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

Laboratory Work 1

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

Elaborated:
std. gr. FAF-231 Alexandru Rudoi

Verified:
asist. univ. Cristofor Fiştic

Chişinău – 2025

Contents

1 Algorithm Analysis

1.1 Objective

The main objective of this laboratory work is to analyze and compare different Fibonacci computation methods in terms of *execution time, efficiency, and scalability.* By implementing and benchmarking multiple approaches, we aim to:

- Understand the impact of algorithmic complexity on performance.
- Identify the most efficient method for computing Fibonacci numbers at various input sizes.
- Visualize performance trends through execution time measurements.
- Reinforce the importance of choosing the right algorithm for a given problem.

1.2 Tasks

To achieve these objectives, the following tasks were completed:

1. Implemented multiple Fibonacci computation methods, ranging from naive recursion to optimized logarithmic-time algorithms.
2. Measured and recorded execution times for different values of n.
3. Plotted execution time graphs to compare algorithm performance.
4. Analyzed complexity and trade-offs between speed, memory usage, and precision.

1.3 Theoretical Notes

The Fibonacci sequence is defined mathematically as:

$$F(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F(n-1) + F(n-2), & n \geq 2 \end{cases}$$

where each number is the sum of the two preceding numbers.

Different algorithms can be used to compute Fibonacci numbers, each with varying time complexity:

- **Naive Recursion:** $O(2^n)$ – Exponential time complexity due to redundant computations.
- **Memoization (Top-Down DP):** $O(n)$ – Uses caching to improve recursion efficiency.

- **Bottom-Up DP:** $O(n)$ – Iterative approach storing all computed values.
- **Space-Optimized DP:** $O(n)$ – Reduces memory usage while maintaining the same time complexity.
- **Matrix Exponentiation:** $O(\log n)$ – Uses matrix power computation to achieve logarithmic time.
- **Fast Doubling:** $O(\log n)$ – Efficient method using properties of Fibonacci numbers.
- **Binet's Formula:** $O(1)$ – Direct formula but suffers from floating-point precision errors for large n.

1.4 Introduction

Fibonacci numbers are widely used in *mathematics, computer science, and real-world applications*, such as:

- Algorithmic problem-solving and dynamic programming.
- Cryptographic sequences and random number generation.
- Mathematical modeling and financial forecasting.

Computing Fibonacci numbers efficiently is crucial for optimizing performance in large-scale applications. This laboratory work explores different algorithms to determine the *most optimal* method based on execution time and complexity.

1.5 Comparison Metric

To compare the performance of Fibonacci methods, the following metric was used:

Execution Time (ms): The time taken to compute $F(n)$ was measured using Python's `time.perf_counter()` function, ensuring accurate timing.

Each method was tested for multiple values of n, and a performance graph was generated to visualize the execution trends.

1.6 Input Format

Each Fibonacci algorithm receives a single integer n as input:

- $0 \leq n \leq 50,000$ for iterative and logarithmic methods.
- $0 \leq n \leq 70$ for Binet's Formula (due to floating-point limitations).
- $0 \leq n \leq 30$ for Recursive Fibonacci (due to exponential complexity).

The output is the n-th Fibonacci number computed using the respective method.

2 IMPLEMENTATION

2.1 Binet's Formula Method

2.1.1 Algorithm Explanation

Binet's Formula is a mathematical expression that provides a *direct computation* of the Fibonacci sequence without requiring recursion or iteration. The formula is defined as:

$$F(n) = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}}$$

where: - $\varphi = \frac{1+\sqrt{5}}{2}$ (the Golden Ratio) - $\sqrt{5}$ is the square root of 5

Time and Space Complexity Since this formula only requires a *constant number of arithmetic operations*, it runs in $O(1)$ *time complexity* and $O(1)$ *space complexity*. However, *floating-point precision errors* make it *unreliable for* $n > 70$.

2.1.2 Python Implementation

```python
import decimal

def nth_fibonacci_binet(n):
    if n > 70:
        return None  # Prevent floating-point precision errors
    decimal.getcontext().prec = 100  # Increase precision
    phi = decimal.Decimal(1 + decimal.Decimal(5).sqrt()) / 2
    return round(phi**n / decimal.Decimal(5).sqrt())
```

2.1.3 Results Analysis

Execution Time: Since Binet's Formula is an $O(1)$ *operation*, the execution time remains *extremely low* for all tested values of n, as shown in *Figure 1*.

```
Execution Times for Binet's Formula (O(1)):

   1      5      10     15     20     30     40     50     60     70
0.056 0.0216 0.0168 0.017 0.0157 0.0229 0.0154 0.0164 0.016 0.0172
```

Figure 1: Binet's Formula Execution Time

Visualization: The performance graph in *Figure 2* confirms that execution time remains almost constant across different values of n, validating its $O(1)$ complexity.

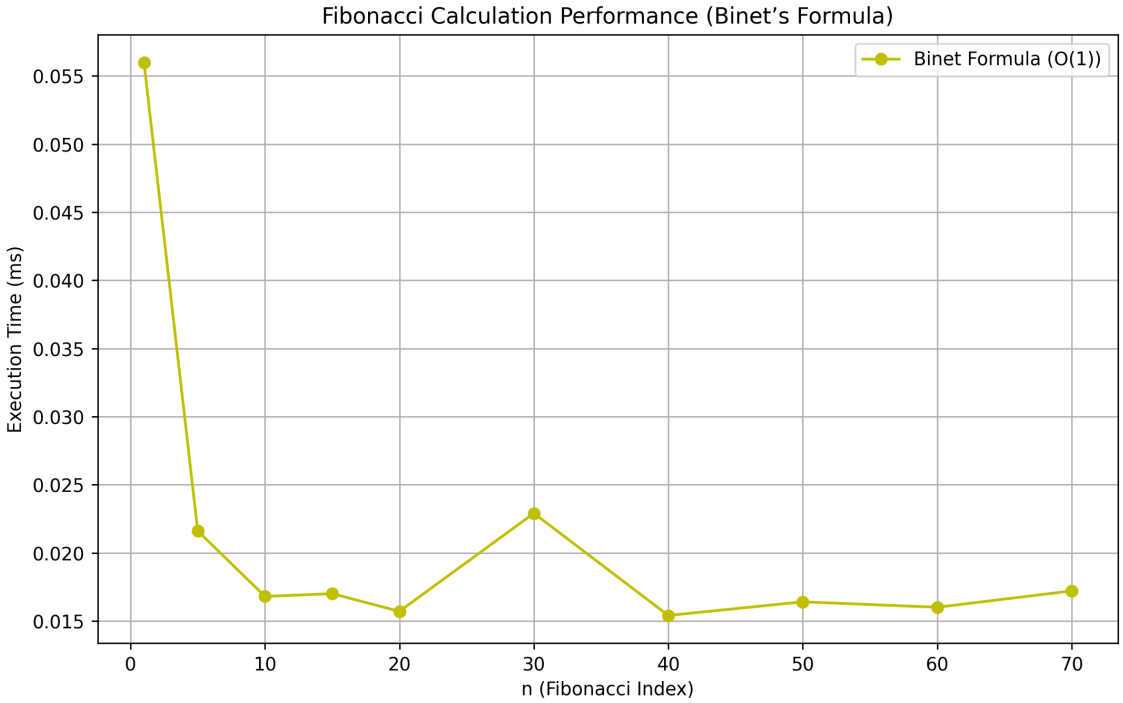

Figure 2: Binet's Formula Execution Time Graph

2.2 Bottom-Up Dynamic Programming Method

2.2.1 Algorithm Explanation

The Bottom-Up Dynamic Programming (DP) method computes Fibonacci numbers by *building a table* from the base cases up to the desired n. This eliminates redundant calculations and avoids recursion. The algorithm follows these steps:

1. Initialize an array fib where fib[0] = 0 and fib[1] = 1.
2. Iterate from $i = 2$ to n, storing previously computed Fibonacci values in the array.
3. Return fib[n] after the loop completes.

The approach *eliminates recursion* and *reduces time complexity* to $O(n)$ at the cost of storing $O(n)$ values.

Time and Space Complexity

- **Time Complexity:** $O(n)$ - The algorithm computes each Fibonacci number once.
- **Space Complexity:** $O(n)$ - Stores results in an array of size n.

2.2.2 Python Implementation

```python
def nth_fibonacci_bottom_up(n):
    if n <= 1:
        return n
    fib = [0, 1] + [0] * (n - 1)
    for i in range(2, n + 1):
        fib[i] = fib[i - 1] + fib[i - 2]
    return fib[n]
```

2.2.3 Results Analysis

Execution Time: Since the Bottom-Up DP method runs in $O(n)$ *time*, execution time increases *linearly* with n. The measured execution times are shown in *Figure 3*.

```
Execution Times for Bottom-Up DP (O(n)):

   1     20     40     70     100    130    200    250    400    500
0.0004 0.005 0.0056 0.0048 0.0342 0.0081 0.0101 0.0145 0.0286 0.0364
```

Figure 3: Bottom-Up DP Execution Time

Visualization: The performance graph in *Figure 4* shows the linear increase in execution time as n grows, confirming its $O(n)$ complexity.

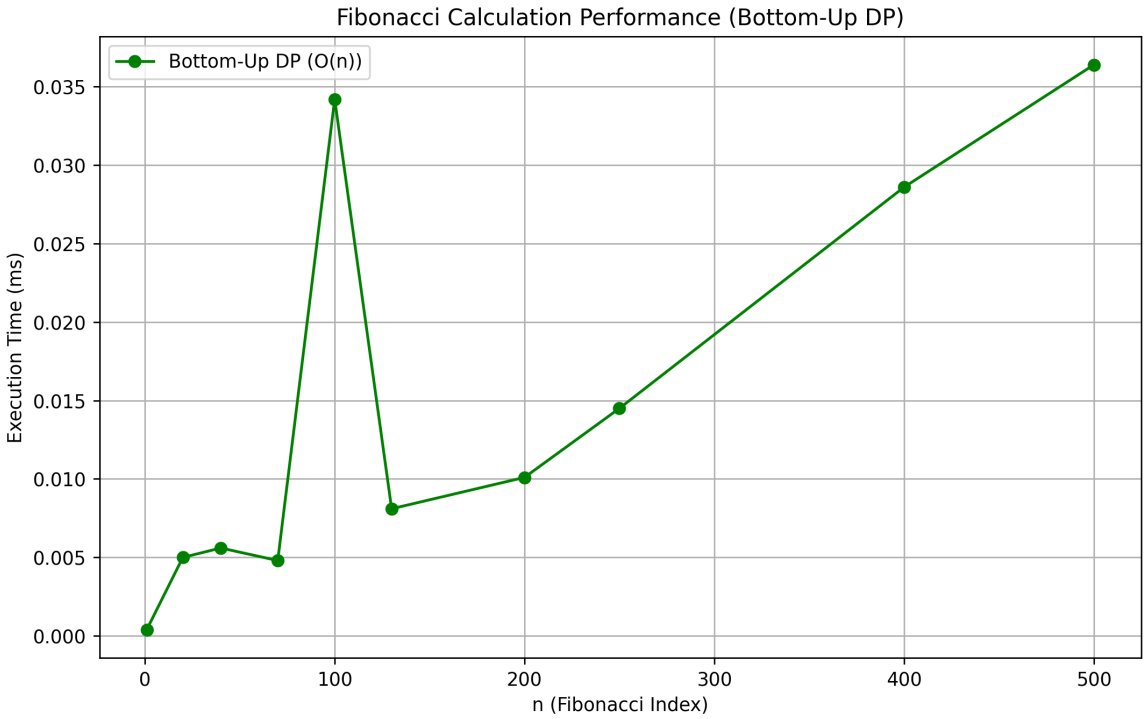

Figure 4: Bottom-Up DP Execution Time Graph

2.3 Fast Doubling Method

2.3.1 Algorithm Explanation

The Fast Doubling method calculates Fibonacci numbers based on *matrix properties* of the Fibonacci sequence, allowing it to compute $F(n)$ in $O(\log n)$ *time*. The recurrence relations used are:

$$F(2k) = F(k) \times [2F(k+1) - F(k)]$$

$$F(2k+1) = F(k+1)^2 + F(k)^2$$

This method *reduces the number of recursive calls* by halving n in each step.

Time and Space Complexity

- **Time Complexity:** $O(\log n)$ - Logarithmic time complexity.
- **Space Complexity:** $O(\log n)$ - Due to recursive depth.

2.3.2 Python Implementation

```python
def nth_fibonacci_fast_doubling(n):
    if n == 0:
        return (0, 1)
    else:
        a, b = nth_fibonacci_fast_doubling(n // 2)
        c = a * ((b << 1) - a)
        d = a * a + b * b
        return (c, d) if n % 2 == 0 else (d, c + d)

def nth_fibonacci_fast(n):
    return nth_fibonacci_fast_doubling(n)[0]
```

2.3.3 Results Analysis

Execution Time: The execution time for the Fast Doubling method grows *logarithmically* with n, as seen in *Figure 5*.

```
Execution Times for Fast Doubling (O(log n)):

 1      20     40     70     100    200    500    1000   5000   10000  50000
0.0028  0.003  0.002 0.0023 0.0027 0.0026  0.007 0.0038 0.0143 0.0439  0.414
```

Figure 5: Fast Doubling Execution Time

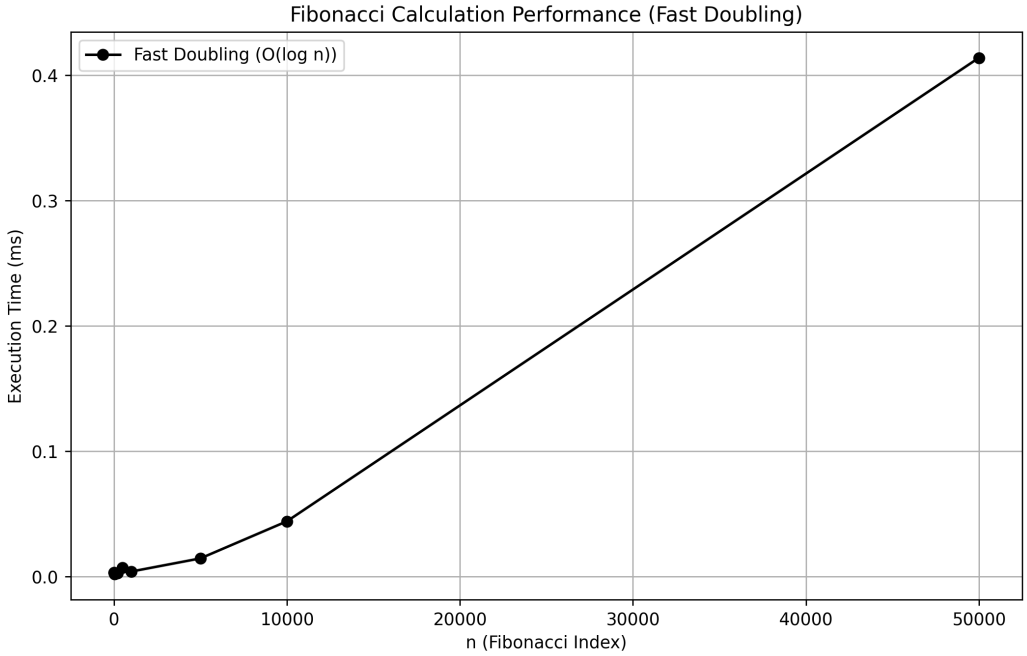

Figure 6: Fast Doubling Execution Time Graph

Visualization: Figure 6 confirms the efficiency of the Fast Doubling approach, as it grows significantly slower than linear methods.

2.4 Matrix Exponentiation Method

2.4.1 Algorithm Explanation

Matrix Exponentiation optimizes Fibonacci computation using the *transformation matrix*:

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

By exponentiating this matrix in $O(\log n)$ *time* using fast exponentiation, we compute Fibonacci numbers efficiently.

Time and Space Complexity

- **Time Complexity:** $O(\log n)$ - Fast exponentiation.
- **Space Complexity:** $O(1)$ - Uses only a few variables.

2.4.2 Python Implementation

```
def nth_fibonacci_matrix(n):
    def multiply(A, B):
        return [[A[0][0] * B[0][0] + A[0][1] * B[1][0],
```

```
                A[0][0] * B[0][1] + A[0][1] * B[1][1]],
               [A[1][0] * B[0][0] + A[1][1] * B[1][0],
                A[1][0] * B[0][1] + A[1][1] * B[1][1]]]

    def power(F, n):
        if n == 0 or n == 1:
            return F
        M = [[1, 1], [1, 0]]
        if n % 2 == 0:
            return power(multiply(F, F), n // 2)
        else:
            return multiply(F, power(multiply(F, F), (n - 1) // 2))

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

2.4.3 Results Analysis

Execution Time: *Figure 7* confirms the logarithmic growth of execution time.

```
Execution Times using Matrix Exponentiation (O(log n))
 1      20     40     70     100    200    300    500    1000   5000   10000
0.0023 0.0098 0.0089 0.0072 0.0068 0.0088 0.0121 0.0122 0.0184 0.0575 0.1735
```

Figure 7: Matrix Exponentiation Execution Time

Visualization: Figure 8 shows how execution time scales efficiently.

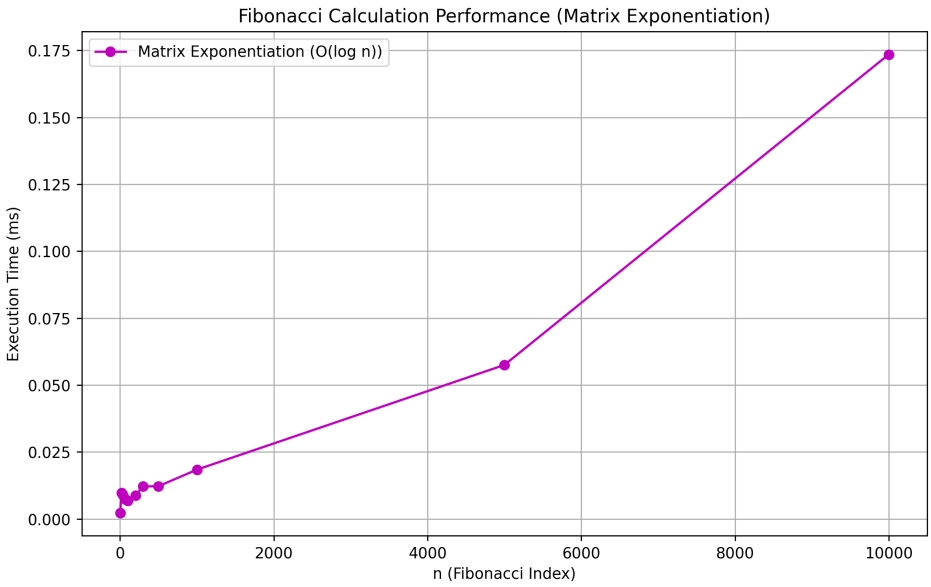

Figure 8: Matrix Exponentiation Execution Time Graph

2.5 Memoization Method

2.5.1 Algorithm Explanation

Memoization improves Fibonacci computation by *caching previously computed values*, preventing redundant calculations. The approach follows:

1. Store computed Fibonacci values in an array.
2. Reuse stored results instead of recomputing them.

Time and Space Complexity

- **Time Complexity:** $O(n)$ - Each Fibonacci number is computed once.
- **Space Complexity:** $O(n)$ - Stores results in an array.

2.5.2 Python Implementation

```
def nth_fibonacci_util(n, memo):
    if n <= 1:
        return n
    if memo[n] != -1:
        return memo[n]
    memo[n] = nth_fibonacci_util(n - 1, memo) + nth_fibonacci_util(n -
       2, memo)
    return memo[n]

def nth_fibonacci_memoization(n):
    memo = [-1] * (n + 1)
    return nth_fibonacci_util(n, memo)
```

2.5.3 Results Analysis

The Memoization method significantly improves performance compared to naive recursion by *storing previously computed Fibonacci values*, reducing redundant calculations. *Figure 9* provides an overview of the measured execution times for different values of n, while *Figure 10* visualizes the increasing trend in execution time, confirming its $O(n)$ *complexity*.

```
Execution Times using Memoization (O(n))
   1      20     40   70     100    130    200   250    400    500
0.0017 0.0056 0.0089 0.01 0.0146 0.0383 0.0502 0.049 0.1148 0.1251
```

Figure 9: Memoization Execution Time for Different Fibonacci Indices

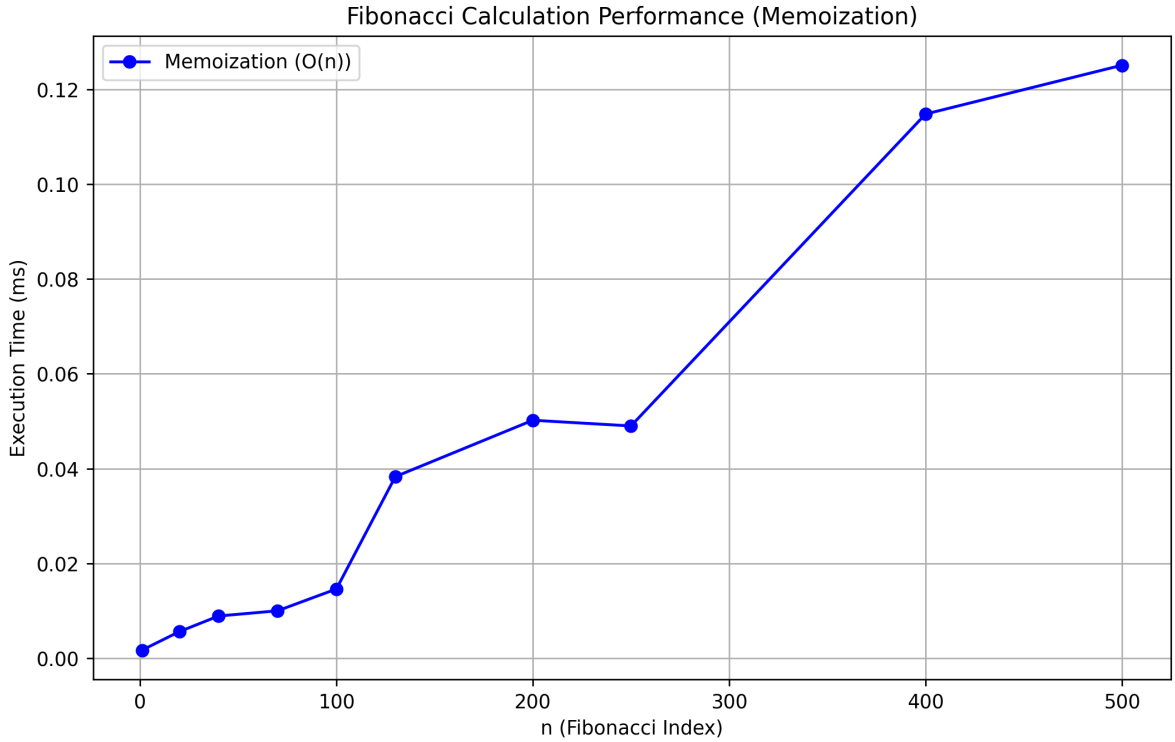

Figure 10: Memoization Execution Time Growth, Confirming $O(n)$ Complexity

2.6 Recursive Method

2.6.1 Algorithm Explanation

The Recursive Fibonacci method follows the *mathematical definition* of Fibonacci numbers:

$$F(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F(n-1) + F(n-2), & n \geq 2 \end{cases}$$

This approach is *intuitive* but highly *inefficient*, as it *recomputes* values multiple times, leading to an exponential time complexity.

Time and Space Complexity

- **Time Complexity:** $O(2^n)$ - Exponential growth due to repeated calculations.
- **Space Complexity:** $O(n)$ - Due to recursive call stack depth.

2.6.2 Python Implementation

```
def nth_fibonacci_recursive(n):
    if n <= 1:
        return n
    return nth_fibonacci_recursive(n - 1) + nth_fibonacci_recursive(n -
        2)
```

2.6.3 Results Analysis

Execution Time: The Recursive method exhibits *exponential growth*, making it impractical for large n. *Table 11* presents execution times, and *Figure 12* visualizes the rapid time increase.

Execution Times using Naïve Recursion (O(2ⁿ))						
1	5	10	15	20	25	30
0.0005	0.0022	0.0067	0.0755	0.7618	8.4547	89.7111

Figure 11: Recursive Fibonacci Execution Time Growth (Exponential)

Visualization: The graph in *Figure 12* confirms the exponential nature of execution time growth.

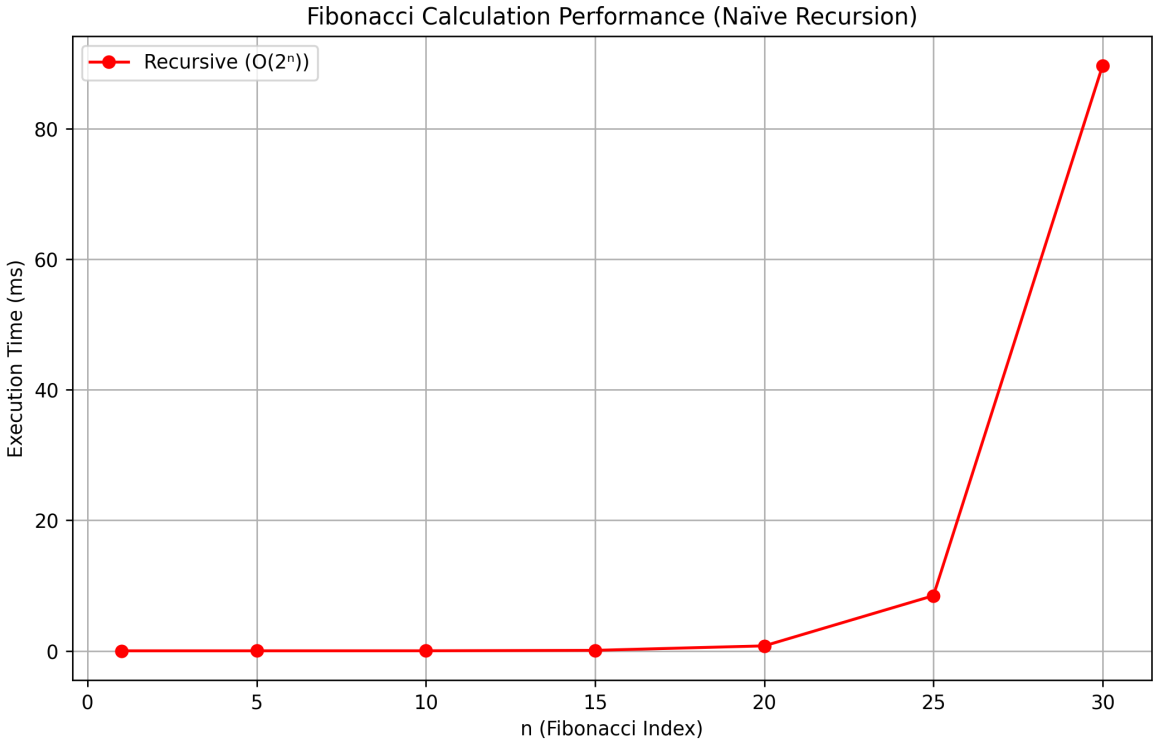

Figure 12: Recursive Fibonacci Execution Time Growth Graph

2.7 Fibonacci Space-Optimized Method

2.7.1 Algorithm Explanation

The *Space-Optimized Fibonacci method* reduces memory usage by storing only the *last two computed values*, instead of maintaining a full array. The recurrence relation remains:

$$F(n) = F(n-1) + F(n-2)$$

Instead of an array, we use *two variables* a and b, updating them iteratively.

Time and Space Complexity

- **Time Complexity:** $O(n)$ - Each Fibonacci number is computed once.
- **Space Complexity:** $O(1)$ - Only two variables are used.

2.7.2 Python Implementation

```
def nth_fibonacci_space_optimized(n):
    if n <= 1:
        return n
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b
```

2.7.3 Results Analysis

Execution Time: The Space-Optimized method maintains *linear execution time* but improves memory usage. *Figure 13* shows the measured times.

```
Execution Times using Space-Optimized Iterative Fibonacci Calculation (O(n)):
   1      20     40     70     100    130    200    250    400    500
0.0005 0.0032 0.002 0.0025 0.0032 0.0037 0.0059 0.0072 0.0134 0.0168
```

Figure 13: Space-Optimized Fibonacci Execution Time Graph

Visualization: The performance graph in *Figure 14* confirms the $O(n)$ *growth rate*, while requiring minimal memory.

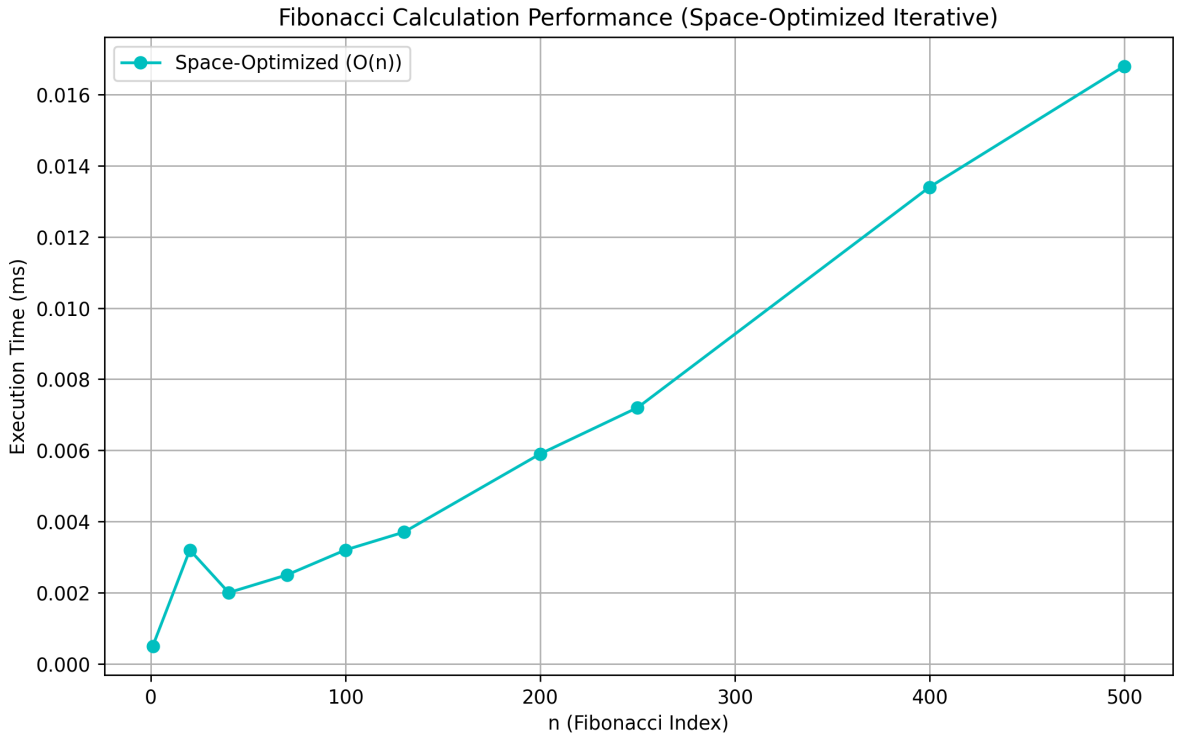

Figure 14: Space-Optimized Fibonacci Execution Time Graph

3 Performance Analysis

3.1 Execution Time Comparison

To evaluate the efficiency of different Fibonacci computation methods, execution times were measured across varying input sizes. The results are visualized in *Figure 15*, which highlights the differences in execution time among the methods.

3.2 Complexity Comparison

The following table summarizes the time complexity of each Fibonacci method:

Method	**Time Complexity**	**Efficiency**
Recursive	$O(2^n)$	Inefficient for large n
Memoization	$O(n)$	Efficient, avoids recomputation
Bottom-Up DP	$O(n)$	Good for moderate n
Space-Optimized DP	$O(n)$	Reduces memory usage
Matrix Exponentiation	$O(\log n)$	Fast for large n
Fast Doubling	$O(\log n)$	Most efficient for very large n
Binet's Formula	$O(1)$	Instant but imprecise for $n > 70$

Table 1: Time Complexity Comparison of Fibonacci Methods

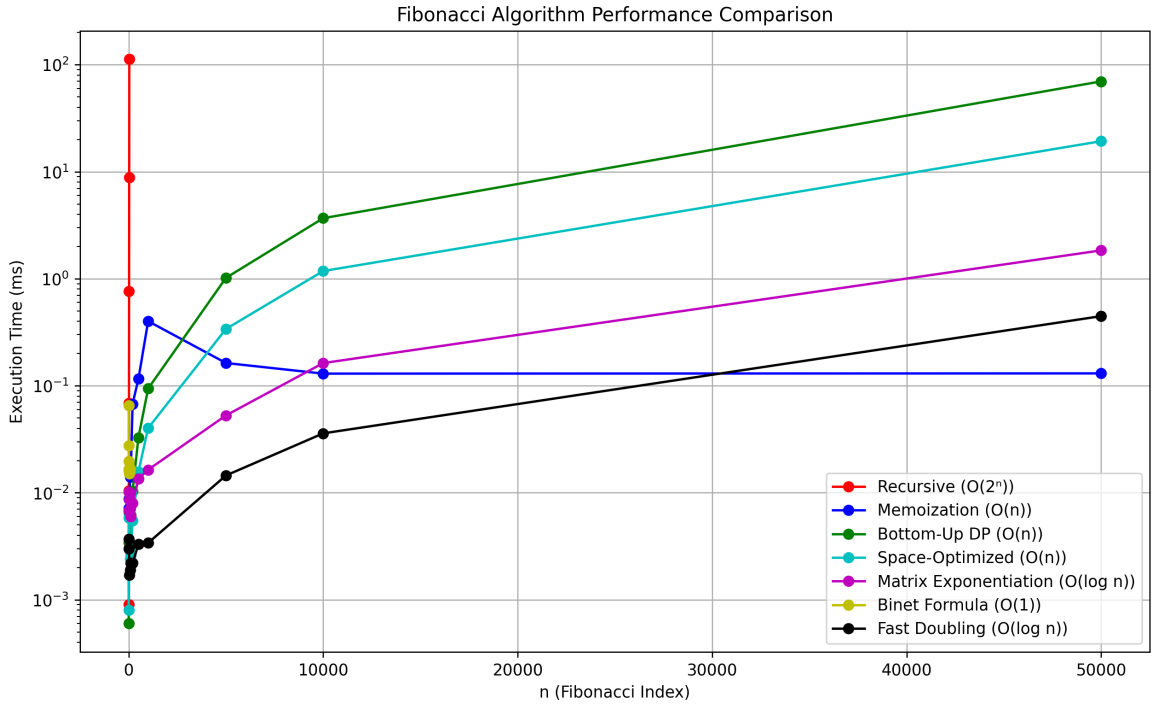

Figure 15: Performance Comparison of Fibonacci Algorithms

3.3 Performance Insights

-*The Recursive Method* ($O(2^n)$) becomes *impractical* for $n > 30$ due to exponential growth.

- *Memoization* and *Bottom-Up DP* ($O(n)$) significantly reduce computation time by avoiding redundant calculations.

- *Space-Optimized DP* maintains $O(n)$ *complexity* but reduces memory usage compared to Bottom-Up DP.

- *Fast Doubling* and *Matrix Exponentiation* ($O(\log n)$) outperform all iterative approaches and are *best for large values of* n.

- *Binet's Formula* ($O(1)$) is the fastest but becomes inaccurate for $n > 70$ due to floating-point limitations.

Best Method Based on Input Size:

- For **small** n **(below 20)**, any method is feasible.
- For **moderate** n **(20–1000)**, *Bottom-Up DP or Space-Optimized DP* is ideal.
- For **large** n **(above 1000)**, *Fast Doubling or Matrix Exponentiation* should be used.

3.4 Final Thoughts

The results confirm that *Fast Doubling* and *Matrix Exponentiation* are the most efficient methods for large-scale Fibonacci computations, while *Memoization and Bottom-Up DP* are effective for moderate values of n. The *Recursive Method is only useful for educational purposes* due to its exponential growth.

4 Conclusion

This laboratory work provided valuable insights into the efficiency of different Fibonacci computation methods. By analyzing execution times and complexity, I gained a deeper understanding of how algorithmic choices impact performance. The experiment demonstrated the limitations of naive recursion, which quickly became impractical due to exponential growth, while memoization and dynamic programming significantly improved efficiency by avoiding redundant computations. The fast doubling and matrix exponentiation methods emerged as the most scalable approaches, confirming their superiority for large-scale calculations.

Through this analysis, I learned the importance of selecting the right algorithm based on input size and computational constraints. The performance graphs and complexity comparisons highlighted how different approaches behave under varying conditions, reinforcing the theoretical concepts of time complexity. Additionally, the experiment emphasized the trade-offs between speed, memory usage, and numerical precision, particularly in the case of Binet's formula, which, despite its constant time complexity, proved unreliable for large values of n.

Overall, this laboratory work deepened my understanding of algorithmic efficiency and reinforced the importance of choosing the most suitable method for a given problem. By benchmarking and visualizing execution times, I was able to see firsthand how theoretical complexity translates into practical performance, making this analysis a valuable learning experience.

5 GitHub Repository

All source code, performance benchmarks, and LaTeX documentation for this laboratory work are available in the following GitHub repository:

`https://github.com/AlexandruRudoi/AA_Course/tree/Lab_1`

This repository contains:

- Implementations of all Fibonacci computation methods.
- Benchmarking scripts for execution time analysis.

- Performance graphs and complexity comparisons.
- Full LaTeX report for documentation and analysis.

By exploring the repository, one can replicate the experiments, modify the code for further optimizations, or use it as a reference for understanding different Fibonacci computation techniques. Contributions and discussions are welcome.