MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

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

Elaborated:
st. gr. FAF-232 Alexei Maxim

Verified:
asist. univ. Fiștic Cristofor

Chișinău - 2025

Content

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 the algorithms empirically;
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).

Therefore, within this laboratory, we will analyze seven algorithms for determining the Fibonacci N-th term 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 are 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 seven algorithms will be implemented in python an analyzed empirically based on the time required for their completion. Although the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport to input will vary depending on the memory of the device used.

The error margin determined will constitute 2.5 seconds as per experimental measurement.

Recursive Method

The recursive method for computing the Fibonacci sequence adopts a natural approach by defining the problem in terms of itself. For any given n, the method computes the n-th term by first calculating the two preceding terms, then summing them to arrive at the final value. It does this by calling itself with arguments $n-1$ and $n-2$ until it reaches the base cases, typically when n is 0 or 1—where the Fibonacci number is already known. This self-referential structure highlights the elegance of recursion, wherein a complex problem is broken down into simpler, identical subproblems. Despite its intuitive formulation, the method results in the same operations being performed multiple times, which is an important consideration when analyzing its behavior and performance.

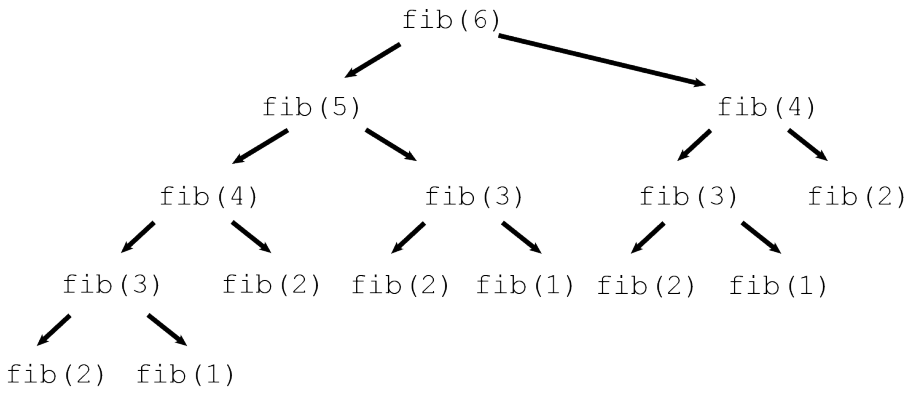


Figure 1 - Fibonacci Recursion Example

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)
```

Listing 1 - Pseudocode Recursive Fibonacci Implementation

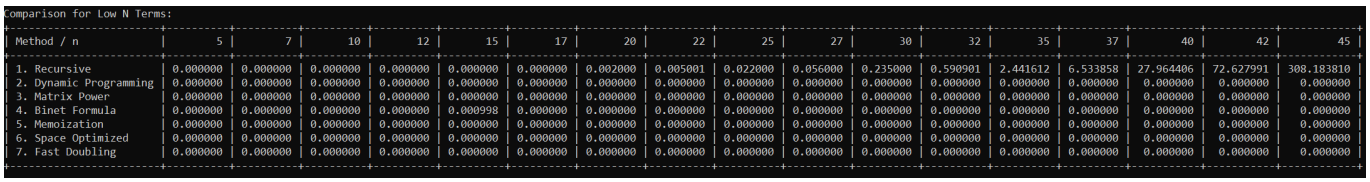
Implementation

```python
# Using the Recursive method
def recursive(n):
    if n <= 1:
        return n
    else:
        return recursive(n-1) + recursive(n-2)
```

Listing 2 - Recursive Fibonacci Implementation in Python

Results

After running the function for each n Fibonacci term proposed in the list from the first Input Format, also known as *Low N Terms list* in my implementation, and saving the time for each n, we obtained the following results:



Comparison for Low N Terms:

Method / n	5	7	10	12	15	17	20	22	25	27	30	32	35	37	40	42	45
1. Recursive	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.002000	0.005001	0.022000	0.056000	0.235000	0.590901	2.441612	6.533858	27.964406	72.627991	308.183810
2. Dynamic Programming	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
3. Matrix Power	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
4. Binet Formula	0.000000	0.000000	0.000000	0.000000	0.000998	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
5. Memoization	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
6. Space Optimized	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
7. Fast Doubling	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

Figure 2 - Fibonacci Recursion Table Output

In *Figure 2* is represented the table of results for the first set of inputs, with low n terms. The highest line, referring to 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 specified Fibonacci function was run till when the function was executed. Therefore, we notice that the only implementation that had its time growing astronomically is the *Recursive Implementation*, which starting from the 25th n, at 0.02 seconds, had the execution time grow up to roughly 308 seconds for the 45th computed term, while for this range of terms the other methods have no issue executing the task, as no change in time execution is seen for such a low value of the n-th term.

Furthermore, in the graph shown in *Figure 3*, we can clearly observe the rapid increase in execution time as n grows. While the recursive implementation remains relatively efficient for small values of n, we see an exponential surge in execution time starting from the 30th term. This spike becomes especially pronounced beyond the 42nd Fibonacci term, confirming the well-known exponential time complexity $T(2^n)$ of the naive recursive approach.

This behavior is a direct consequence of the excessive recomputation inherent in the recursive method, where the function repeatedly recalculates previously computed values instead of storing them for reuse. As a result, the number of function calls grows exponentially, making this approach infeasible for larger Fibonacci terms.

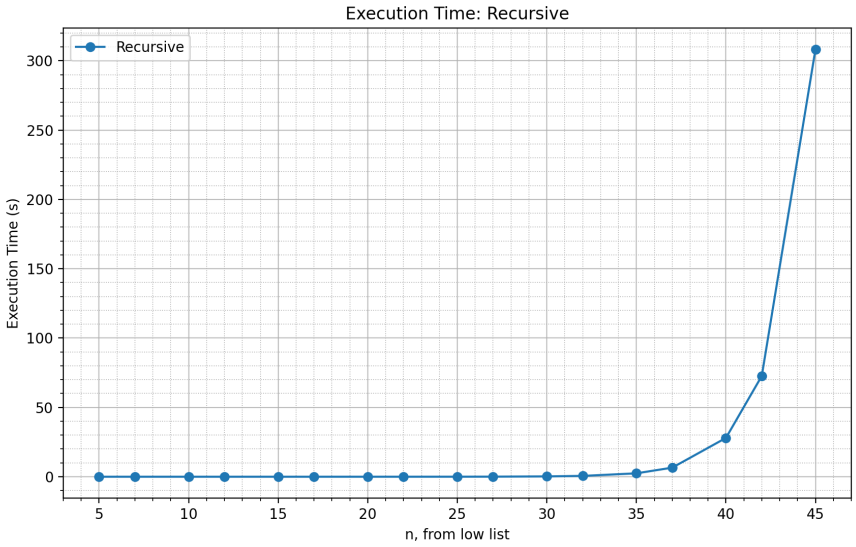


Figure 3 - Recursion Method Execution Time

We can also compare the results of the *Recursion Method* on this range with the other available methods, for a better visual understanding of the situation.

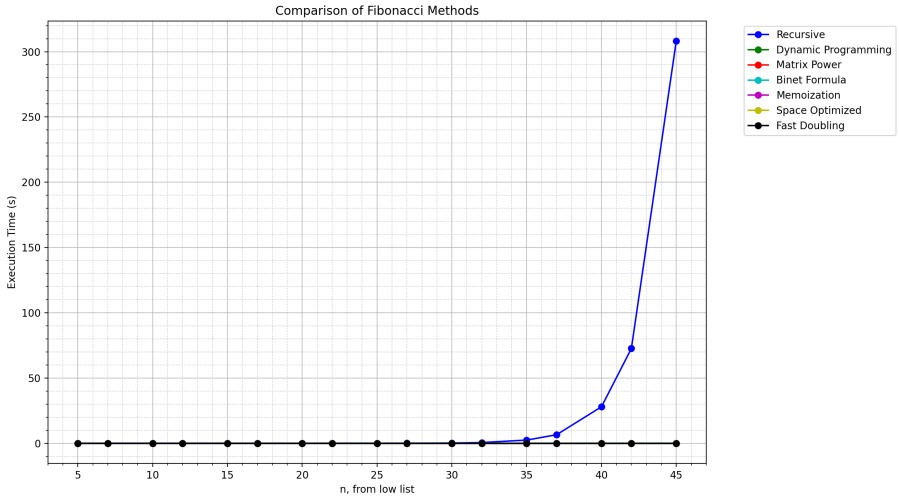


Figure 4 - Execution Time All Methods, Low *n* list

In contrast, all other methods presented in *Figure 4* maintain an execution time of essentially zero for this range of *n*, showcasing their efficiency. Techniques such as *Dynamic Programming*, *Memoization*, and *Matrix Exponentiation* leverage stored results or mathematical optimizations to drastically reduce the number of operations needed. Thus, while the recursive method serves as a useful theoretical demonstration of Fibonacci sequence generation, it is highly impractical for real-world applications involving large *n*.

Dynamic Programming Method

The Dynamic Programming (DP) approach, specifically the Bottom-Up method, is a systematic way of solving problems by building the solution iteratively from smaller subproblems. Instead of using recursion, which repeatedly recalculates the same values, the Bottom-Up approach starts with the smallest cases and works its way up to the desired result, storing intermediate values along the way.

When applied to the Fibonacci sequence, this method begins by initializing the first two terms ($F(0)$ and $F(1)$) and then iteratively computes each subsequent Fibonacci number by using the previously computed values. This eliminates unnecessary recalculations, ensuring that each term is derived only once and then stored for further use.

By structuring the computation in this way, the Bottom-Up approach should provide a more efficient and systematic method for calculating Fibonacci numbers, leveraging an array or simple variables to maintain previously computed terms.

Algorithm Description

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

```
Fibonacci_DP(n):
    if n <= 1:
        return n

    fib_list <- array of size (n + 1)
    fib_list[0] <- 0
    fib_list[1] <- 1

    for i from 2 to n:
        fib_list[I] <- fib_list[i - 1] + fib_list[i - 2]
    return fib_list[n]
```

Listing 3 - Pseudocode Dynamic Programming Implementation

Implementation

```
# Dynamic Programming implementation
def dynamic_programming(n):
    if n <= 1:
        return n

    # List to store Fibonacci nums
    li = [0] * (n + 1)

    # Initialize first two Fibonacci nums
    li[0] = 0
```

```
    li[1] = 1

    # Fill rest the list iteratively
    for i in range(2, n + 1):
        li[i] = li[i - 1] + li[i - 2]

    # Return the 'n' Fibonacci
    return li[n]
```

Listing 4 - Dynamic Programming Implementation in Python

Results

After running the function for each n Fibonacci term proposed in the list from the second Input Format, also known as *Medium N Terms list* in my implementation, and saving the time for each n, we obtained the following results:

Comparison for Medium N Terms:

Method / n	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
2. Dynamic Programming	0.000000	0.000000	0.000998	0.000000	0.000000	0.000000	0.000999	0.000000	0.000999	0.001000	0.001000	0.002001	0.003999	0.006003	0.008998	0.014999
3. Matrix Power	0.000000	0.001000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001000	0.000000	0.000000
4. Binet Formula	0.001000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001000	0.000000	0.001000	0.001001
5. Memoization	0.000000	0.000000	0.000000	0.000000	0.000999	0.000000	0.000000	0.000000	0.001015	0.000000	0.000990	0.000996	0.002999	0.003000	0.004000	0.007009
6. Space Optimized	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000994	0.000000	0.000000	0.000998	0.000999	0.002000	0.002000	0.004000	0.005001
7. Fast Doubling	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000999

Figure 5 - Dynamic Programming Table Output, medium n range

Comparison for High N Terms:

Method / n	0	4000	8000	24000	31000	50000	60000	72000	84000	95000	105000	117000	124000	140000	160000	170000	190000	220000	250000	300000	350000	400000	450000	500000
2. Dynamic Programming	0.000000	0.001001	0.003997	0.012000	0.048000	0.110000	0.171000	0.249000	0.344000	0.441117	0.537002	0.681601	0.773005	1.019555	1.318835	1.480810	3.134396	6.207786	9.439508	16.577718	28.084960	37.368466	68.885534	78.523404
3. Matrix Power	0.000000	0.002996	0.000000	0.001997	0.000997	0.003000	0.005000	0.004999	0.008000	0.009999	0.010000	0.012000	0.014001	0.019002	0.020998	0.026000	0.035002	0.035999	0.040999	0.057003	0.076005	0.083997	0.105668	0.135001
4. Binet Formula	0.009001	0.001998	0.001002	0.001995	0.003999	0.010000	0.014000	0.020001	0.027999	0.037003	0.041997	0.052997	0.060002	0.076000	0.098998	0.112000	0.137999	0.188003	0.241009	0.346997	0.477989	0.619000	0.789000	0.972004
5. Memoization	0.000000	0.004005	0.003995	0.095001	0.036001	0.186999	0.124002	0.204003	0.247003	0.235998	0.247999	0.133999	0.097997	0.323157	0.514998	0.293000	1.412000	0.786694	0.906482	1.858550	2.286274	2.780624	3.429888	7.508998
6. Space Optimized	0.001003	0.006000	0.008999	0.023000	0.010000	0.050001	0.058000	0.089000	0.117000	0.138999	0.155003	0.189995	0.211001	0.265001	0.344497	0.404001	0.504003	1.993995	4.236002	0.953526	13.437999	19.674001	25.511997	31.313202
7. Fast Doubling	0.001003	0.000998	0.000999	0.000000	0.000000	0.001003	0.001999	0.000996	0.003002	0.003999	0.003008	0.003992	0.004000	0.005000	0.007010	0.007991	0.010000	0.010000	0.013000	0.019000	0.025000	0.028020	0.035005	0.039991

Figure 6 - Dynamic Programming Table Output, high n range

In *Figure 5* and *Figure 6*, we observe the execution times for various methods when calculating Fibonacci numbers up to a medium-sized and high-sized provided range of n. According to the table, the Dynamic Programming (DP) method grows at a steady rate. Specifically, the DP row in the table increases in small increments, reflecting its iterative, bottom-up strategy. Unlike recursion, which recalculates previously solved subproblems, DP stores intermediate values, preventing redundant computations. This characteristic is evident in the near-constant growth pattern for the method's runtime as n increases.

Turning to the graph in *Figure 7* and *Figure 8*, we can see that the Dynamic Programming approach has a linear time execution complexity, $T(n)$. Its execution time curve rises in a nearly linear fashion, underscoring the benefit of building solutions from smaller subproblems. While methods such as Fast Doubling or Matrix Exponentiation,as seen in *Figure 30*, *Figure 31*, or *Figure 6*, may offer even faster performance, the DP approach remains a strong, intuitive option for calculating Fibonacci numbers, especially when balancing simplicity of implementation and computational efficiency.

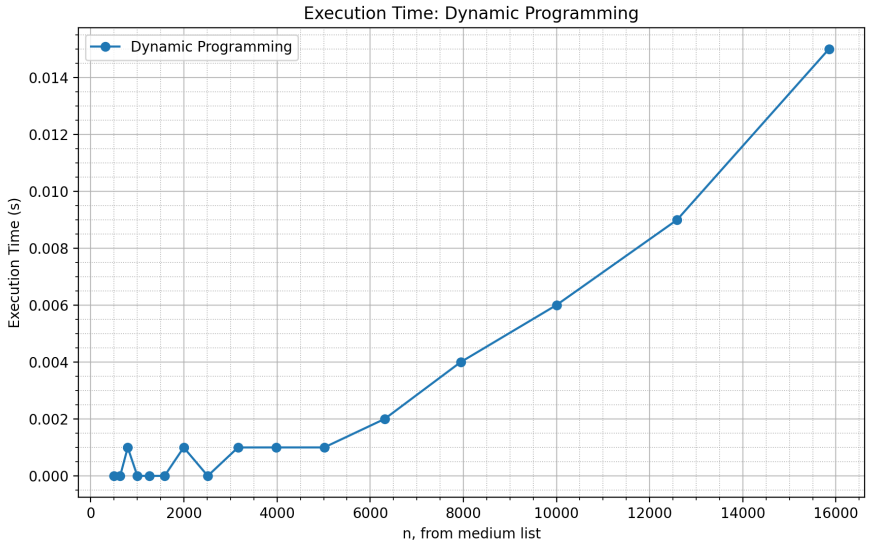


Figure 7 - Dynamic Programming Execution Time, medium *n* range

In *Figure 30* we see how Dynamic Programming approach behaves compared to the other implementations in a visual method, which makes things more clear.

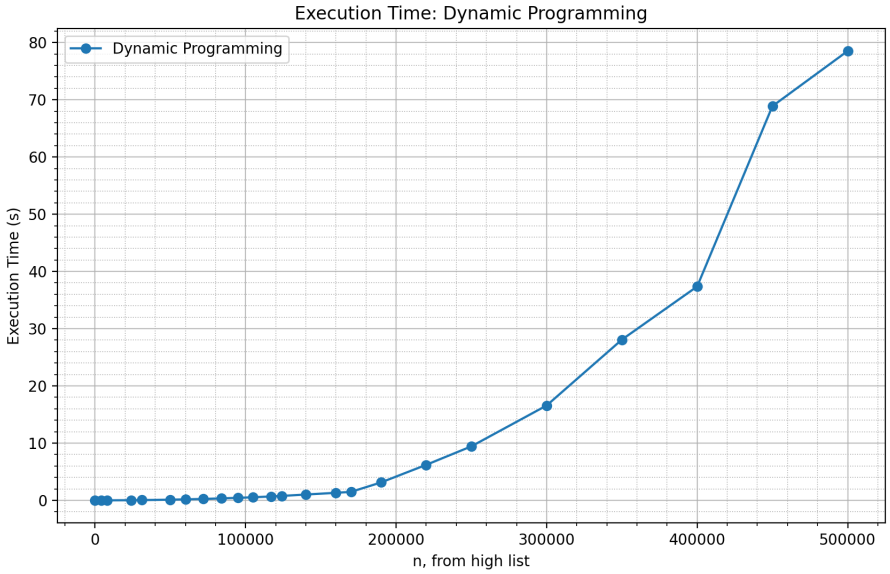


Figure 8 - Dynamic Programming Execution Time, high *n* range

The reason behind Dynamic Programming's predictable growth lies in its core principle: it computes each Fibonacci number once and stores the result. By using an array (or other data structures) to hold previously computed terms, the method avoids the repetitive function calls seen in the naive recursive approach. This systematic reuse of results translates into stable, incremental runtime increases, as confirmed by both the table and the graph. Thus, while not the absolute fastest, Dynamic Programming offers a robust middle ground—balancing clarity, ease of understanding, and efficient performance across a wide range of input sizes.

Fast Matrix Power Method

The matrix power algorithm, which technically should be more effective than the previous ones, leverages the fact that the Fibonacci sequence can be represented through matrix exponentiation. Given the matrix:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

raising this matrix to the power of n gives us another matrix where the top-left element contains the Fibonacci number $F(n+1)$, the top-right contains $F(n)$, the bottom-left contains $F(n)$, and the bottom-right contains $F(n-1)$.

So the essence of the whole algorithm is:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix}.$$

Algorithm Description

1. Matrix Representation: The Fibonacci sequence can be computed by repeated multiplication of the matrix $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
2. Exponentiation by Squaring: To efficiently compute high powers of the matrix, we use exponentiation by squaring. The basic idea is to reduce the number of multiplications required by breaking the power n down recursively:

   If n is even: $A^n = (A^{n/2}) \times (A^{n/2})$

   If n is odd: $A^n = A \times A^{n-1}$

   By recursively halving n in each step, we achieve a logarithmic number of multiplications, significantly speeding up the process.

Implementation

```
# Matrix Exponentiation implementation
def matrix_power(n):
    if n <= 1:
        return n

    # Initialize transformation matrix
    mat1 = [[1, 1],
            [1, 0]]
```

```
    # Raise the matrix 'mat1' to the power (n - 1)
    power(mat1, n - 1)

    return mat1[0][0]

# Multiplies two 2x2 matrices
def multiply(mat1, mat2):
    # Perform matrix multiplication
    x = (mat1[0][0] * mat2[0][0] +
         mat1[0][1] * mat2[1][0])
    y = (mat1[0][0] * mat2[0][1] +
         mat1[0][1] * mat2[1][1])
    z = (mat1[1][0] * mat2[0][0] +
         mat1[1][1] * mat2[1][0])
    w = (mat1[1][0] * mat2[0][1] +
         mat1[1][1] * mat2[1][1])

    # Update matrix mat1 with the result
    mat1[0][0], mat1[0][1] = x, y
    mat1[1][0], mat1[1][1] = z, w

# Performs matrix exponentiation
def power(mat1, n):
    if n <= 1:
        return

    # Initialize helper matrix
    mat2 = [[1, 1],
            [1, 0]]

    power(mat1, n // 2)
    multiply(mat1, mat1)

    # Recursively compute mat1^(n // 2)
    if n % 2 != 0:
        multiply(mat1, mat2)
```

Listing 5 - Matrix Power Implementation in Python

Results

The results of the research regarding the execution of this Fast Matrix Power method are explained in this section.

Method / n	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
2. Dynamic Programming	0.000000	0.000000	0.000998	0.000000	0.000000	0.000000	0.000999	0.000000	0.000999	0.001000	0.001000	0.002001	0.003999	0.006003	0.008998	0.014999
3. Matrix Power	0.000000	0.001000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001000	0.000000	0.000000
4. Binet Formula	0.001000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001000	0.000000	0.001000	0.001001
5. Memoization	0.000000	0.000000	0.000000	0.000000	0.000999	0.000000	0.000000	0.000000	0.001015	0.000000	0.000990	0.000996	0.002999	0.003000	0.004000	0.007009
6. Space Optimized	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000994	0.000000	0.000000	0.000998	0.000999	0.002000	0.002000	0.004000	0.005001
7. Fast Doubling	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000999

Figure 9 - Matrix Power Table Output, medium *n* range

Comparison for High N Terms:

Method / n	0	4000	8000	24000	31000	50000	60000	72000	84000	95000	105000	117000	124000	140000	160000	170000	190000	220000	250000	300000	350000	400000	450000	500000
2. Dynamic Programming	0.000000	0.001001	0.003997	0.032000	0.048000	0.118000	0.171000	0.249000	0.344000	0.441117	0.537002	0.681601	0.773005	1.019565	1.318835	1.480810	3.134396	6.207786	9.439508	16.577718	28.084960	37.368466	68.885534	78.523404
3. Matrix Power	0.000000	0.002996	0.000000	0.001997	0.000997	0.003000	0.005000	0.004999	0.008000	0.009999	0.010000	0.012000	0.014001	0.018002	0.020998	0.026000	0.035002	0.035999	0.040999	0.057003	0.076005	0.083997	0.105668	0.135001
4. Binet Formula	0.009001	0.001998	0.001002	0.001995	0.003999	0.010000	0.014000	0.020001	0.027999	0.037003	0.041997	0.052997	0.060002	0.076000	0.098998	0.112000	0.137999	0.188003	0.241009	0.346997	0.477989	0.619000	0.789000	0.972004
5. Memoization	0.000000	0.004005	0.003995	0.095001	0.036001	0.186999	0.124002	0.204003	0.247003	0.235998	0.247999	0.133999	0.097997	0.323157	0.514998	0.293000	1.412000	0.786694	0.906482	1.858550	2.286274	2.780624	3.429888	7.508998
6. Space Optimized	0.001003	0.006000	0.000999	0.023000	0.010000	0.050001	0.058000	0.089000	0.117000	0.138999	0.155003	0.189995	0.219001	0.265001	0.344497	0.404001	0.504003	1.993995	4.236002	8.953526	13.437999	19.674001	25.511997	31.313202
7. Fast Doubling	0.001003	0.000998	0.000999	0.000000	0.000000	0.001003	0.001999	0.000996	0.003002	0.003999	0.003008	0.003992	0.004000	0.005000	0.007010	0.007991	0.010000	0.010000	0.013000	0.019000	0.025000	0.028020	0.035005	0.039991

Figure 10 - Matrix Power Table Output, high *n* range

Matrix Power, as seen in *Figure 9* and *Figure 10*, is a more efficient approach for computing Fibonacci numbers compared to methods like Dynamic Programming and Memoization. In the charts, we see that the execution time for Matrix Power remains relatively stable even for larger *n* values, as opposed to methods like Dynamic Programming, which show a significant increase in execution time.

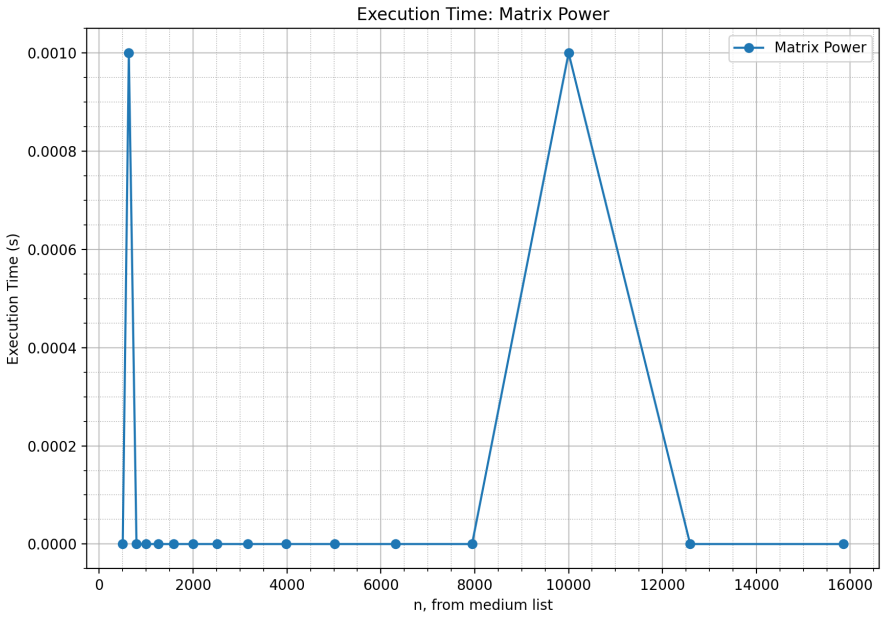


Figure 11 - Matrix Power Execution Time, medium *n* range

In *Figure 30* and *Figure 31* we see how Matrix Power approach behaves compared to the other implementations in a visual method, which makes things more clear.

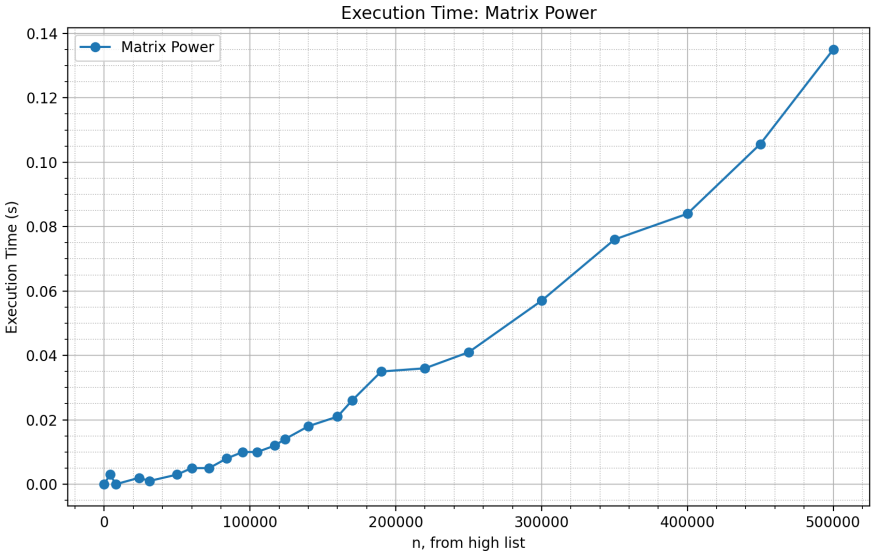


Figure 12 - Matrix Power Execution Time, high n range

The time complexity of the Matrix Power method is $T(logn)$, thanks to the exponentiation by squaring technique, as seen in *Figure 11* and *Figure 12*. This allows it to perform much better on larger inputs, which is reflected in the time complexity plot. The time complexity grows more slowly as n increases, compared to the linear growth seen in methods like Dynamic Programming. Therefore, Matrix Power remains a more scalable solution for larger Fibonacci numbers.

Iterative Space Optimized Method

In the traditional iterative Fibonacci algorithm, we often store all the Fibonacci numbers up to the desired index in an array, and then return the last element as the result. However, this consumes extra space, around O(n), since we only need the last two Fibonacci numbers at any point in the sequence to calculate the next one.

To optimize this, we can reduce the space usage by keeping track of only the previous two Fibonacci numbers. This way, instead of maintaining a full array of Fibonacci numbers, we update and reuse just two variables as we iterate through the sequence.

Algorithm Description

Approach:

1. Start with two variables, a and b, initialized to the first two Fibonacci numbers, i.e., $a = 0(F(0))$ and $b = 1(F(1))$.
2. For each subsequent number in the Fibonacci sequence, calculate the next Fibonacci number as the sum of a and b (i.e., $nextFib = a + b$).
3. After calculating the next Fibonacci number, update the values of a and b to represent the last two Fibonacci numbers. Set a to b, and b to $nextFib$.
4. Continue this process until the desired Fibonacci number is reached.

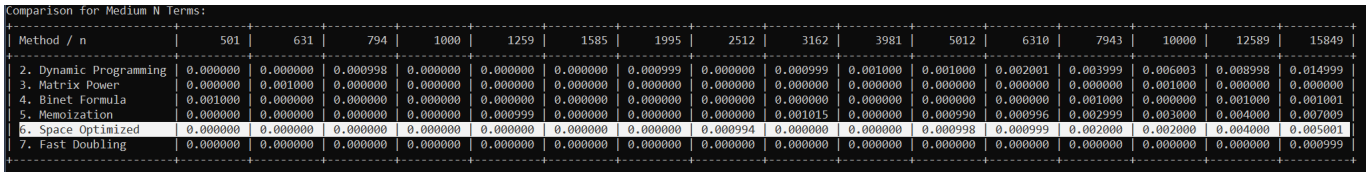
Implementation

```python
# Iterative Space Optimized implementation
def space_optimized(n):
    if n <= 1:
        return n

    # To store current Fibonacci num
    current = 0

    # To store the previous Fibonacci num
    prev1 = 1
    prev2 = 0

    # Loop to compute the Fibonacci from 2 to n
    for i in range(2, n + 1):
        # Compute the current Fibonacci num
        current = prev1 + prev2

        # Update prev num
        prev2 = prev1
        prev1 = current
    return current
```

Listing 6 - Iterative Space Optimized Implementation in Python

Results

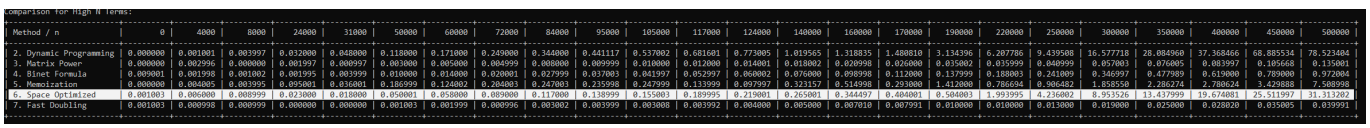
The results of the research regarding the execution of this Iterative Space Optimized method are explained in this section.



Comparison for Medium N Terms:

Method / n	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
2. Dynamic Programming	0.000000	0.000000	0.000998	0.000000	0.000000	0.000000	0.000999	0.000000	0.000999	0.001000	0.001000	0.002001	0.003999	0.006003	0.008998	0.014999
3. Matrix Power	0.000000	0.001000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001000	0.000000	0.000000
4. Binet Formula	0.001000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001000	0.000000	0.001000	0.001001
5. Memoization	0.000000	0.000000	0.000000	0.000000	0.000999	0.000000	0.000000	0.000000	0.001015	0.000000	0.000990	0.000996	0.002999	0.003000	0.004000	0.007009
6. Space Optimized	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000994	0.000000	0.000000	0.000998	0.000999	0.002000	0.002000	0.004000	0.005001
7. Fast Doubling	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000999

Figure 13 - Space Optimized Table Output, medium *n* range



Comparison for High N Terms:

Method / n	0	4000	8000	24000	31000	50000	60000	72000	84000	95000	105000	117000	124000	140000	160000	170000	190000	220000	250000	300000	350000	400000	450000	500000
2. Dynamic Programming	0.000000	0.001001	0.003997	0.032000	0.048000	0.118000	0.171000	0.249000	0.344000	0.441117	0.537002	0.681601	0.773005	1.019565	1.318835	1.480810	3.134396	6.207786	9.439508	16.577718	28.084960	37.368466	68.885534	78.523404
3. Matrix Power	0.000000	0.002996	0.000000	0.001997	0.000997	0.003000	0.005000	0.004999	0.008000	0.009999	0.010000	0.012000	0.014001	0.018002	0.020998	0.026000	0.035002	0.035999	0.040999	0.057003	0.076005	0.083997	0.105668	0.135001
4. Binet Formula	0.009001	0.001998	0.001002	0.001995	0.003999	0.010000	0.014000	0.020001	0.027999	0.037003	0.041997	0.052997	0.060002	0.076000	0.098998	0.112000	0.137999	0.188003	0.241009	0.346997	0.477989	0.619000	0.789000	0.972004
5. Memoization	0.000000	0.004005	0.003995	0.035001	0.036001	0.186999	0.124002	0.204003	0.247003	0.235998	0.247999	0.133999	0.097997	0.321157	0.514998	0.293000	1.412000	0.786694	0.906482	1.858550	2.286274	2.780624	3.429888	7.508998
6. Space Optimized	0.001003	0.006000	0.008999	0.023000	0.018000	0.050001	0.058000	0.089000	0.117000	0.138999	0.155003	0.189995	0.219001	0.265001	0.344497	0.404001	0.504003	1.993995	4.236002	8.953526	13.437999	19.674081	25.511997	31.313202
7. Fast Doubling	0.001003	0.000998	0.000999	0.000000	0.000000	0.001003	0.001999	0.000996	0.003002	0.003999	0.003008	0.003992	0.004000	0.005000	0.007010	0.007991	0.010000	0.010000	0.013000	0.019000	0.025000	0.028020	0.035005	0.039991

Figure 14 - Space Optimized Table Output, high *n* range

As seen in *Figure 13* for the medium n range and in *Figure 14* where we try the method on the high n range.

Figure 15 - Space Optimized Execution Time, medium n range

In *Figure 30* and *Figure 31* we see how Iterative Space Optimized approach behaves compared to the other implementations in a visual method, which makes things more clear.

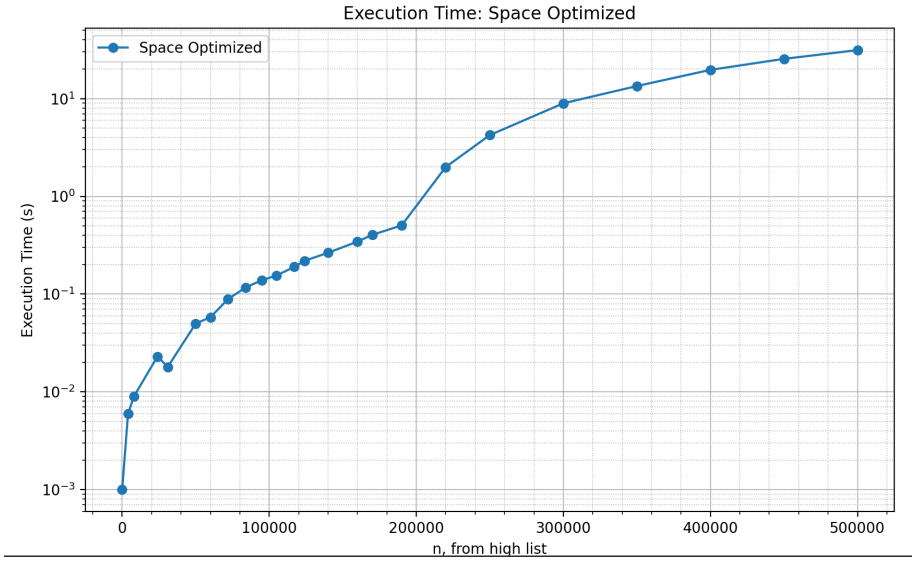


Figure 16 - Space Optimized Execution Time, high n range

The Space Optimized method for Fibonacci calculation has a time complexity of $T(n)$, meaning its execution time increases as the input size n grows, though at a slower rate compared to other methods. Despite this, it maintains $O(1)$ space complexity, meaning it requires constant memory regardless of the input size. This makes it a highly efficient option in terms of memory usage while still being able to handle

larger values of n.

Memoization Recursive Method

In the traditional recursive Fibonacci algorithm, we repeatedly compute the same Fibonacci numbers multiple times, leading to unnecessary recomputation. This happens because each Fibonacci number depends on two previous numbers, and without remembering previously computed values, we end up recalculating them repeatedly.

To optimize this, we can store already computed Fibonacci numbers in a dictionary (or an array) and reuse them whenever needed. This technique, known as memoization, ensures that each Fibonacci number is computed only once and then stored for future reference.

Instead of making redundant recursive calls, we first check whether the required Fibonacci number is already stored. If it is, we return it immediately. If not, we compute it, store the result, and then return it. This way, we efficiently build up solutions to smaller subproblems and reuse them instead of recalculating them.

Algorithm Description

Approach:

```
memo <- {0: 0, 1: 1}
Fibonacci_Memo(n):
    if n is in memo:
        return memo[n]

    # Compute and store the Fibonacci number
    memo[n] <- Fibonacci_Memo(n - 1) + Fibonacci_Memo(n - 2)

    return memo[n]
```

Listing 7 - Pseudocode Memoization Recursive Implementation

Implementation

```
# Memo for memoization implementation
memo = {0: 0, 1: 1}
# Memoization implementation
def memo_recursive(n):
    if n in memo:
        return memo[n]
    # Compute and memo the Fibonacci num
    memo[n] = memo_recursive(n - 1) + memo_recursive(n - 2)
    return memo[n]
```

Listing 8 - Memoization Recursive in Python

Results

The results of the research regarding the execution of the Memoization Recursive method are explained in this section.

Comparison for Medium N Terms:

Method / n	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
2. Dynamic Programming	0.000000	0.000000	0.000998	0.000000	0.000000	0.000000	0.000999	0.000000	0.000999	0.001000	0.001000	0.002001	0.003999	0.006003	0.008998	0.014999
3. Matrix Power	0.000000	0.001000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001000	0.000000	0.000000
4. Binet Formula	0.001000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001000	0.000000	0.001000	0.001001
5. Memoization	0.000000	0.000000	0.000000	0.000000	0.000999	0.000000	0.000000	0.000000	0.001015	0.000000	0.000990	0.000996	0.002999	0.003000	0.004000	0.007009
6. Space Optimized	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000994	0.000000	0.000000	0.000998	0.000999	0.002000	0.002000	0.004000	0.005001
7. Fast Doubling	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000999

Figure 17 - Memoization Table Output, medium *n* range

Comparison for High N Terms:

Method / n	0	4000	8000	24000	31000	50000	60000	72000	84000	95000	105000	117000	124000	140000	160000	170000	190000	220000	250000	300000	350000	400000	450000	500000
2. Dynamic Programming	0.000000	0.001001	0.003997	0.032000	0.048000	0.118000	0.171000	0.249000	0.344000	0.441117	0.557002	0.681601	0.773005	1.019565	1.318835	1.488010	3.134396	6.207786	9.439508	16.577718	28.804960	37.368466	68.885534	78.523404
3. Matrix Power	0.000000	0.002996	0.000000	0.001997	0.000997	0.003000	0.005000	0.004999	0.008000	0.009999	0.010000	0.012000	0.014001	0.018002	0.020998	0.026000	0.035002	0.035999	0.040999	0.057003	0.076005	0.083997	0.105668	0.135001
4. Binet Formula	0.009001	0.001998	0.001002	0.001995	0.003999	0.010000	0.014000	0.020001	0.027999	0.037003	0.041997	0.052997	0.060002	0.076000	0.098998	0.112000	0.137999	0.188003	0.241009	0.346997	0.477989	0.619000	0.789000	0.972004
5. Memoization	0.000000	0.004005	0.003995	0.095001	0.036001	0.186999	0.124002	0.204003	0.247003	0.235998	0.247999	0.133999	0.097997	0.323157	0.514998	0.293000	1.412000	0.786694	0.906482	1.858550	2.286274	2.780624	3.429888	7.508998
6. Space Optimized	0.001003	0.000000	0.000999	0.023000	0.010000	0.050001	0.050000	0.070000	0.117000	0.138999	0.151003	0.189995	0.210001	0.265001	0.344577	0.404001	0.584003	1.003995	4.236002	8.953526	13.437999	19.674001	25.511997	31.313202
7. Fast Doubling	0.001003	0.000998	0.000999	0.000000	0.000000	0.001003	0.001999	0.000996	0.003002	0.003999	0.003008	0.003992	0.004000	0.005000	0.007010	0.007991	0.010000	0.010000	0.013000	0.019000	0.025000	0.028020	0.035005	0.039991

Figure 18 - Memoization Table Output, high *n* range

Figure 19 - Memoization Execution Time, medium *n* range

In *Figure 30* and *Figure 31* we see how the Memoization Recursive approach behaves compared to the other implementations in a visual method, which makes things more clear, as this simple change to the Recursive method makes a big change in the time execution complexity, as plotted in the graphs from *Figure 19* and *Figure 20*.

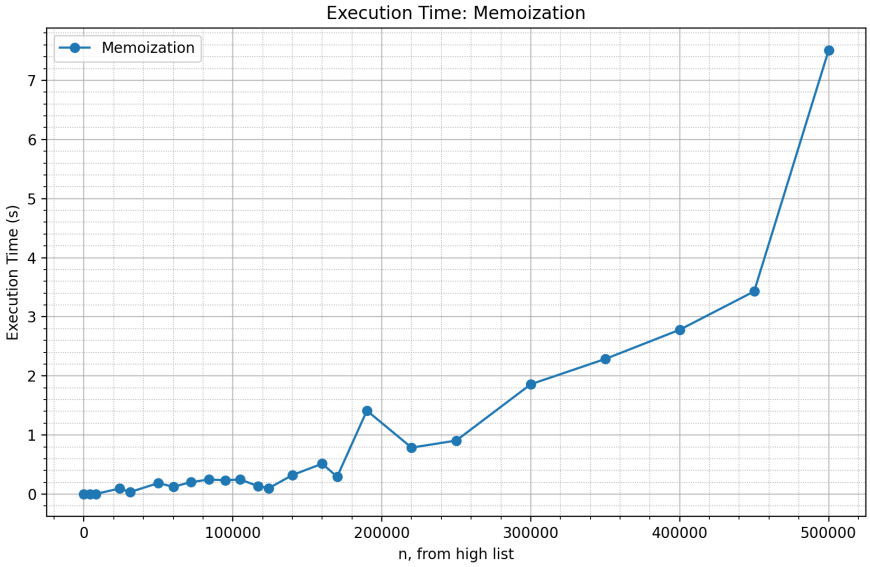


Figure 20 - Memoization Execution Time, high n range

The Memoization method optimizes Fibonacci computation by storing previously computed values, reducing redundant calculations and achieving $T(n)$ time complexity. However, it has a high space complexity of $O(n)$ due to storing all computed values in a memo table, making it inefficient for very large n. While significantly faster than naive recursion, it lags behind space-efficient methods like Fast Doubling for extremely large inputs due to memory overhead.

Binet Formula Method

Another approach to finding the n-th Fibonacci number involves using the golden ratio, as Fibonacci numbers closely follow this ratio as they progress toward infinity.

The Binet Formula Method is an alternative technique for computing the n-th Fibonacci number by leveraging the golden ratio (ϕ). However, since this method relies on decimal calculations, rounding errors in Python start to accumulate over time. These errors become noticeable around the 70th Fibonacci number, making the method impractical for larger values, despite its computational efficiency.

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))
```

Listing 9 - Pseudocode Binet Formula Implementation

Implementation

```
# Binet Formula implementation
def binet_formula(n):
    ctx = Context(prec=60, rounding=ROUND_HALF_EVEN)
    phi = Decimal((1 + Decimal(5**(1/2))))
    psi = Decimal((1 - Decimal(5**(1/2))))
    return int((ctx.power(phi, Decimal(n)) - ctx.power(psi, Decimal(n))) / (2 **
        n * Decimal(5 ** (1/2))))
```

Listing 10 - Binet Formula in Python

Results

The results of the research regarding the execution of the Binet Formula method are explained in this section.

Comparison for Medium N Terms:

Method / n	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
2. Dynamic Programming	0.000000	0.000000	0.000998	0.000000	0.000000	0.000000	0.000999	0.000000	0.000999	0.001000	0.001000	0.002001	0.003999	0.006003	0.008998	0.014999
3. Matrix Power	0.000000	0.001000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001000	0.000000	0.000000
4. Binet Formula	0.001000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001000	0.000000	0.001000	0.001001
5. Memoization	0.000000	0.000000	0.000000	0.000000	0.000999	0.000000	0.000000	0.000000	0.001015	0.000000	0.000990	0.000996	0.002999	0.003000	0.004000	0.007009
6. Space Optimized	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000994	0.000000	0.000000	0.000998	0.000999	0.002000	0.002000	0.004000	0.005001
7. Fast Doubling	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000999

Figure 21 - Binet Formula Table Output, medium *n* range

Comparison for High N Terms:

Method / n	0	4000	8000	24000	31000	50000	60000	72000	84000	95000	105000	117000	124000	140000	160000	170000	190000	220000	250000	300000	350000	400000	450000	500000
2. Dynamic Programming	0.000000	0.001001	0.003997	0.032000	0.048000	0.118000	0.171000	0.249000	0.344000	0.441117	0.537002	0.681601	0.773005	1.019565	1.318835	1.480810	3.134396	6.207786	9.439508	16.577718	28.084960	37.368466	68.885534	78.523404
3. Matrix Power	0.000000	0.002996	0.000000	0.001997	0.000997	0.003000	0.005000	0.004999	0.008000	0.009999	0.010000	0.012000	0.014001	0.018002	0.020998	0.026000	0.035002	0.035999	0.040999	0.057003	0.076005	0.083997	0.105668	0.135001
4. Binet Formula	0.000001	0.001998	0.001002	0.001995	0.003999	0.010000	0.014000	0.020001	0.027999	0.037003	0.041997	0.052997	0.060002	0.076000	0.098998	0.112000	0.137999	0.188003	0.241009	0.346997	0.477985	0.619000	0.789000	0.972001
5. Memoization	0.000000	0.004005	0.003995	0.095001	0.036001	0.186999	0.124002	0.204003	0.247003	0.235998	0.247999	0.133999	0.097997	0.323157	0.514998	0.293000	1.412000	0.786694	0.906482	1.858550	2.286274	2.780624	3.429888	7.508998
6. Space Optimized	0.001003	0.006000	0.008999	0.023000	0.018000	0.050001	0.058000	0.089000	0.117000	0.138999	0.155003	0.189995	0.219001	0.265001	0.344497	0.404001	0.504003	1.993995	4.236002	8.953526	13.437999	19.674081	25.511997	31.313202
7. Fast Doubling	0.001003	0.000998	0.000999	0.000000	0.000000	0.001003	0.001999	0.000996	0.003002	0.003999	0.003008	0.003992	0.004000	0.005000	0.007010	0.007991	0.010000	0.010000	0.013000	0.019000	0.025000	0.028020	0.035005	0.039991

Figure 22 - Binet Formula Table Output, high *n* range

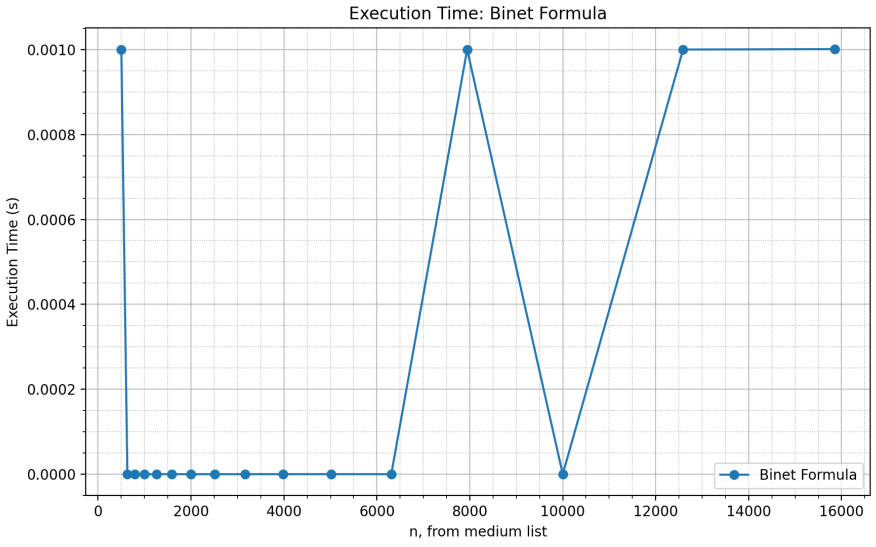


Figure 23 - Binet Formula Execution Time, medium *n* range

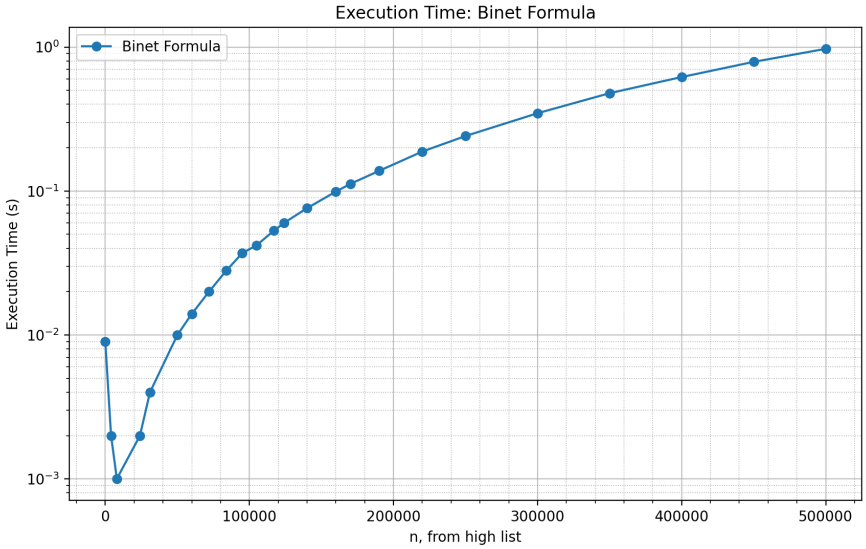


Figure 24 - Binet Formula Execution Time, high *n* range

```
Binet Formula: 53583592549909879676339 82
Fast Doubling: 53583592549909666408718 40
Error: 21326762142
```

Figure 25 - Proof that Binet Formula is not accurate enough

The Binet formula, while offering an impressive logarithmic time complexity $O(logn)$ and minimal space requirements, suffers from a critical flaw in accuracy due to floating-point precision limitations when calculating large Fibonacci numbers, as evidenced by the error shown in *Figure 25*. The Fast Doubling method and Matrix Power approach provide much better alternatives, maintaining the same $O(logn)$ time complexity while delivering exact results without the precision issues. This makes them significantly more reliable for practical applications, with Fast Doubling being particularly efficient as shown in both the execution time comparisons *Figure 22* and the accuracy comparison *Figure 25*.

Fast Doubling Method

The Fast Doubling algorithm is a clever method for calculating Fibonacci numbers by utilizing relationships between $F(k)$, $F(k+1)$, $F(2k)$, and $F(2k+1)$. Instead of computing numbers sequentially, it allows us to jump ahead by doubling the index each time, effectively calculating $F(2k)$ and $F(2k+1)$ from $F(k)$ and $F(k+1)$. This approach stems from the mathematical properties inherent in the matrix form of Fibonacci calculations but eliminates redundant computations. The method gets its name from the way it doubles the index in each step, making it particularly efficient for calculating large Fibonacci numbers.

Algorithm Description

Algorithm proof, consists of: We will assume the fact that the matrix exponentiation method is correct for all $n >= 1$

$$\begin{bmatrix} F(2n+1) & F(2n) \\ F(2n) & F(2n-1) \end{bmatrix} = \left(\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \right)^2$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{2n} \qquad = \begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix}^2$$

$$= \begin{bmatrix} F(n+1)^2 + F(n)^2 & F(n+1)F(n) + F(n)F(n-1) \\ F(n)F(n+1) + F(n-1)F(n) & F(n)^2 + F(n-1)^2 \end{bmatrix}.$$

Therefore, by equating cells in the top matrix to cells in the bottom matrix: $F(2n+1) = F(n+1)^2 + F(n)^2$.

$$\begin{aligned} F(2n) &= F(n)\left[F(n+1) + F(n-1)\right] \\ &= F(n)\left[F(n+1) + (F(n+1) - F(n))\right] \\ &= F(n)\left[2F(n+1) - F(n)\right]. \\ F(2n-1) &= F(n)^2 + F(n-1)^2. \end{aligned}$$

Implementation

```
# The main method for Fast Doubling implementation
def fast_doubling(n):
    return _fib(n)[0]
# Auxiliary method for Fast Doubling
def _fib(n):
    if n == 0:
        return 0, 1
    else:
        a, b = _fib(n // 2)
        c = a * (b * 2 - a)
        d = a * a + b * b
        if n % 2 == 0:
            return c, d
        else:
            return d, c + d
```

Listing 11 - Fast Doubling in Python

Results

The results of the research regarding the execution of the Fast Doubling method are explained in this section.

Comparison for Medium N Terms:

Method / n	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
2. Dynamic Programming	0.000000	0.000000	0.000998	0.000000	0.000000	0.000000	0.000999	0.000000	0.000999	0.001000	0.001000	0.002001	0.003999	0.006003	0.008998	0.014999
3. Matrix Power	0.000000	0.001000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001000	0.000000	0.000000
4. Binet Formula	0.001000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001000	0.000000	0.001000	0.001001
5. Memoization	0.000000	0.000000	0.000000	0.000000	0.000000	0.000999	0.000000	0.000000	0.000000	0.001015	0.000000	0.000990	0.000996	0.002999	0.003000	0.004000	0.007009
6. Space Optimized	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000994	0.000000	0.000000	0.000998	0.000999	0.002000	0.002000	0.004000	0.005001
7. Fast Doubling	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000999

Figure 26 - Fast Doubling Table Output, medium *n* range

Comparison for High N Terms:

Method / n	0	4000	8000	24000	31000	50000	60000	72000	84000	95000	105000	117000	124000	140000	160000	170000	190000	220000	250000	300000	350000	400000	450000	500000
2. Dynamic Programming	0.000000	0.001001	0.003997	0.032000	0.048000	0.118000	0.171000	0.249000	0.344000	0.441117	0.537002	0.681601	0.773005	1.019565	1.318835	1.480810	3.134396	6.207786	9.439508	16.577718	28.804960	37.368466	68.885534	78.523404
3. Matrix Power	0.000000	0.002996	0.000000	0.001997	0.000997	0.003000	0.005000	0.004999	0.008000	0.009999	0.010000	0.012000	0.014001	0.018002	0.020998	0.026000	0.035002	0.035999	0.040999	0.057003	0.076005	0.083997	0.105668	0.135001
4. Binet Formula	0.009001	0.001998	0.001002	0.001995	0.003999	0.010000	0.014000	0.020001	0.027999	0.037003	0.041997	0.052997	0.060002	0.076000	0.098998	0.112000	0.137999	0.188003	0.241009	0.346997	0.477989	0.619000	0.789000	0.972004
5. Memoization	0.000000	0.004005	0.003995	0.095001	0.036001	0.186999	0.124002	0.204003	0.247003	0.235998	0.247999	0.133999	0.097997	0.323157	0.514998	0.293000	1.412000	0.786694	0.906482	1.858558	2.286274	2.780624	3.429888	7.508998
6. Space Optimized	0.001003	0.006000	0.008999	0.023000	0.018000	0.050001	0.058000	0.089000	0.117000	0.138999	0.155003	0.189995	0.219001	0.265001	0.344497	0.404001	0.504003	1.993995	4.236002	8.953526	13.437999	19.674081	25.511997	31.313202
7. Fast Doubling	0.001003	0.000998	0.000999	0.000000	0.000000	0.001003	0.001999	0.000996	0.003002	0.003999	0.003008	0.003992	0.004000	0.005000	0.007010	0.007991	0.010000	0.010000	0.013000	0.019000	0.025000	0.028020	0.035005	0.039991

Figure 27 - Fast Doubling Table Output, high *n* range

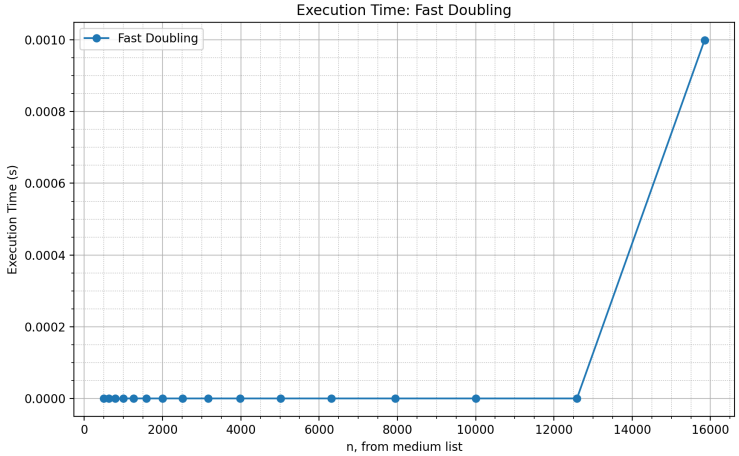


Figure 28 - Fast Doubling Execution Time, medium *n* range

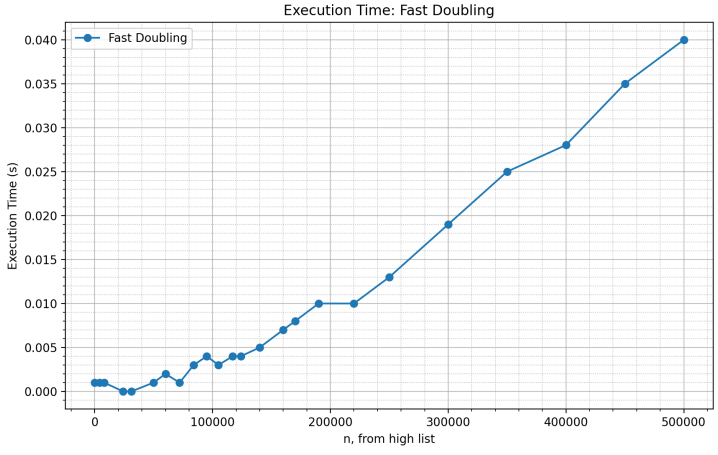


Figure 29 - Fast Doubling Execution Time, high *n* range

The Fast Doubling method for computing Fibonacci numbers demonstrates exceptional efficiency, maintaining consistently low execution times, as seen in *Figure 27*, even for very large values of *n*, as seen in both the table and graphs. Its execution time grows very slowly compared to other methods, making it the most scalable option for high *n* terms. As seen in *Figure 29*, focusing on Fast Doubling alone, shows a steady but minimal increase in computation time, reinforcing its near-logarithmic complexity. The time execution complexity of Fast Doubling is $T(logn)$, making it one of the best methods for computing large Fibonacci numbers efficiently.

All Graphs Overview

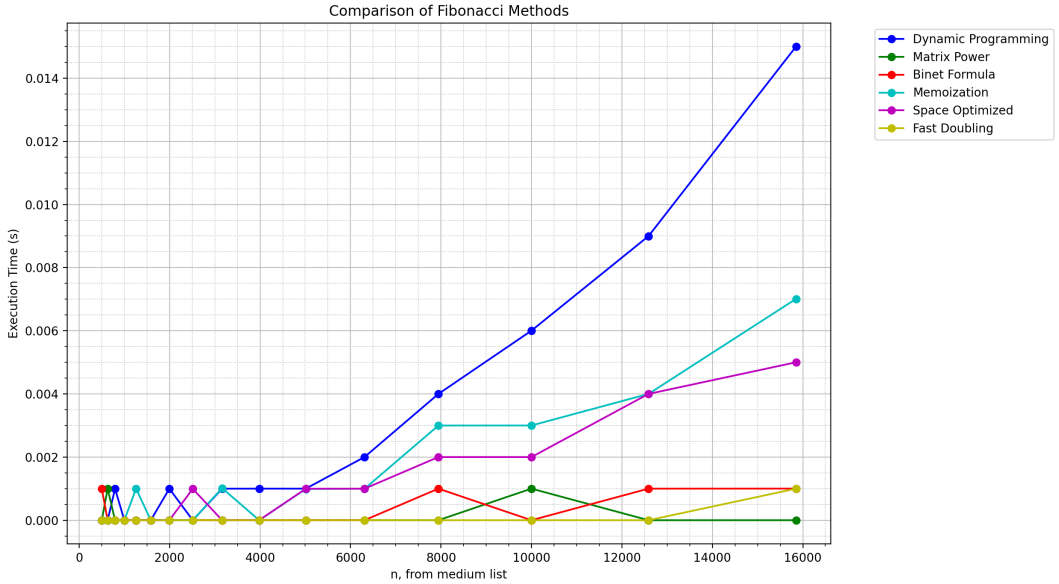


Figure 30 - Execution Time All Methods, Medium *n* list

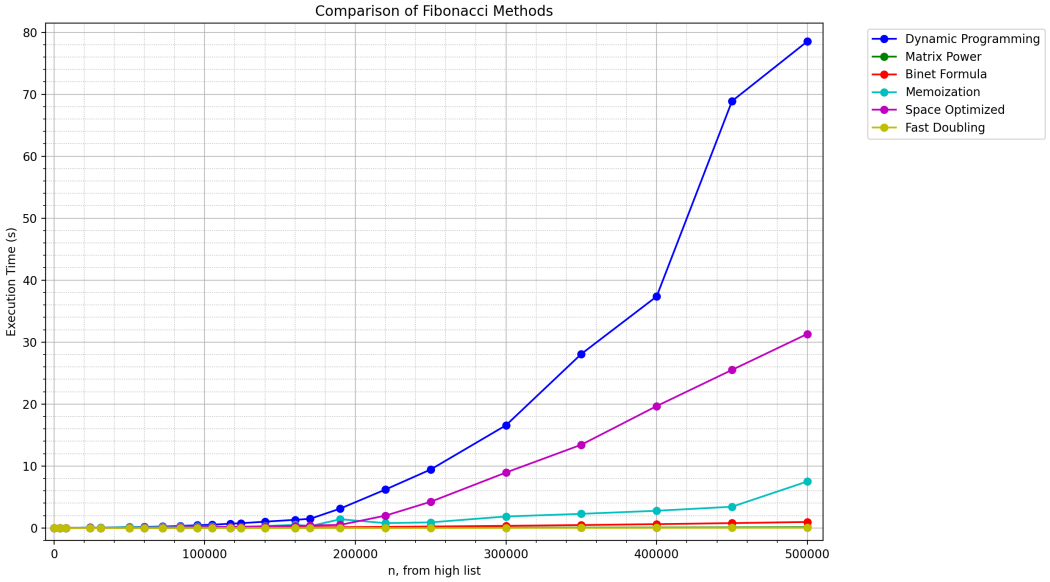


Figure 31 - Execution Time All Methods, High *n* list

CONCLUSION

In this laboratory work, I have implemented seven distinct algorithms for determining the Fibonacci n-th term, beside the implementation I have also empirically analyzed the approaches: *Recursive*, *Dynamic Programming (DP)*, *Iterative Space Optimized*, *Memoization Recursive*, *Binet Formula*, *Matrix Power* (via exponentiation by squaring), and *Fast Doubling*. The main objective, which required to evaluate their efficiency using execution time as the primary metric, with inputs ranging from small $(n <= 45)$ to medium $(n <= 15849)$, or even to large $(n <= 5000000)$.

Each algorithm demonstrated unique characteristics in time and space complexity, highlighting trade-offs between computational speed, memory usage, and accuracy.

For instance, we have the Recursive Method, which is a naive approach, while elegant, suffers from exponential time complexity ($T(2^n)$) due to redundant recalculations. Execution time becomes impractical beyond $n = 30$, reaching 308 seconds at $n = 45$, rendering it unsuitable for large inputs.

In continuity we have the Dynamic Programming method, characterized by storing intermediate results iteratively, DP achieves linear time ($T(n)$) but requires linear space. It is significantly faster than recursion but becomes memory-intensive for extremely large n.

We also have the Iterative Space Optimized approach that keeping in mind the DP's linear time ($T(n)$), this method reduces space complexity to O(1) by tracking only the last two Fibonacci numbers. It balances efficiency and memory, making it practical for moderately large n.

Furthermore, the Memoization Recursive, which is the recursive variant that caches results, achieves linear time ($T(n)$) but with $O(n)$ space overhead. While faster than naive recursion, it lags behind iterative methods due to function call overhead and memory constraints.

In addition, I have also implemented the Binet Formul, by leverating the golden ratio, this method operates in logarithmic time ($T(logn)$) but introduces floating-point inaccuracies beyond $n = 70$. Though fast and space-efficient, its approximation errors make it unreliable for precise calculations.

Another analyzed algorithm was Matrix Power (Using Exponentiation by Squaring) that works by decomposing matrix multiplication recursively, this method achieves logarithmic time ($T(logn)$) and constant space ($O(1)$). It scales efficiently for large n and delivers exact results, outperforming linear-time methods.

And last but not least, Fast Doubling, the fastest algorithm, combining matrix properties with divide-and-conquer, also runs in logarithmic time ($T(logn)$) with $O(1)$ space. Its efficiency comes from computing pairs of Fibonacci numbers simultaneously, making it ideal for extremely large numbers n.

So, for accurate and scalable solutions, Fast Doubling and Matrix Power are optimal, with logarith-

mic time complexity and minimal memory usage. Combining Fast Doubling with Karatsuba multiplication could further enhance its speed. Linear-time algorithms like DP, Iterative, and Memoization are acceptable for moderate n but lack scalability. The Binet Formula, while fast, is limited by precision issues. The Recursive method, with its exponential growth, is entirely impractical beyond small inputs. This whole analysis during the laboratory underscores the importance of selecting algorithms based on input size, resource constraints, and precision requirements.

Bibliography

[1] Project Nayuki. ”Fast Fibonacci algorithms”. https://www.nayuki.io/page/fast-fibonacci-algorithms.

[2] Geeksforgeeks. ”Nth Fibonacci Number Algorithms”. https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/.

Appendix

[3] GitHub Repo. https://github.com/MaxNoragami/AA-labs-2025