio**n - conjugate**n) / decimal.Decimal(5).sqrt()
```

*Figure 17. Binet’s Formula in Python*

***Results:***

| **n-th term, seconds** | **2006** | **2546** | **8768** | **9577** | **12409** | **14557** | **14827** | **23497** | **27233** | **38982** | **40879** | **47018** | **56137** | **64163** | **88847** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dynamic | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | 0.02 | 0.02 | 0.04 | 0.06 | 0.11 | 0.12 | 0.16 | 0.24 | 0.31 | 0.52 |
| Matrix | 0.00 | 0.01 | 0.05 | 0.06 | 0.08 | 0.12 | 0.12 | 0.24 | 0.30 | 0.56 | 0.58 | 0.75 | 1.02 | 1.28 | 2.30 |
| Matrix optimized | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 |
| ***Binet*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** |
| Kartik | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | 0.02 | 0.03 | 0.04 | 0.07 | 0.07 | 0.13 |
| Modulo | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | 0.02 | 0.02 | 0.02 | 0.03 | 0.03 | 0.05 |

The numbers in bold represent n-th Fibonacci term and the values for each Algorithm represent the time in seconds and I measure that using the function **perf_counter** from **time** module. The Binet Formula Function is not accurate enough to be considered within the analyzed 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. Time complexity is **T(log n).**

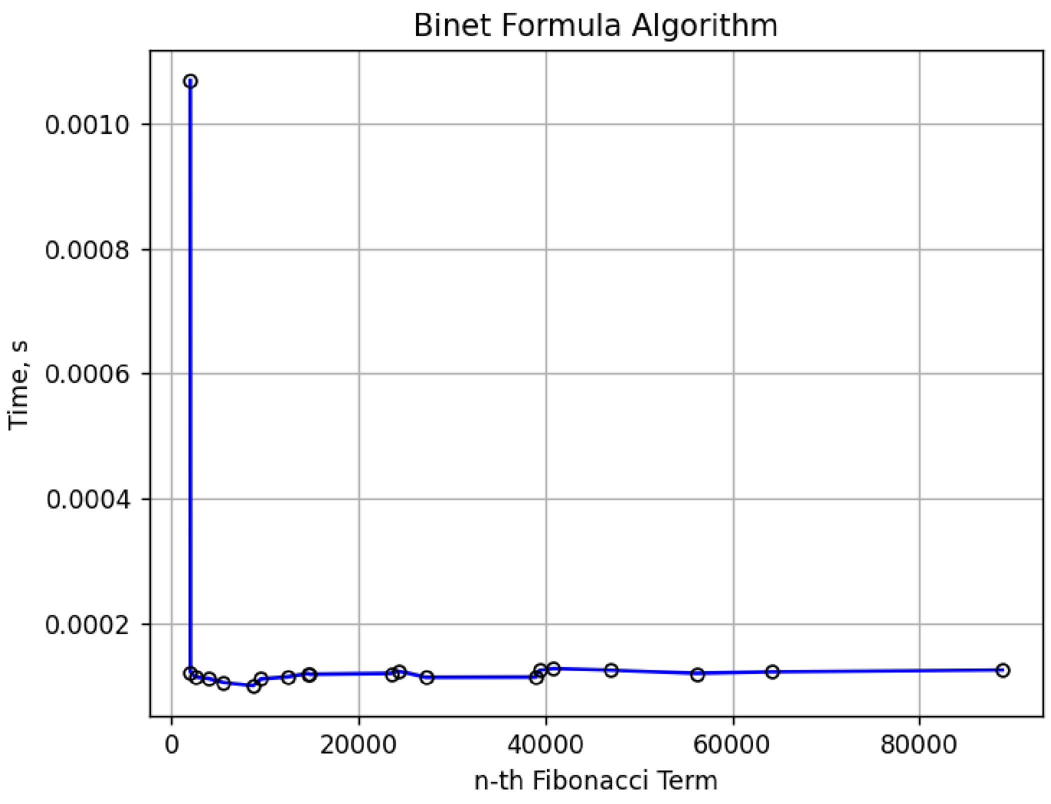


*Figure 18. Graph of Binet's Formula*

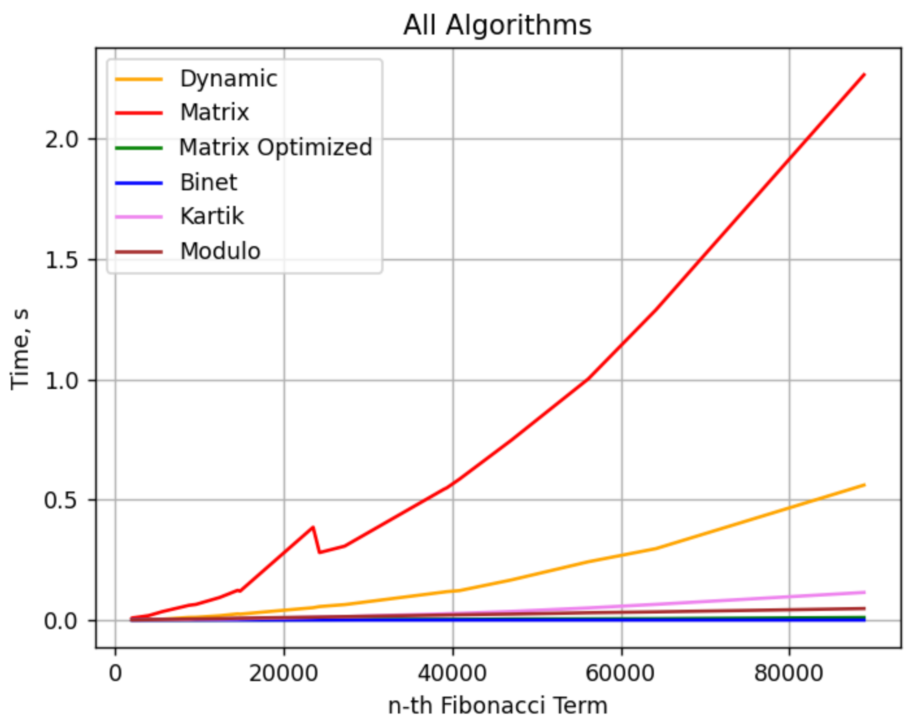
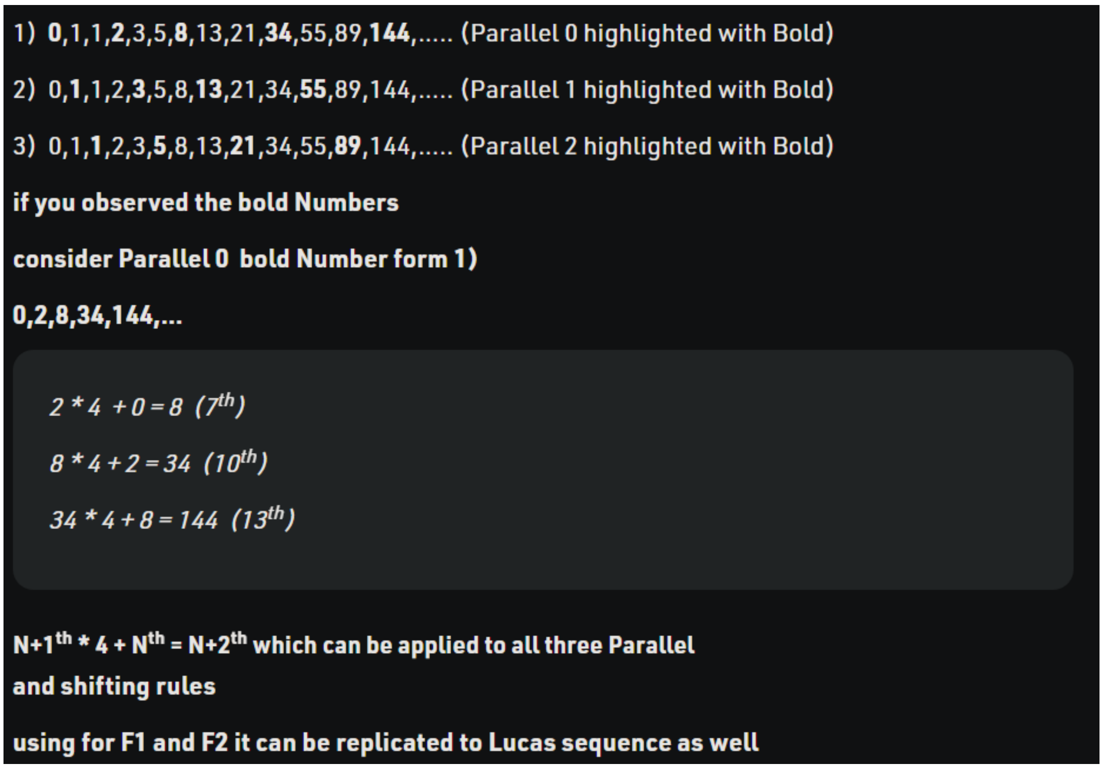


*Figure 19. Graph of Binet's Formula vs other algorithms*

## Kartik

Below is shown how we can deduce Fibonacci from the Kartik sequence.



1) **0**,1,1,**2**,3,5,**8**,13,21,**34**,55,89,**144**,..... (Parallel 0 highlighted with Bold)

2) 0,**1**,1,2,**3**,5,8,**13**,21,34,**55**,89,144,..... (Parallel 1 highlighted with Bold)

3) 0,1,**1**,2,3,**5**,8,13,**21**,34,55,**89**,144,..... (Parallel 2 highlighted with Bold)

**if you observed the bold Numbers**

**consider Parallel 0 bold Number form 1)**

**0,2,8,34,144,...**

$2 * 4 + 0 = 8$ $(7^{th})$

$8 * 4 + 2 = 34$ $(10^{th})$

$34 * 4 + 8 = 144$ $(13^{th})$

**$N+1^{th} * 4 + N^{th} = N+2^{th}$ which can be applied to all three Parallel and shifting rules**

**using for F1 and F2 it can be replicated to Lucas sequence as well**

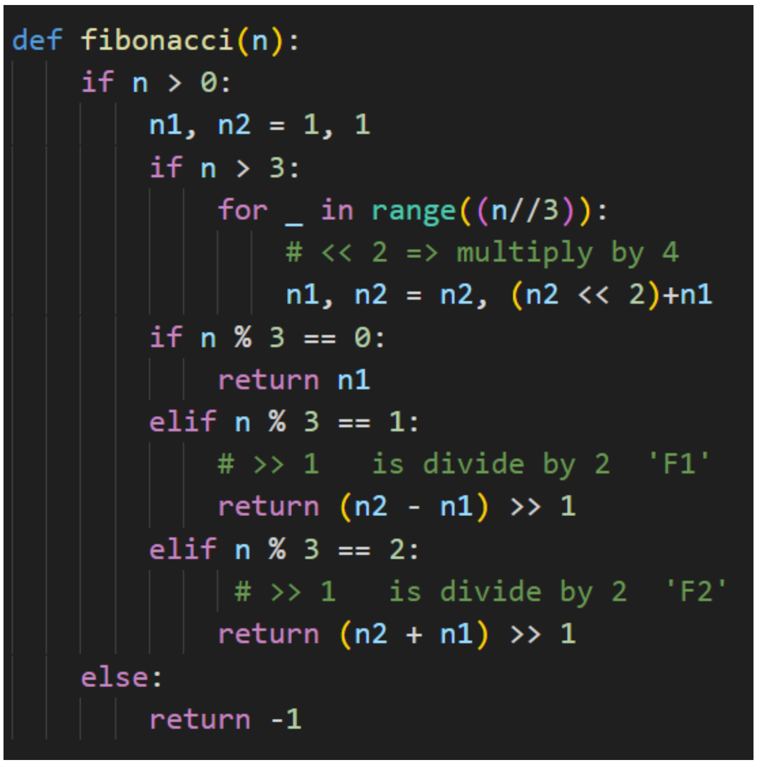
*Figure 20. Kartik Fibonacci*

## *Algorithm Description:*

The Kartik sequence follows the algorithm as shown in the next pseudocode:

```
Fibonacci(n):

     if n > 0 then
        n2 <- 1
        n1 <- n2
             if n > 3 then
                 for i to n div 3 do
                    n2 <- (n2 << 2)+n1
                    n1 <- n2
             if n mod 3 == 0 then
                 return n1
             otherwise n mod 3 == 1 then
                 return (n2 - n1) >> 1
             otherwise n mod 3 == 2 then
                 return (n2 + n1) >> 1
     otherwise:
        return -1
```

## *Implementation:*



```python
def fibonacci(n):
    if n > 0:
        n1, n2 = 1, 1
        if n > 3:
            for _ in range((n//3)):
                # << 2 => multiply by 4
                n1, n2 = n2, (n2 << 2)+n1
        if n % 3 == 0:
            return n1
        elif n % 3 == 1:
            # >> 1   is divide by 2  'F1'
            return (n2 - n1) >> 1
        elif n % 3 == 2:
             # >> 1   is divide by 2  'F2'
            return (n2 + n1) >> 1
    else:
        return -1
```

*Figure 21. Kartik sequence in Python*

## ***Results:***

| **n-th term, seconds** | **2006** | **2546** | **8768** | **9577** | **12409** | **14557** | **14827** | **23497** | **27233** | **38982** | **40879** | **47018** | **56137** | **64163** | **88847** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dynamic | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | 0.02 | 0.02 | 0.04 | 0.06 | 0.11 | 0.12 | 0.16 | 0.24 | 0.31 | 0.52 |
| Matrix | 0.00 | 0.01 | 0.05 | 0.06 | 0.08 | 0.12 | 0.12 | 0.24 | 0.30 | 0.56 | 0.58 | 0.75 | 1.02 | 1.28 | 2.30 |
| Matrix optimized | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 |
| Binet | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| ***Kartik*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.01*** | ***0.01*** | ***0.02*** | ***0.03*** | ***0.04*** | ***0.07*** | ***0.07*** | ***0.13*** |
| Modulo | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | 0.02 | 0.02 | 0.02 | 0.03 | 0.03 | 0.05 |

The numbers in bold represent n-th Fibonacci term and the values for each Algorithm represent the time in seconds and I measure that using the function **perf_counter** from **time** module. Time complexity is in between T(log n) and T(n) or T(n/3)

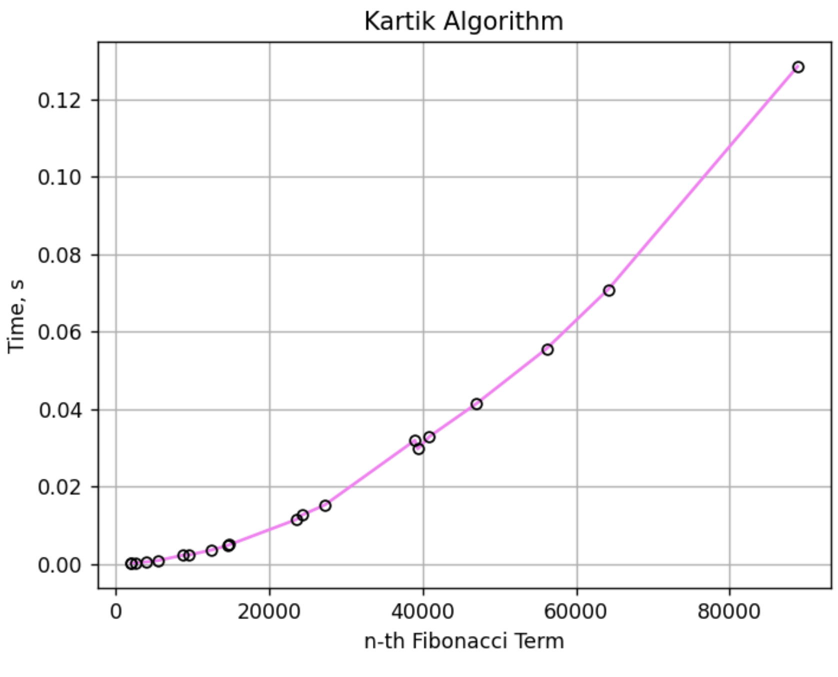


*Figure 22. Graph of Kartik sequence*

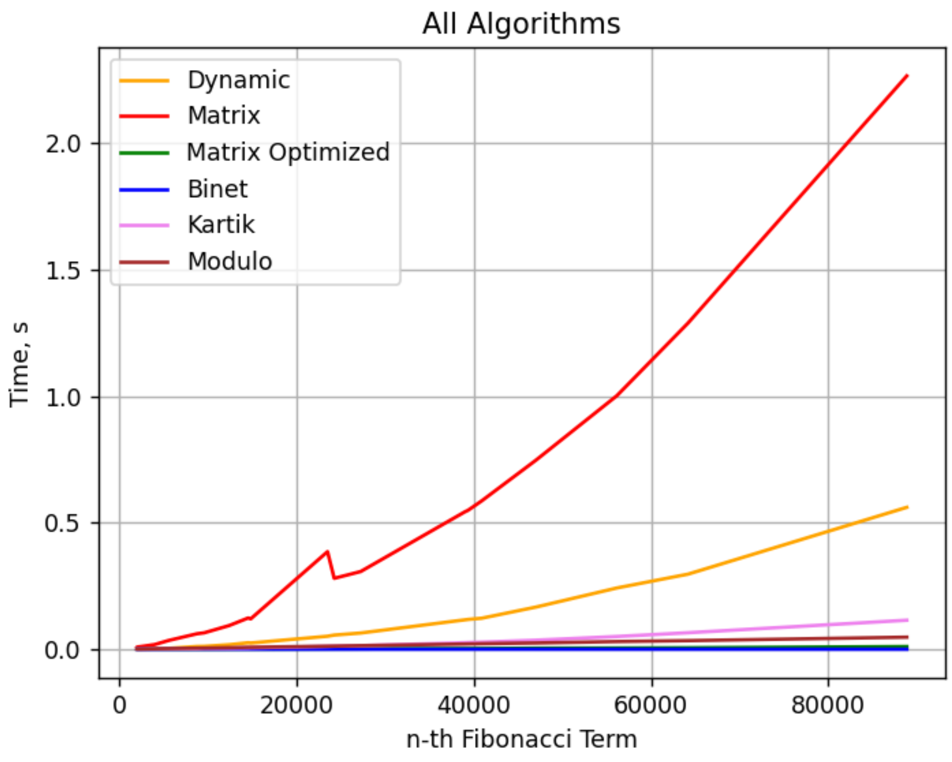


*Figure 23. Graph of Kartik sequence vs other algorithms*

## Modulo

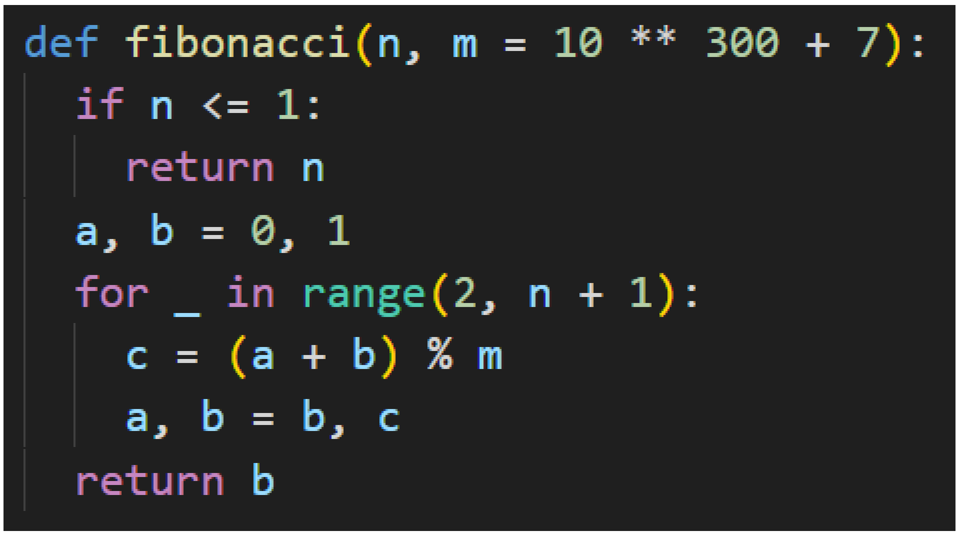
The modulo Fibonacci algorithm is a mathematical technique used to calculate the nth Fibonacci number modulo m, where n is a non-negative integer and m is a positive integer. The algorithm uses the properties of the Fibonacci sequence and modular arithmetic to efficiently compute the result in a much faster time complexity compared to traditional methods. The output of the modulo Fibonacci algorithm is used in various applications such as cryptography and random number generation.

### *Algorithm Description:*

The Kartik sequence follows the algorithm as shown in the next pseudocode:

```
Fibonacci(n):

     if n <= 1 then
         return n
     b <- 1
     a <- b
       for i to n + 1 do
         c <- (a + b) mod m
         a <- b
         b <- c
     return b
```

***Implementation:***



```python
def fibonacci(n, m = 10 ** 300 + 7):
    if n <= 1:
        return n
    a, b = 0, 1
    for _ in range(2, n + 1):
        c = (a + b) % m
        a, b = b, c
    return b
```

*Figure 24. Modulo in Python*

***Results:***

| **n-th term, seconds** | **2006** | **2546** | **8768** | **9577** | **12409** | **14557** | **14827** | **23497** | **27233** | **38982** | **40879** | **47018** | **56137** | **64163** | **88847** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dynamic | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | 0.02 | 0.02 | 0.04 | 0.06 | 0.11 | 0.12 | 0.16 | 0.24 | 0.31 | 0.52 |
| Matrix | 0.00 | 0.01 | 0.05 | 0.06 | 0.08 | 0.12 | 0.12 | 0.24 | 0.30 | 0.56 | 0.58 | 0.75 | 1.02 | 1.28 | 2.30 |
| Matrix optimized | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 |
| Binet | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| Kartik | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | 0.02 | 0.03 | 0.04 | 0.07 | 0.07 | 0.13 |
| ***Modulo*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.00*** | ***0.01*** | ***0.01*** | ***0.02*** | ***0.02*** | ***0.02*** | ***0.03*** | ***0.03*** | ***0.05*** |

The numbers in bold represent n-th Fibonacci term and the values for each Algorithm represent the time in seconds and I measure that using the function **perf_counter** from **time** module. The time complexity of the modulo Fibonacci algorithm is O(log n), which means that the time it takes to calculate the nth Fibonacci number modulo m grows logarithmically with the size of n. The use of modular arithmetic and the properties of the Fibonacci sequence allow the modulo Fibonacci algorithm to perform the calculation much more efficiently.

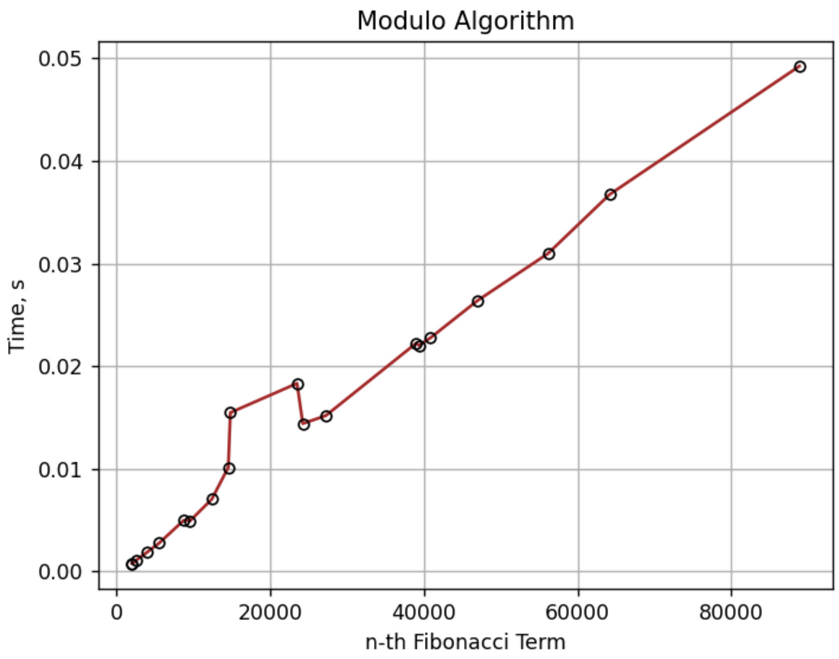


*Figure 25. Graph of Modulo*

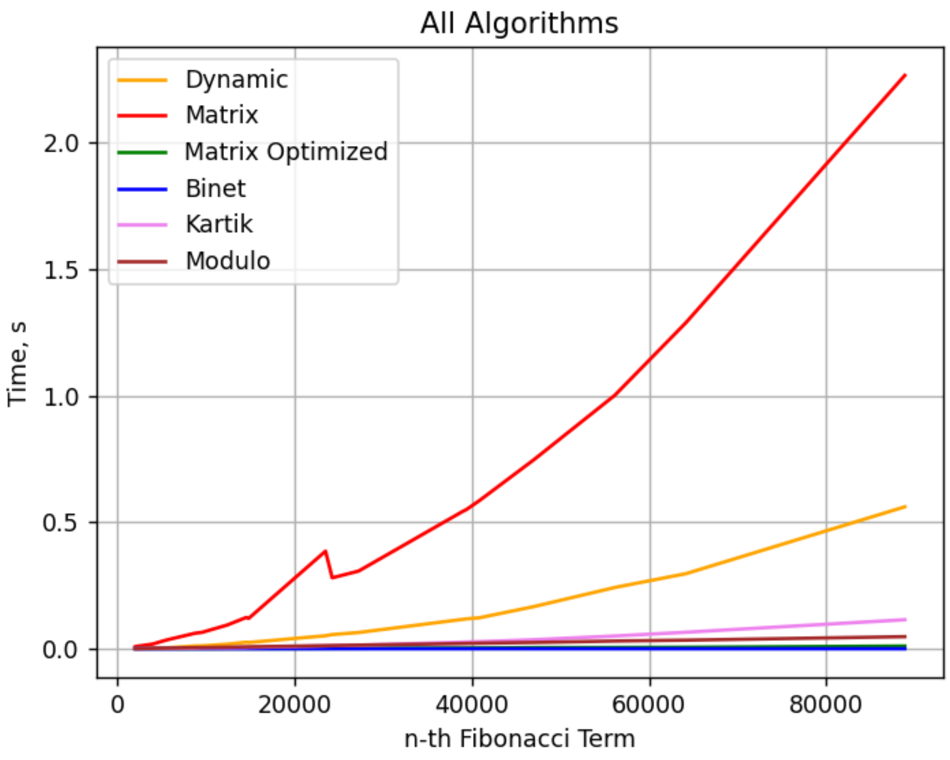


*Figure 26. Graph of Modulo vs other algorithms*

## CONCLUSION

**Recursive**: This is the most straightforward and intuitive approach to finding the nth Fibonacci term, but it is not the most efficient. The time complexity of this algorithm is $O(2^n)$, which can be very slow for large values of n. The main advantage of this algorithm is its simplicity, making it an excellent starting point for learning about Fibonacci numbers. It's best used for small values of n or for educational purposes.

**Dynamic**: This algorithm is an improvement over the recursive algorithm. It stores intermediate results in an array and uses them to calculate future values, reducing the time complexity to O(n). The main advantage of this algorithm is its improved time complexity, making it much faster for larger values of n. However, it requires additional memory to store the intermediate results, so its space complexity is O(n).

**Matrix Power Optimized**: This algorithm uses matrix multiplication to find the nth Fibonacci term in O(log n) time. The main advantage of this algorithm is its time efficiency, making it one of the fastest methods for finding the nth Fibonacci term. However, it can be difficult to understand and implement, making it less accessible to beginners.

**Binet's Formula**: This algorithm uses a mathematical formula to directly calculate the nth Fibonacci term in O(1) time. The main advantage of this algorithm is its constant time complexity, making it the fastest method for finding the nth Fibonacci term. However, it can be less intuitive than other methods and may not be suitable for learning about Fibonacci numbers.

**Kartik's Sequence**: This algorithm is a variation of the matrix power algorithm that reduces the number of matrix multiplications required. It has a time complexity of O(log n) and is faster than the standard matrix power algorithm.

**Modulo**: This algorithm uses modular arithmetic to calculate the nth Fibonacci term modulo m in O(log n) time. The main advantage of this algorithm is its time efficiency and its ability to find the result modulo m, which can be useful in certain applications such as cryptography and random number generation. However, it may not be suitable for all applications, and the choice of m can affect the results.

By analyzing the performance of the six algorithms for finding the nth Fibonacci term, we can see that each algorithm has its own strengths and weaknesses. Recursive and dynamic programming algorithms are simple to understand and implement, but they have poor time complexity for larger values of n. Matrix power, Binet's formula, and Kartik's sequence algorithms have fast time complexity but may require more complex implementations. The modulo algorithm is efficient for finding the result modulo m, but the choice of m can affect the results. In conclusion, each algorithm has its own use cases and the choice of algorithm depends on the specific requirements of the application.