MINISTRY OF EDUCATION AND RESEARCH
OF THE REPUBLIC OF MOLDOVA

TECHNICAL UNIVERSITY OF MOLDOVA

FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE AND AUTOMATION ENGINEERING

Report

Laboratory Work No. 1

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

Elaborated by:
st. gr. FAF-241
Tihon Aurelian-Mihai

Verified by:
univ. assist.
Fiștic Cristofor

Chișinău – 2026

Contents

1. INTRODUCTION

1.1. Objective

The objective of this laboratory work is to study, implement, and empirically analyze different algorithms for determining the N-th Fibonacci term. The analysis focuses on Time Complexity and Space Complexity to understand the efficiency of each method when applied to small, large, and very large inputs.

1.2. Theoretical Notes

An alternative to mathematical analysis of complexity (Big O notation) is empirical analysis. This is useful for obtaining preliminary information on the complexity class of an algorithm, comparing the efficiency of two or more algorithms for solving the same problem, and comparing several implementations.

In this analysis, the following steps were followed:

1. Implementation of algorithms in Python.
2. Generation of input data sets (Small N for recursive, Large N for optimized methods).
3. Measurement of execution time using `time.perf_counter`.
4. Measurement of memory usage (RAM) for specific large inputs.
5. Comparison of results using graphs and tables.

1.3. Tasks

1. Implement 6 algorithms for determining Fibonacci n-th term (Recursive, DP Array, Space Optimized, Matrix Power, Fast Doubling, Binet).
2. Analyze empirically the algorithms based on execution time.
3. Compare memory usage for large N (ex: $N = 300,000$).
4. Present the results using graphs and execution screenshots.

2. ALGORITHM ANALYSIS

All algorithms were implemented in Python. The tests were conducted on a standard machine, utilizing the `time` library for performance tracking and `tracemalloc` for memory profiling.

2.1. Recursive Method

2.1.1 Description

The recursive method is the direct translation of the mathematical definition $F_n = F_{n-1} + F_{n-2}$. While simple to implement, it performs redundant calculations, leading to an exponential time complexity of $O(2^n)$.

2.1.2 Implementation

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

2.1.3 Empirical Results

As shown in Figure 1, the execution time remains negligible for $N < 30$ but spikes dramatically afterwards. For $N = 40$, the time reached approximately 15 seconds. This confirms the exponential nature of the algorithm ($O(2^n)$), making it unusable for real-world applications.

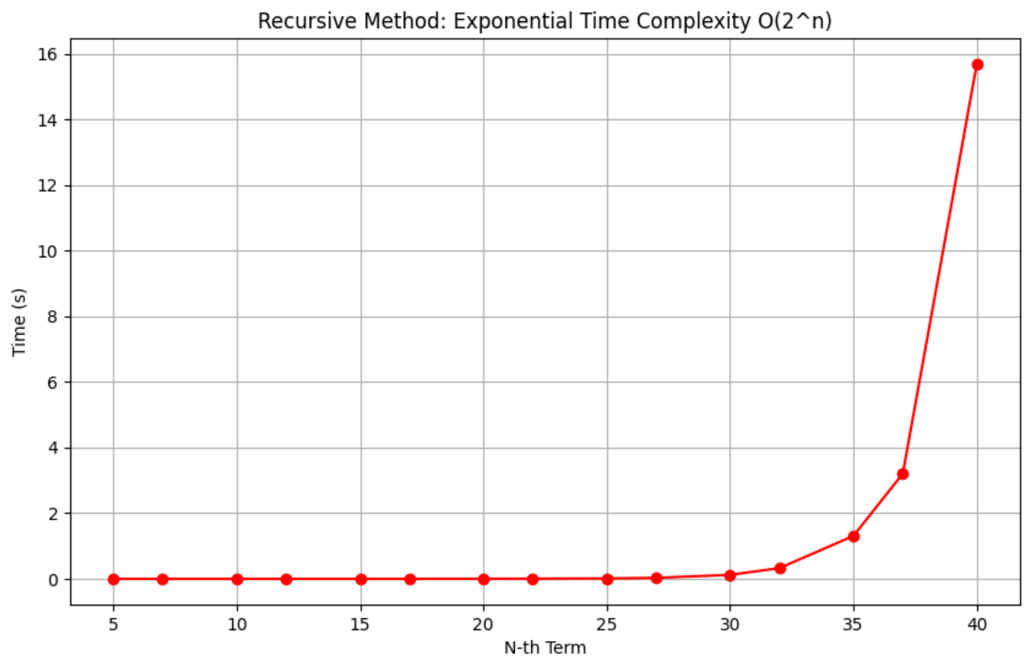


Figure 1: Exponential Growth of Recursive Method

2.2. Dynamic Programming (Array)

2.2.1 Description

This method optimizes recursion by storing the results of subproblems in an array (memoization/tabulation). This reduces time complexity to $O(n)$ but requires $O(n)$ space to store the list.

2.2.2 Implementation

```python
def fib_dp_array(n):
    if n <= 1: return n
    fib_list = [0, 1]
    for i in range(2, n + 1):
        fib_list.append(fib_list[i-1] + fib_list[i-2])
    return fib_list[n]
```

2.2.3 Empirical Results

The graph in Figure 2 demonstrates a linear relationship between input size N and time. However, a significant limitation was observed during memory testing: for $N = 300,000$, this method consumed over **3.9 GB of RAM** (see Experimental Data section), proving that $O(n)$ space complexity is dangerous for very large inputs.

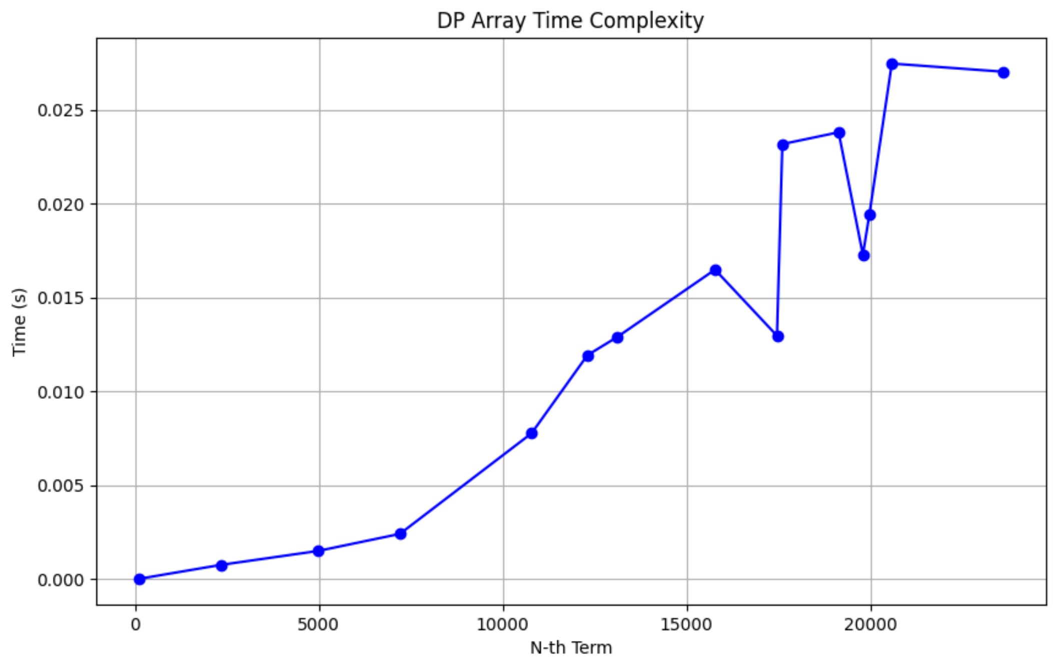


Figure 2: Linear Time Complexity of DP Array Method

2.3. Space Optimized Iterative Method

2.3.1 Description

This algorithm improves upon the DP approach by storing only the last two calculated values (a and b) instead of the entire history. This maintains $O(n)$ time complexity but reduces space complexity to $O(1)$.

2.3.2 Implementation

```python
def fib_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
```

2.3.3 Empirical Results

Figure 3 shows the time complexity is still linear $O(n)$, similar to the array method. However, the memory usage is negligible (81 KB for $N = 300,000$). While efficient, for extremely large N (ex: $N > 100,000$), the linear time iteration becomes noticeably slower than logarithmic methods.

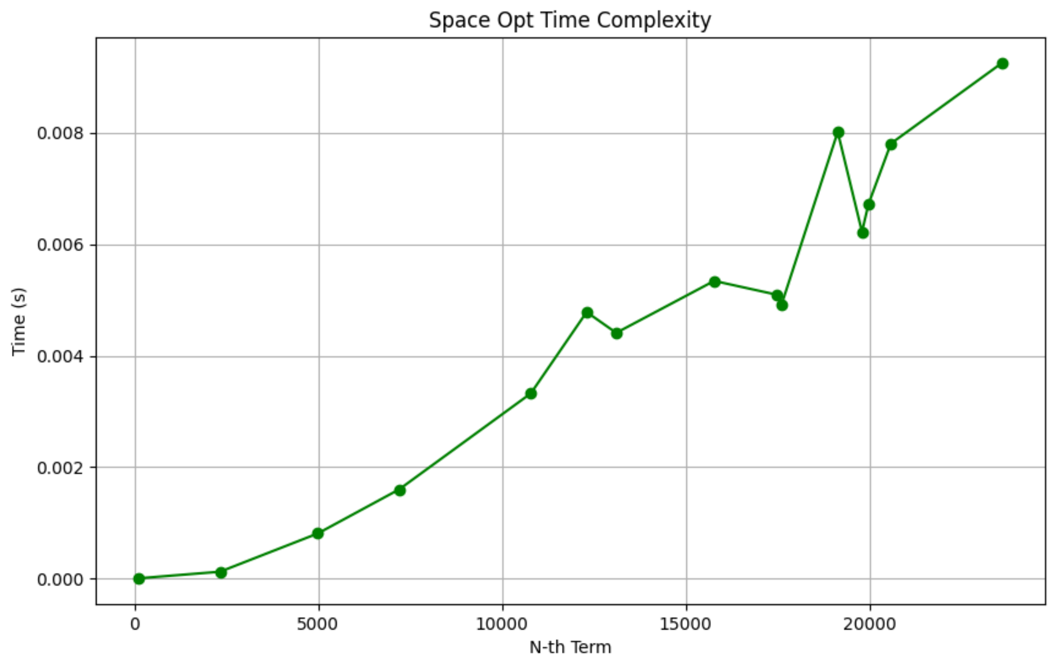


Figure 3: Space Optimized Method Results

2.4. Matrix Power Method

2.4.1 Description

This method utilizes the property that raising the matrix $M = \left(\begin{smallmatrix} 1 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ to the power of n yields the Fibonacci numbers. Using "exponentiation by squaring," we achieve $O(\log n)$ time complexity.

2.4.2 Implementation

```python
def fib_matrix(n):
    # (Multiply and Power functions omitted for brevity)
    if n == 0: return 0
    F = [[1, 1], [1, 0]]
    power(F, n - 1)
    return F[0][0]
```

2.4.3 Empirical Results

The graph below demonstrates significant speed improvements over the iterative methods. The fluctuations in the graph are due to the varying number of bitwise operations required for different values of N, but the overall trend is logarithmic.

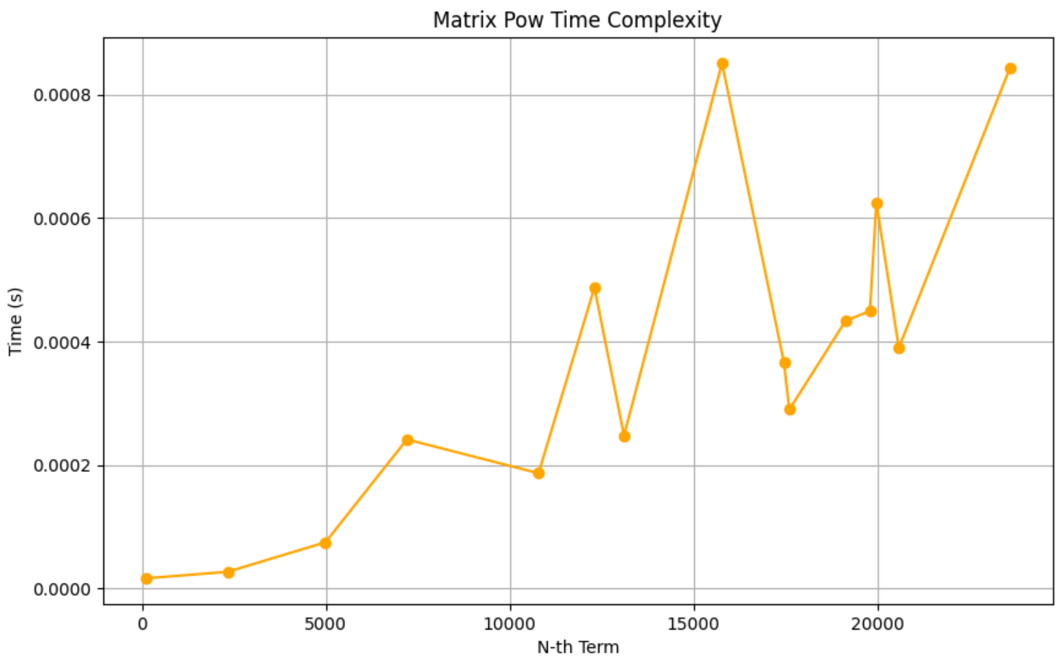


Figure 4: Matrix Power Method (Logarithmic Complexity)

2.5. Fast Doubling Method

2.5.1 Description

Fast Doubling is a further optimization of the Matrix method. It removes the overhead of matrix multiplication loops by using derived formulas to calculate $F(2n)$ and $F(2n+1)$ directly.

2.5.2 Implementation

```python
def fib_fast_doubling(n):
    def _fib(n):
        if n == 0: return (0, 1)
        a, b = _fib(n >> 1)
        c = a * (2 * b - a)
        d = a * a + b * b
        if n & 1: return (d, c + d)
        else: return (c, d)
    return _fib(n)[0]
```

2.5.3 Empirical Results

This was empirically the fastest algorithm. As seen in the combined comparison later, it consistently outperforms even the Matrix Power method due to lower constant factors.

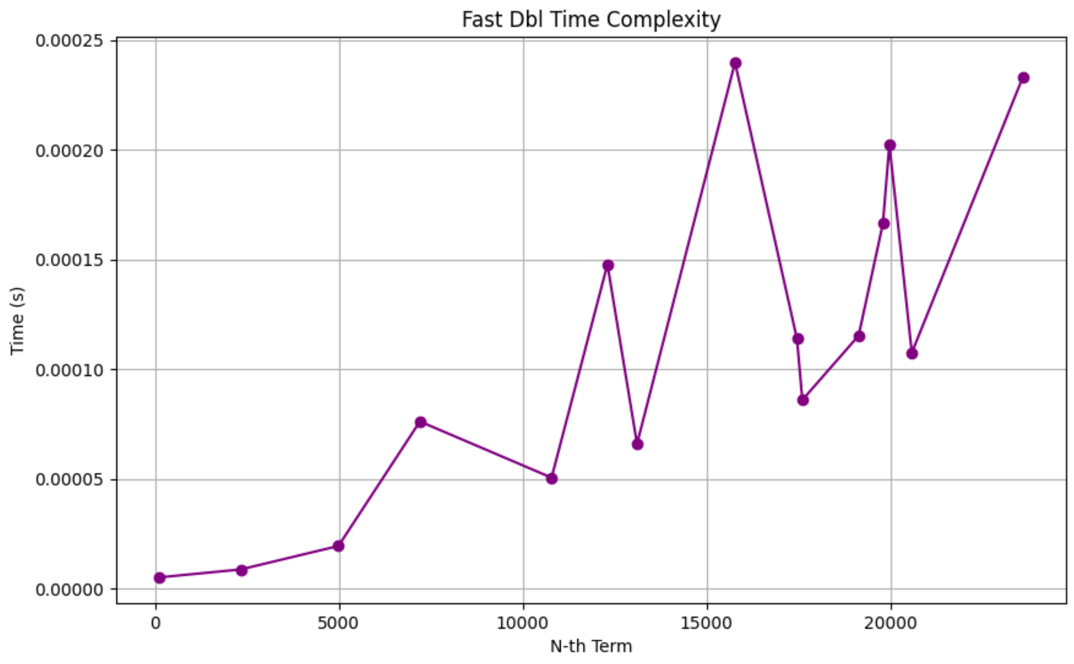


Figure 5: Fast Doubling Method Performance

2.6. Binet Formula (High Precision)

2.6.1 Description

The Binet formula uses the Golden Ratio (ϕ) to calculate F_n. Standard floating-point types lose precision after $N = 71$. To support large N, this implementation uses Python's `decimal` module with high precision.

2.6.2 Implementation

```python
def fib_binet(n):
    getcontext().prec = 10000
    phi = (Decimal(1) + Decimal(5).sqrt()) / Decimal(2)
    return int(round((phi**n - (1 - phi)**n) / Decimal(5).sqrt()))
```

2.6.3 Empirical Results

While theoretically $O(1)$, the use of software-based high-precision arithmetic (`Decimal`) introduces overhead. As seen in Figure 6, it is slower than the Fast Doubling method for large inputs but remains faster than the linear iterative approach.

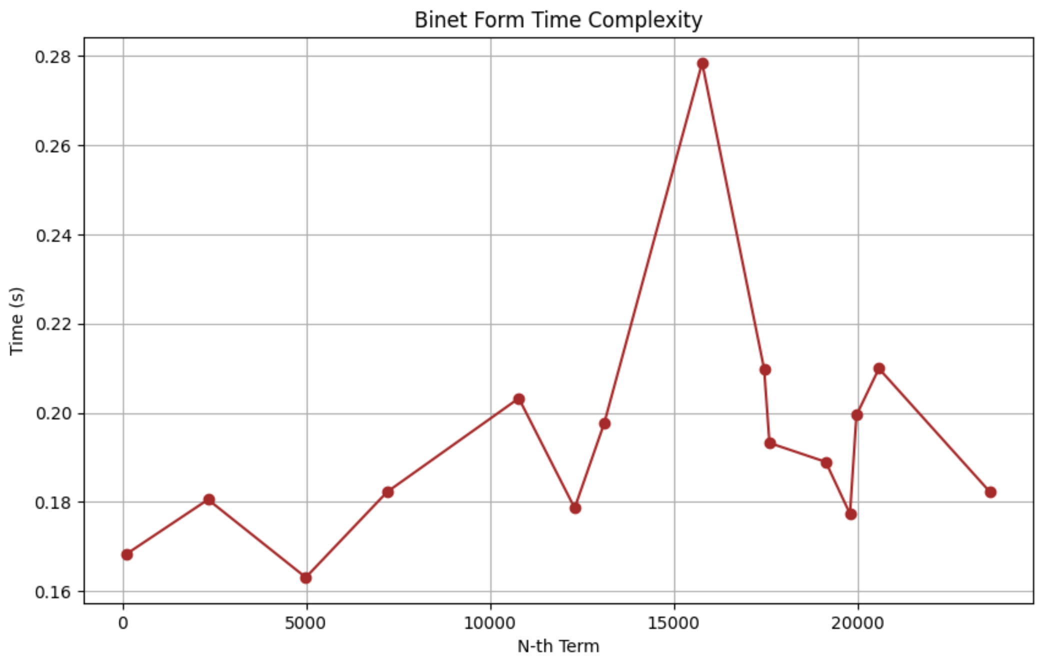


Figure 6: Binet Formula (Decimal Implementation)

3. COMPARISON & EXPERIMENTAL DATA

3.1. Combined Time Analysis

Figure 7 presents a combined view of all optimized algorithms. It clearly shows that **Fast Doubling** (Purple) and **Matrix Power** (Green) are superior for large inputs. The **Binet Formula** (Red) shows higher latency due to ‘Decimal‘ overhead.

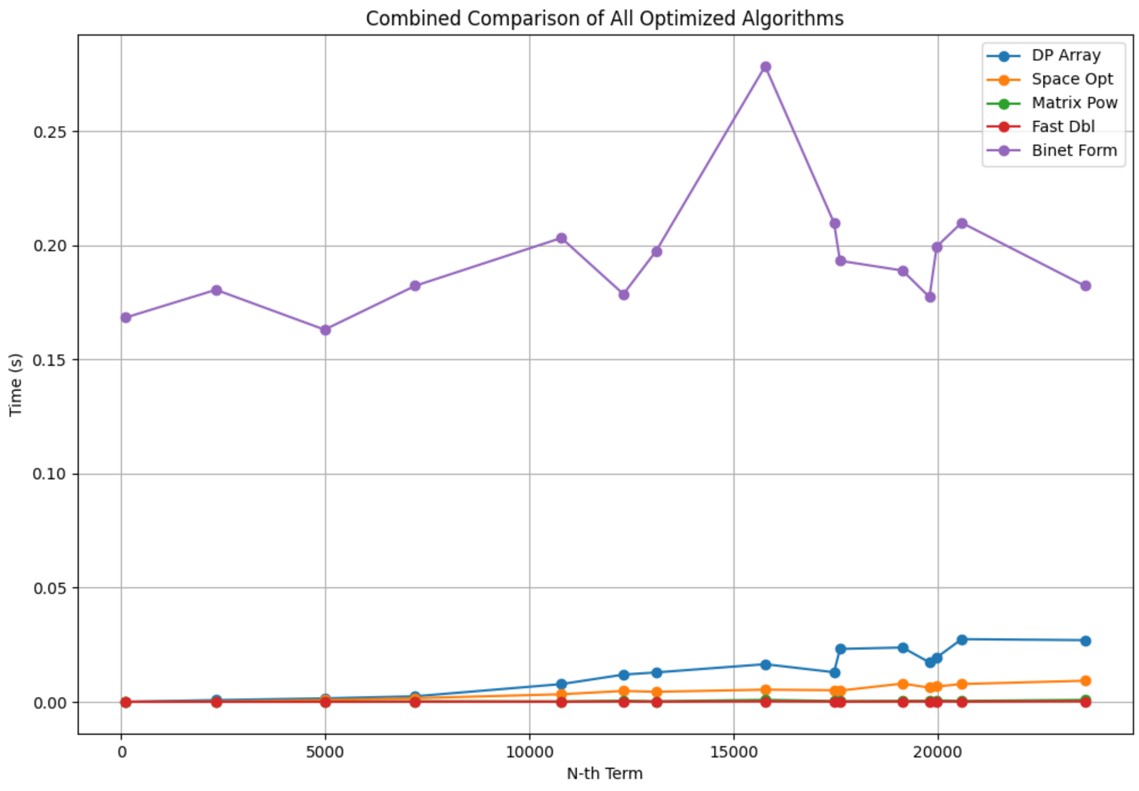
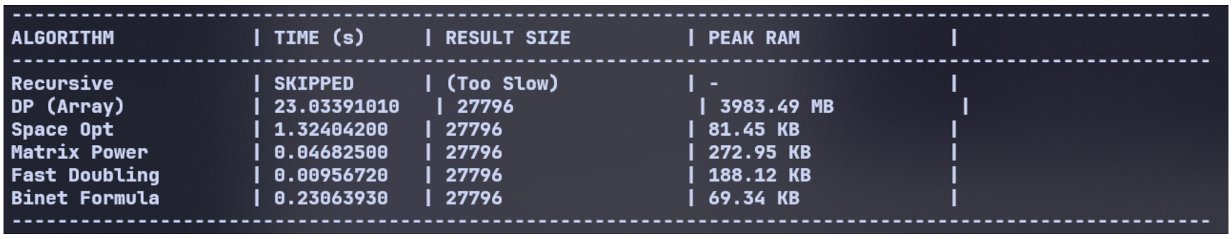


Figure 7: Combined Comparison of Optimized Algorithms

3.2. Memory Usage Analysis

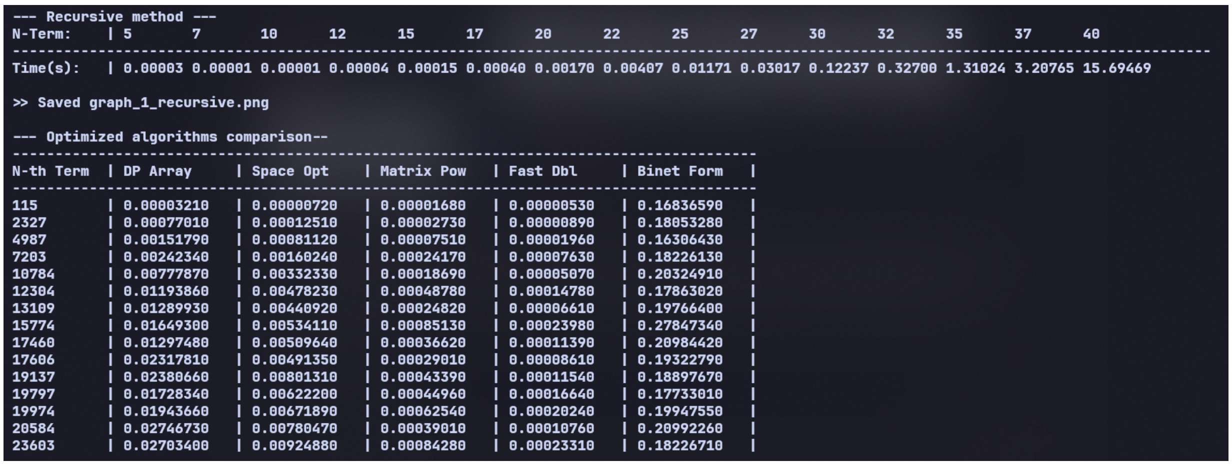
A specific test was run for $N = 300,000$ to compare RAM usage. The results (Figure 8) highlight the critical flaw of the standard DP Array method.

- **DP (Array):** 3983.49 MB (Crash risk)
- **Space Optimized:** 81.45 KB
- **Fast Doubling:** 188.12 KB



```
-----------------------------------------------------------------------------------------------------------
ALGORITHM            | TIME (s)     | RESULT SIZE         | PEAK RAM            |
-----------------------------------------------------------------------------------------------------------
Recursive            | SKIPPED      | (Too Slow)          | -                   |
DP (Array)           | 23.03391010   | 27796               | 3983.49 MB          |
Space Opt            | 1.32404200   | 27796               | 81.45 KB            |
Matrix Power         | 0.04682500   | 27796               | 272.95 KB           |
Fast Doubling        | 0.00956720   | 27796               | 188.12 KB           |
Binet Formula        | 0.23063930   | 27796               | 69.34 KB            |
-----------------------------------------------------------------------------------------------------------
```

Figure 8: Memory Profile Screenshot for N=300,000



```
--- Recursive method ---
N-Term:     | 5       7       10      12      15      17      20      22      25      27      30      32      35      37      40
-------------------------------------------------------------------------------------------------------------------------------------
Time(s):    | 0.00003 0.00001 0.00001 0.00004 0.00015 0.00040 0.00170 0.00407 0.01171 0.03017 0.12237 0.32700 1.31024 3.20765 15.69469

>> Saved graph_1_recursive.png

--- Optimized algorithms comparison--
-----------------------------------------------------------------------------------------
N-th Term   | DP Array      | Space Opt     | Matrix Pow    | Fast Dbl      | Binet Form    |
-----------------------------------------------------------------------------------------
115         | 0.00003210    | 0.00000720    | 0.00001680    | 0.00000530    | 0.16836590    |
2327        | 0.00077010    | 0.00012510    | 0.00002730    | 0.00000890    | 0.18053280    |
4987        | 0.00151790    | 0.00081120    | 0.00007510    | 0.00001960    | 0.16306430    |
7203        | 0.00242340    | 0.00160240    | 0.00024170    | 0.00007630    | 0.18226130    |
10784       | 0.00777870    | 0.00332330    | 0.00018690    | 0.00005070    | 0.20324910    |
12304       | 0.01193860    | 0.00478230    | 0.00048780    | 0.00014780    | 0.17863020    |
13109       | 0.01289930    | 0.00440920    | 0.00024820    | 0.00006610    | 0.19766400    |
15774       | 0.01649300    | 0.00534110    | 0.00085130    | 0.00023980    | 0.27847340    |
17460       | 0.01297480    | 0.00509640    | 0.00036620    | 0.00011390    | 0.20984420    |
17606       | 0.02317810    | 0.00491350    | 0.00029010    | 0.00008610    | 0.19322790    |
19137       | 0.02380660    | 0.00801310    | 0.00043390    | 0.00011540    | 0.18897670    |
19797       | 0.01728340    | 0.00622200    | 0.00044960    | 0.00016640    | 0.17733010    |
19974       | 0.01943660    | 0.00671890    | 0.00062540    | 0.00020240    | 0.19947550    |
20584       | 0.02746730    | 0.00780470    | 0.00039010    | 0.00010760    | 0.20992260    |
23603       | 0.02703400    | 0.00924880    | 0.00084280    | 0.00023310    | 0.18226710    |
```

Figure 9: Console Output: Recursive vs Optimized Timing

4. CONCLUSION

In this laboratory work, six algorithms for calculating Fibonacci numbers were implemented and analyzed. The recursive method is suitable only for educational purposes (N < 40) due to its O(2) complexity, while the DP array approach is fast with O(n) time but risky in terms of memory usage (O(n)) for large inputs. The space-optimized method emerges as the best general-purpose solution, balancing simplicity, speed, and minimal memory usage (O(1)). Matrix fast doubling is the superior option for high-performance computing with extremely large numbers (N > 10,000), offering O(log n) complexity. The Binet formula works well for N < 70 using standard floating-point arithmetic, but for larger N its reliance on arbitrary-precision arithmetic makes it slower than fast doubling. Overall, for standard applications the space-optimized iterative approach is recommended, while for scientific computing involving massive Fibonacci numbers, fast doubling is the optimal choice.