Ministry of Education and Research of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Software Engineering Department

Algorithm Analysis

Laboratory work #1

An Empirical Analysis of Algorithms for Determining the Fibonacci N-th Term

Author:
Mihaela Catan
std. gr. FAF-231

Verified:
Cristofor Fiștic

Chișinău 2025

Contents

1 Objective

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

2 The Task

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.

3 Theorethical Notes

Empirical Analysis is an alternative to mathematical analysis of complexity, which helps to obtain information on the efficiency of an algorithm based on experimental observations. It is useful for:

1. Preliminary insights into the complexity class of an algorithm.
2. Comparing the performance of multiple algorithms or implementations.
3. Understanding how an algorithm behaves on a specific machine or system.

Steps in Empirical Analysis:

1. Establish the purpose of the analysis.
2. Decide the efficiency metric.
3. Establish the properties of the input data in relation to which the analysis is performed.
4. Implement the algorithms in a programming language.
5. Generate multiple sets of inputs.
6. Analyze the results.

4 Introduction

The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding ones, typically starting with 0 and 1. The sequence is defined recursively as follows:

$$F(0) = 0, \quad F(1) = 1$$

$$F(n) = F(n-1) + F(n-2) \quad \text{for} \quad n \geq 2$$

The first few terms of the Fibonacci sequence are:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

4.1 History

The Fibonacci sequence was introduced to the West by the Italian mathematician Leonardo of Pisa, also known as Fibonacci, in his 1202 book *Liber Abaci*. However, the sequence had been known in India much earlier. Fibonacci used it to model the growth of rabbit populations, which is why the sequence is also known as the *rabbit problem*. In this context, the sequence models how a pair of rabbits produces another pair every month, and the total number of pairs follows the Fibonacci sequence.

4.2 Properties of the Fibonacci Sequence

The Fibonacci sequence has several interesting mathematical properties:

- **Golden Ratio**: As the Fibonacci sequence progresses, the ratio of successive terms approaches the golden ratio, $\phi \approx 1.618$. That is:

$$\lim_{n \to \infty} \frac{F(n+1)}{F(n)} = \phi$$

- **Parity**: The Fibonacci sequence follows a repeating pattern of even and odd numbers.

- **Sum of Fibonacci Numbers**: The sum of the first n Fibonacci numbers is given by:

$$\sum_{i=0}^{n} F(i) = F(n+2) - 1$$

4.3 Applications

The Fibonacci sequence appears in various fields of mathematics and science:

- **Nature**: Fibonacci numbers can be observed in the arrangement of leaves, flowers, and seeds in plants. For example, the number of petals in many flowers is a Fibonacci number, and the spiral patterns in pinecones and sunflower heads follow Fibonacci patterns.
- **Computer Science**: The Fibonacci sequence is important in algorithm design, particularly in dynamic programming and search algorithms. Fibonacci heaps, for example, are a data structure used to optimize certain graph algorithms.
- **Art and Architecture**: The Fibonacci sequence is closely related to the golden ratio, which is often used in art and architecture to achieve aesthetically pleasing proportions.
- **Financial Markets**: Fibonacci retracement levels are used in technical analysis to predict potential support and resistance levels in financial markets.

4.4 Binet's Formula

In addition to the recursive definition, there is a closed-form expression for the n-th Fibonacci number, known as *Binet's formula*:

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

This formula allows for the direct calculation of Fibonacci numbers without the need for recursion or iteration, though it involves irrational numbers.

5 The Comparison Metric

The time of execution $T(n)$ for each algorithm will be the primary metric. Execution time will be measured in seconds using precise timers. In this analysis, the space usage in bytes will also be provided, calculated via the resources that have been used for computing the results.

6 Technical Implementation

This analysis implements eight algorithms and evaluates their time and space complexities. The experiments are conducted on a backend hosted on Google Compute Engine, utilizing Python 3 and approximately 12 GB of RAM. While this environment offers a suitable platform for algorithmic analysis, several limitations and potential sources of error must be acknowledged. These include the possibility that certain algorithms may exhibit suboptimal performance on larger input sizes, as well as inherent inaccuracies in time and memory measurements due to factors such as Python's interpreter overhead and the effects of garbage collection.

6.1 Input Sequences

For the analysis, the following two sequences of inputs will be used:

- **For the recursive approach**, which typically works only for small inputs:

$$10, 20, 30, 35, 40$$

- **For the remaining algorithms**:

$$10, 100, 1000, 2500, 5000$$

6.2 Recursive Implementation

A recursive approach can be employed to solve the Fibonacci number problem, as each Fibonacci number $F(n)$ is dependent on the two preceding Fibonacci numbers. The problem is repeatedly broken down into smaller subproblems until the base cases are reached.

The recurrence relation is expressed as follows:

$$F(n) = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1, \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

In this formulation, the base case corresponds to the trivial Fibonacci numbers $F(0) = 0$ and $F(1) = 1$, while the recursive case captures the essence of the Fibonacci sequence, where each number is the sum of the two preceding numbers.

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

The complexity of the algorithm is the following:

Time complexity: $O(2^n)$

Space complexity: $O(n)$

Since this algorithm has a time complexity of $O(2^n)$, the first sequence will be used to analyze the algorithm.

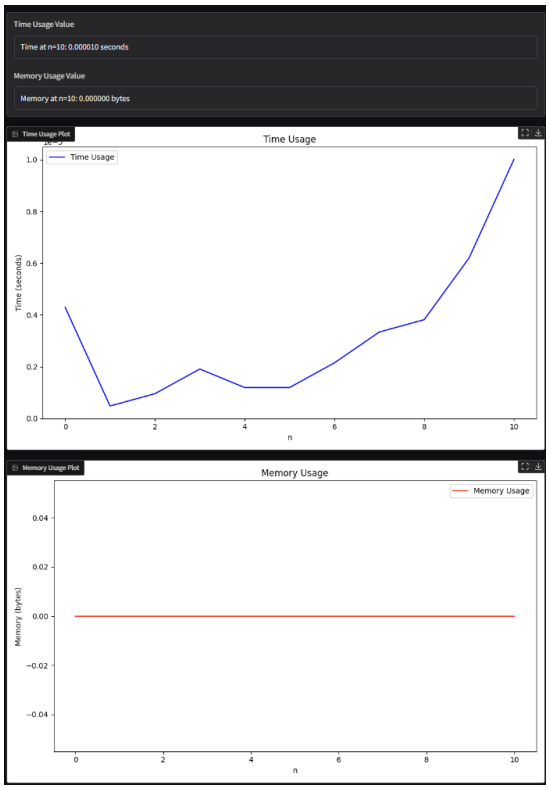

Figure 1: Recursive Algorithm results for n=10

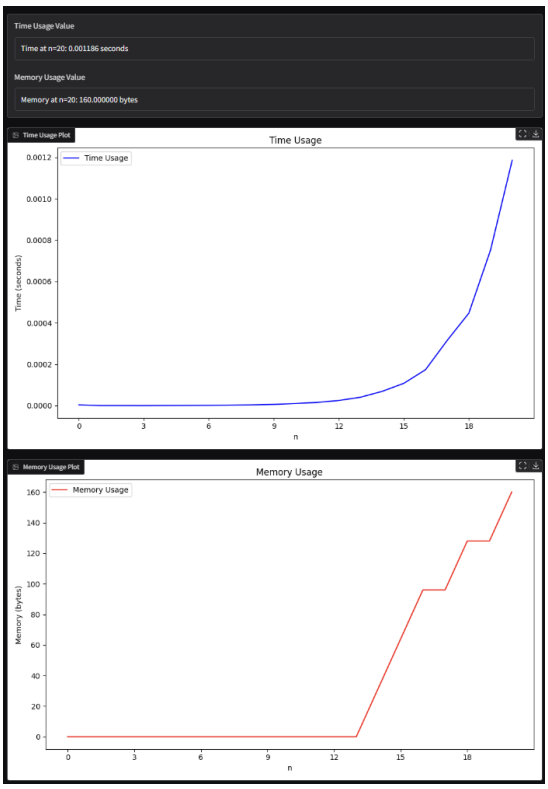

Figure 2: Recursive Algorithm results for n=20

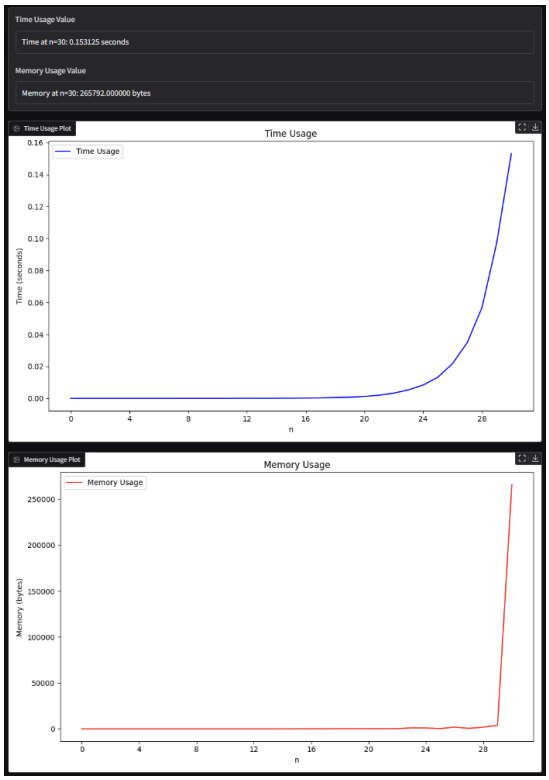

Figure 3: Recursive Algorithm results for n=30

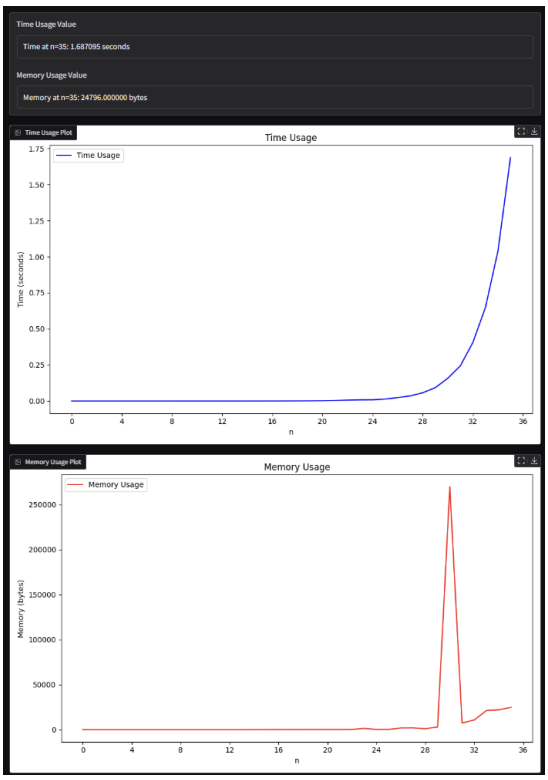

Figure 4: Recursive Algorithm results for n=35

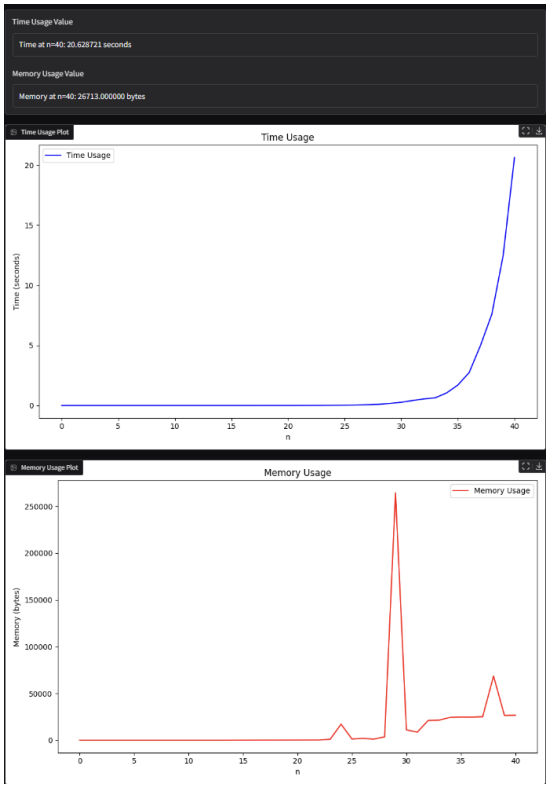

Figure 5: Recursive Algorithm results for n=40

In the provided figures, the time usage grows exponentially, with a sharp increase as n moves from 10 to 40.

The memory usage will also increase exponentially but at a slower rate than time usage, as it mainly depends on the recursion depth (stack usage).

6.3 Bottom-Up Implementation

A Bottom-Up approach can be employed to solve the Fibonacci number problem by iteratively calculating the Fibonacci numbers starting from the base cases up to the desired number $F(n)$. This approach avoids redundant calculations by storing previously computed values in a table, which can be accessed as needed.

In this approach, the base cases are initialized, and a table of values is progressively filled until the Fibonacci number for the desired n is obtained.

```
def fibonacci_bottom_up(n):
  if n == 0 or n == 1:
    return n
  else:
    values = np.zeros(n + 1, dtype=int)
    values[0], values[1] = 0, 1
    for i in range(2, n+1):
      values[i] = values[i-1] + values[i-2]
    return values[n]
```

The complexity of the algorithm is as follows:

Time complexity: $O(n)$

Space complexity: $O(n)$

This iterative approach is more efficient than the recursive approach, as it avoids recomputation and reduces the time complexity to $O(n)$.

The second sequence will be used to compute the results for time and space usage.

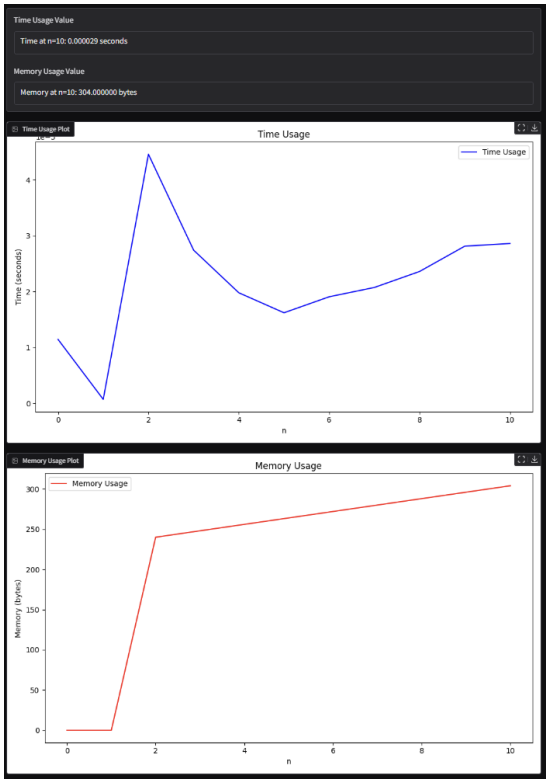

Figure 6: Bottom-Up Algorithm results for n=10

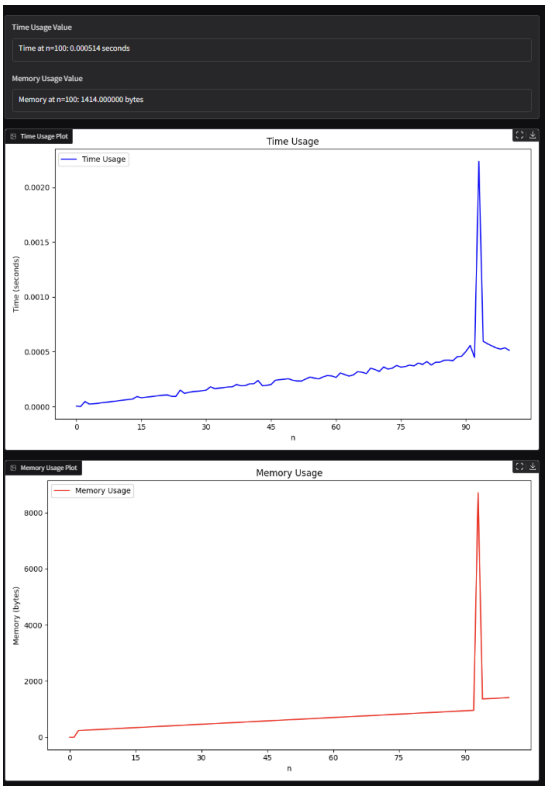

Figure 7: Bottom-Up Algorithm results for n=100

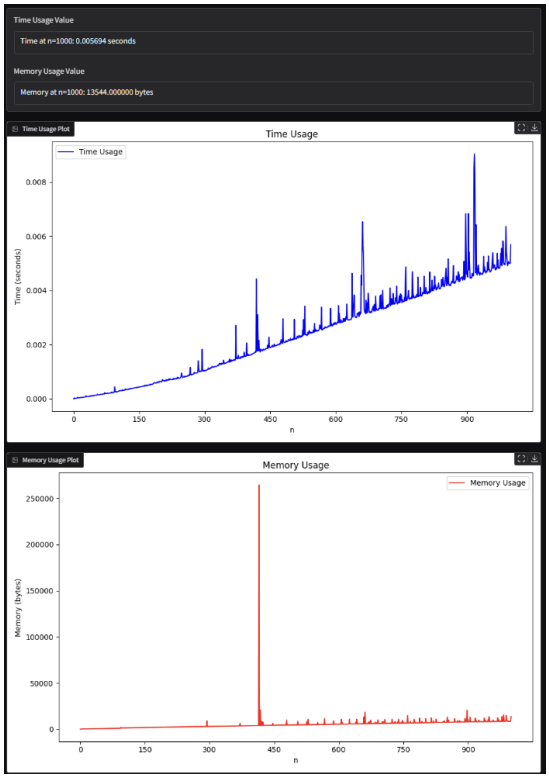

Figure 8: Bottom-Up Algorithm results for n=1000

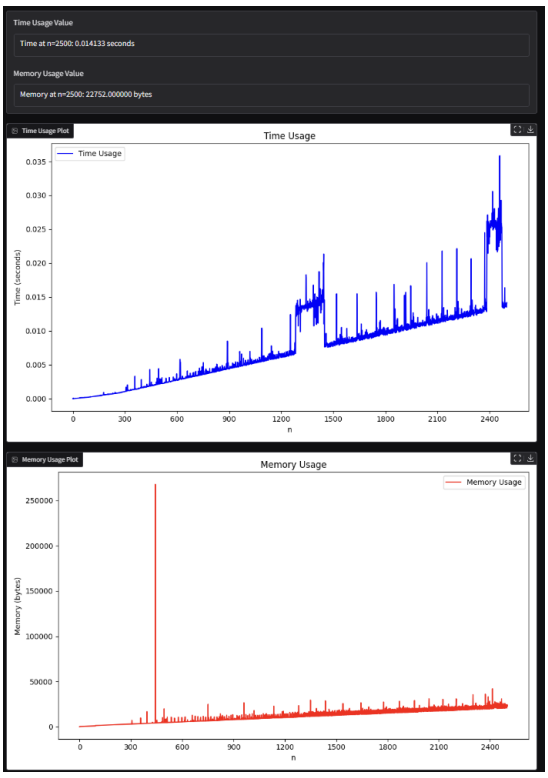

Figure 9: Bottom-Up Algorithm results for n=2500

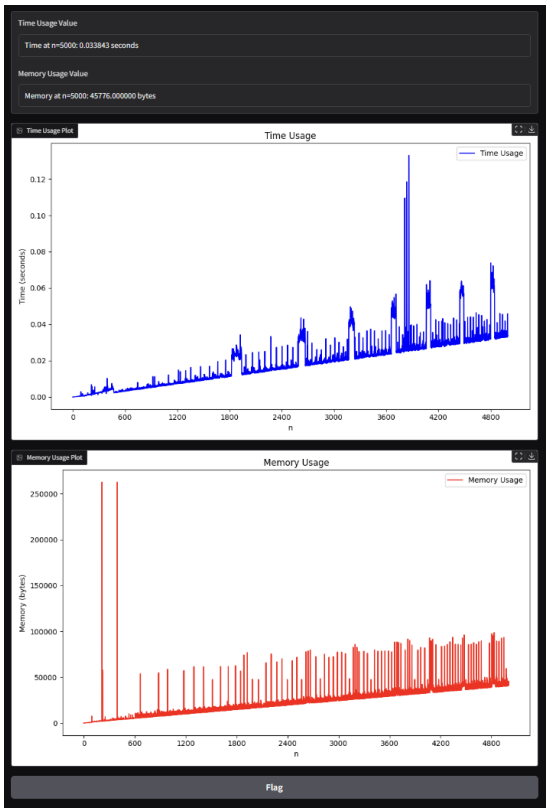

Figure 10: Bottom-Up Algorithm results for n=5000

In the provided figures, the time usage of the Fibonacci computation using the bottom-up approach grows linearly, with the time taken increasing steadily as n increases. This linear growth is expected, as each Fibonacci number is calculated only once and stored for future reference, resulting in $O(n)$ time complexity.

The memory usage also grows linearly, but at a slower rate than time complexity. The memory usage primarily depends on the array `values`, which stores all Fibonacci numbers up to n. This leads to an $O(n)$ space complexity. However, the rate of increase in memory usage is not as steep as the time complexity, as the memory required remains proportional to the number of Fibonacci numbers calculated and stored.

6.4 Top-Down Approach with Memoization Algorithm

A Top-Down approach with memoization optimizes the naive recursive algorithm by storing previously computed Fibonacci numbers, thus avoiding redundant calculations and reducing the time complexity.

$$F(n) = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1, \\ F(n-1) + F(n-2) & \text{if } n > 1, \text{ and if not already computed.} \end{cases}$$

In this case, the result for $F(n)$ is stored in an array and reused when needed.

```
def fibonacci_top_down(n, values):
  if n == 0 or n == 1:
    return n
  elif values[n] != 0:
    return values[n]
  else:
    values[n] = fibonacci_top_down(n - 1, values) +
       fibonacci_top_down(n - 2, values)
    return values[n]

def fibonacci_top_down_helper(n):
  values = np.zeros(n + 1, dtype=int)
  return fibonacci_top_down(n, values)
```

The complexity of the algorithm is as follows:

Time complexity: $O(n)$

Space complexity: $O(n)$

Since this algorithm has a time complexity of $O(n)$, it significantly improves on the naive recursive approach and is suitable for larger inputs, so the second sequence will be used.

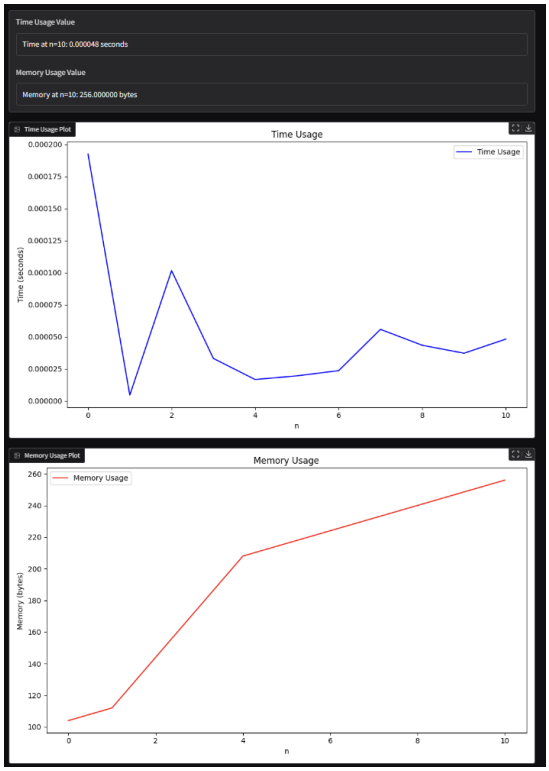

Figure 11: Top-Down Approach with Memoization Algorithm results for n=10

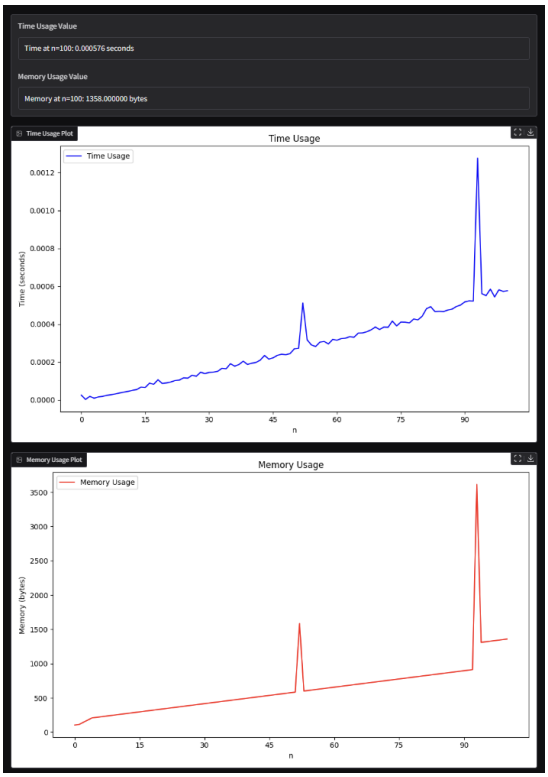

Figure 12: Top-Down Approach with Memoization Algorithm results for n=100

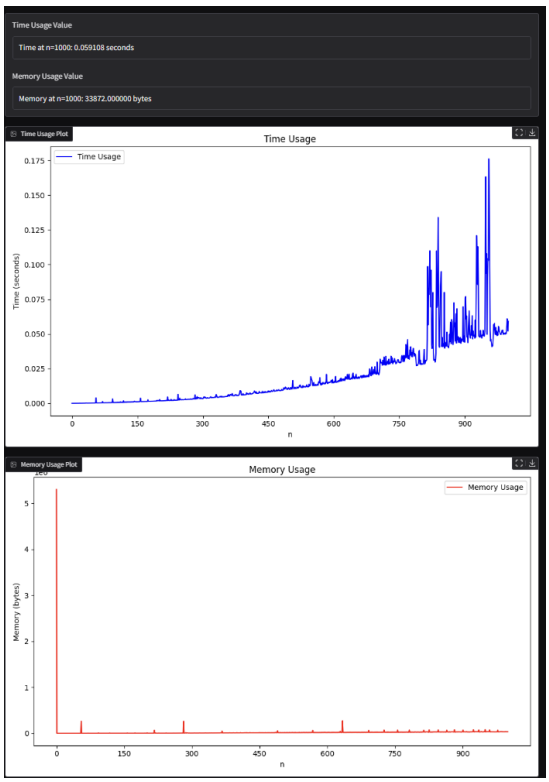

Figure 13: Top-Down Approach with Memoization Algorithm results for n=1000

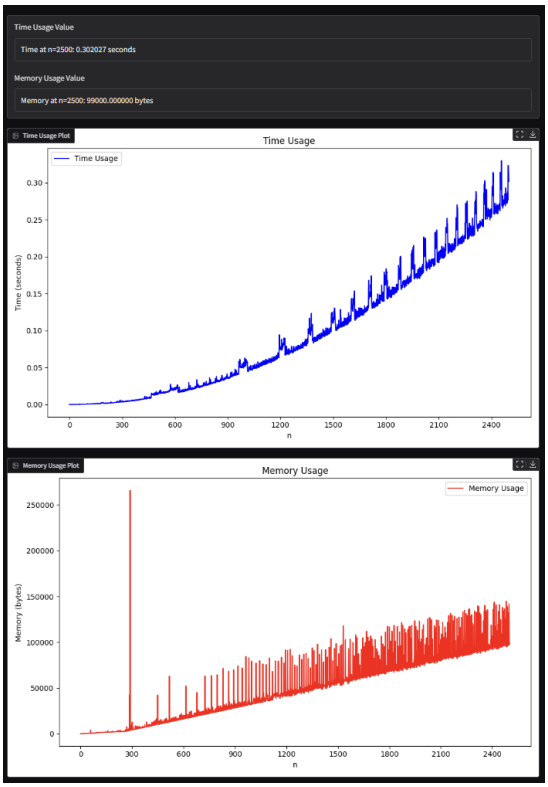

Figure 14: Top-Down Approach with Memoization Algorithm results for n=2500

In the provided figures, the time usage of the Fibonacci computation using the Top-Down approach with memoization grows linearly, with the time taken increasing steadily as n increases. This linear growth is expected, as each Fibonacci number is calculated only once and stored for future reference, resulting in $O(n)$ time complexity.

The memory usage also grows linearly, but at a slower rate than time usage. The memory usage primarily depends on the array `values`, which stores all Fibonacci numbers up to n. This leads to an $O(n)$ space complexity. However, the rate of increase in memory usage is not as steep as the time complexity, as the memory required remains proportional to the number of Fibonacci numbers calculated and stored.

There is no picture result for n=5000, because there was an overflow error encountered.

6.5 Space Optimized Implementation

In this space-optimized approach, the Fibonacci number is calculated iteratively, using only two variables to store the last two Fibonacci numbers at any point in the calculation. This method reduces the space complexity significantly while maintaining a linear time complexity.

The algorithm begins by initializing two variables a and b to store the first two Fibonacci numbers, $F(0) = 0$ and $F(1) = 1$. Then, starting from $n = 2$, the Fibonacci numbers are calculated iteratively in a loop. At each iteration, the next Fibonacci number is computed as the sum of the previous two, and the values of a and b are updated accordingly to store the two most recent Fibonacci numbers.

Since only two variables are needed to compute each Fibonacci number, the space complexity is reduced to $O(1)$, meaning the amount of memory used does not grow with the value of n.

```
def fibonacci_space_optimized(n):
  if n == 0 or n == 1:
    return n
  else:
    a, b = 0, 1
    for i in range(2, n+1):
      temp = a + b
      a = b
      b = temp
    return b
```

The complexity is as follows:

Time complexity: $O(n)$

Space complexity: $O(1)$

This space-optimized approach offers an efficient solution to the Fibonacci number problem, using minimal memory while maintaining a straightforward iterative calculation.

The second sequence will be used to compute the results for time and space usage.

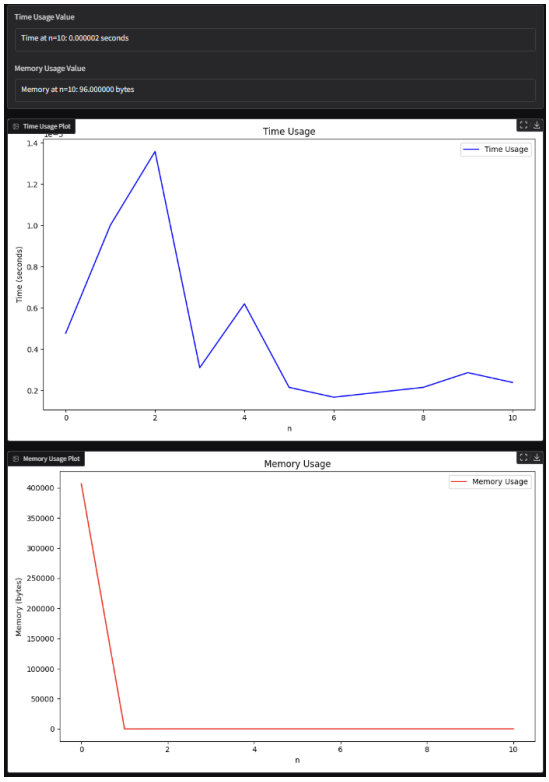

Figure 15: Space Optimized Implementation Algorithm results for n=10

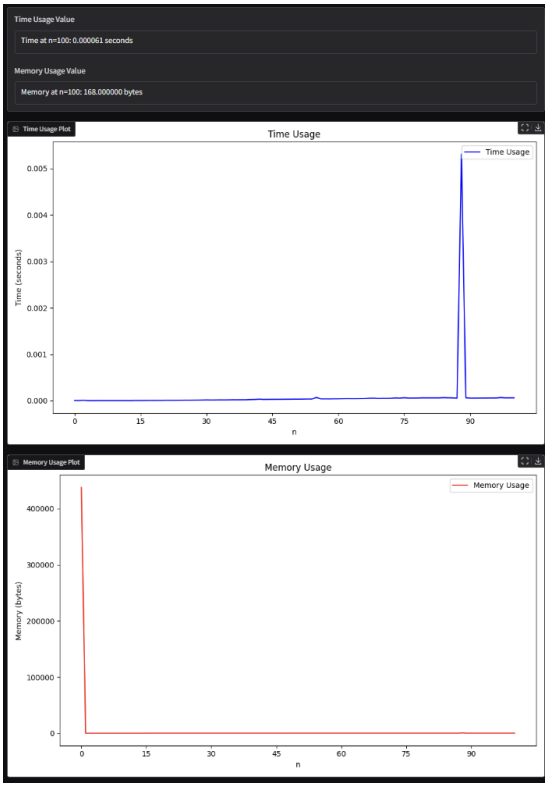

Figure 16: Space Optimized Implementation Algorithm results for n=100

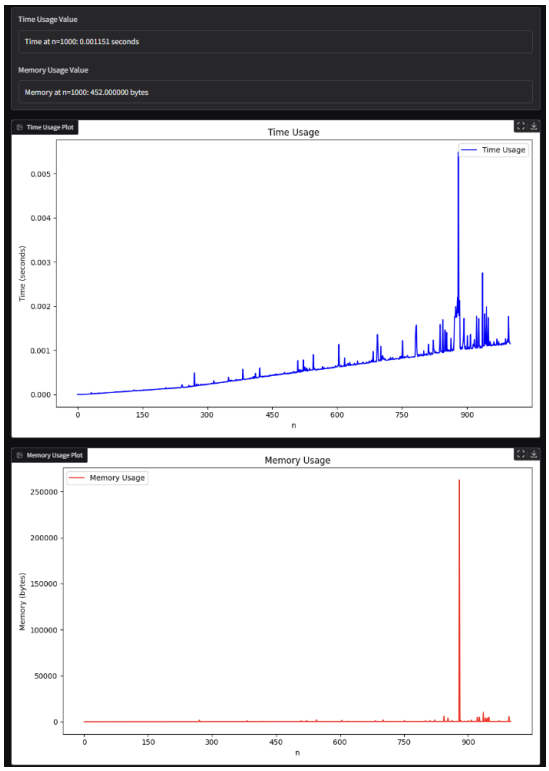

Figure 17: Space Optimized Implementation Algorithm results for n=1000

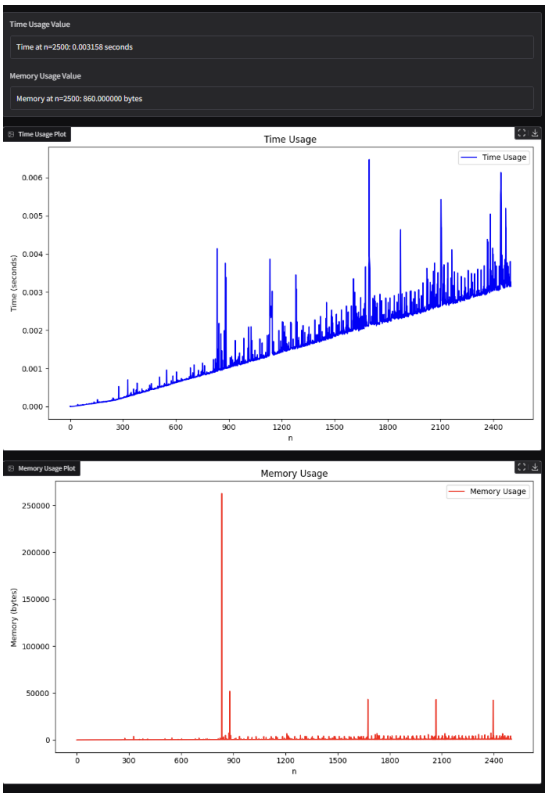

Figure 18: Space Optimized Implementation Algorithm results for n=2500

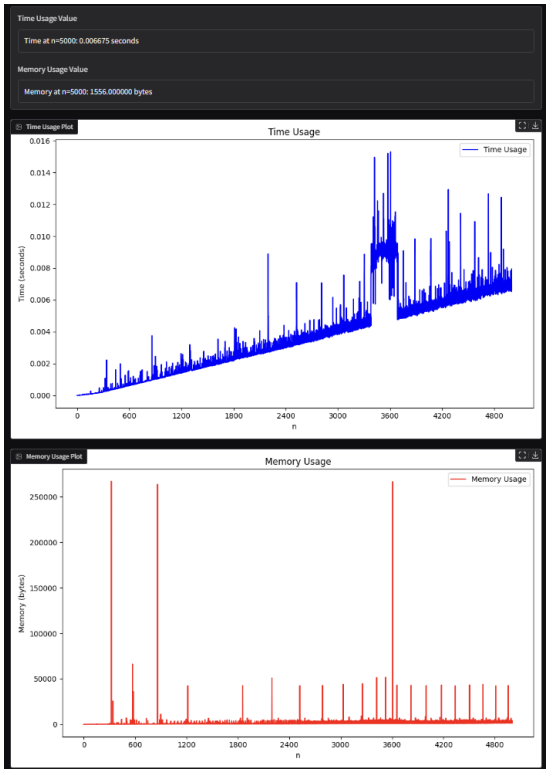

Figure 19: Space Optimized Implementation Algorithm results for n=5000

In the provided figures, the time usage of the Fibonacci computation using the space-optimized approach grows linearly, with the time taken increasing steadily as n increases. This linear growth is expected, as each Fibonacci number is calculated once in a loop, with each iteration requiring only constant time. This results in $O(n)$ time complexity.

The memory complexity, however, is significantly reduced compared to other approaches. The memory usage mostly remains constant, as only three variables are used to store the last two Fibonacci numbers at any given time. This leads to an $O(1)$ space complexity. Unlike other approaches that store all Fibonacci numbers, this space-optimized implementation only requires space proportional to three variables, ensuring minimal memory usage even as n grows.

6.6 Matrix Exponentiation Algorithm

A Matrix Exponentiation approach can be employed to solve the Fibonacci number problem by raising a base matrix to the power n using the divide-and-conquer method. This approach reduces the time complexity to $O(\log n)$, making it much more efficient for large n than the recursive approach.

In this approach, the base case is handled directly, and the matrix exponentiation function is used to compute the matrix raised to the power $n-1$. The Fibonacci number $F(n)$ is then extracted from the resulting matrix.

The Fibonacci sequence can be expressed with the following matrix:

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

```
def raise_power(mat, n):
  if n == 1:
    return mat
  elif n % 2 == 0:
    half_power = raise_power(mat, n // 2)
    return np.dot(half_power, half_power)
  else:
    half_power = raise_power(mat, (n - 1) // 2)
    return np.dot(np.dot(half_power, half_power), mat)

def fibonacci_matrix_exponentiation(n):
  if n == 0 or n == 1:
    return n
```

```
    else:
        mat = np.array([[1, 1], [1, 0]])
        result = raise_power(mat, n - 1)
    return result[0][0]
```

The complexity of the algorithm is as follows:

Time complexity: $O(\log n)$

Space complexity: $O(\log n)$

The second sequence will be used to compute the results for time and space usage.

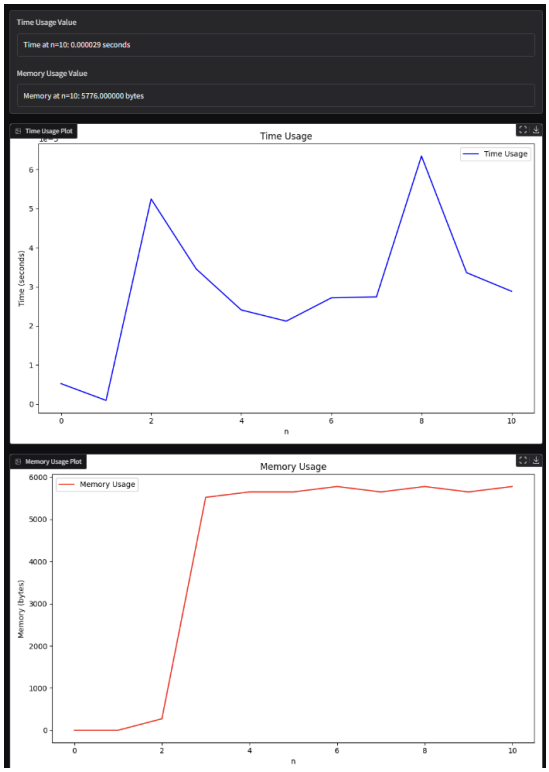

Figure 20: Matrix Exponentiation Algorithm results for n=10

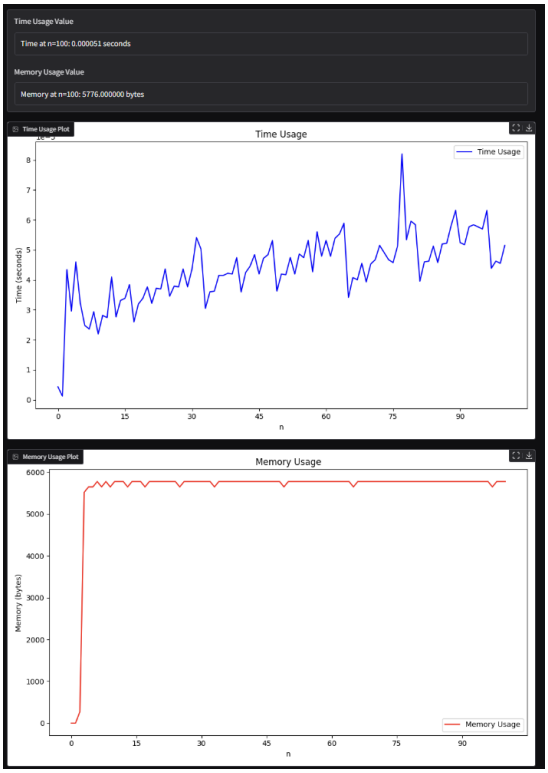

Figure 21: Matrix Exponentiation Algorithm results for n=100

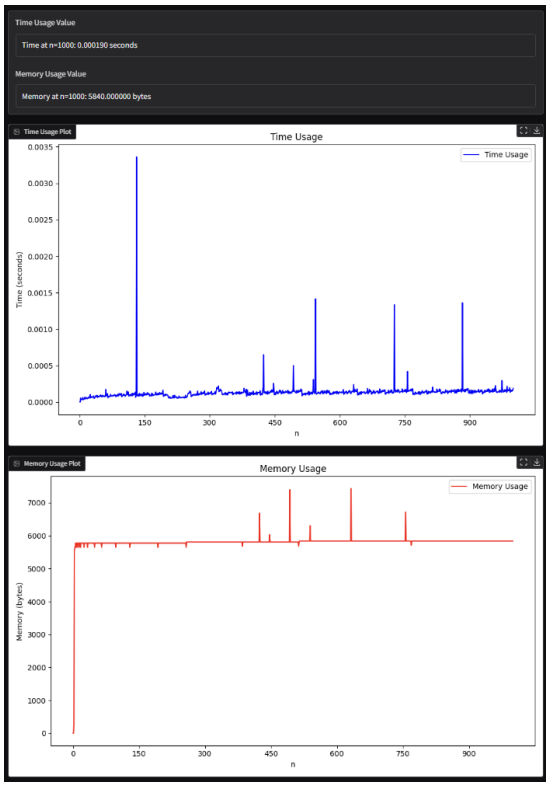

Figure 22: Matrix Exponentiation Algorithm results for n=1000

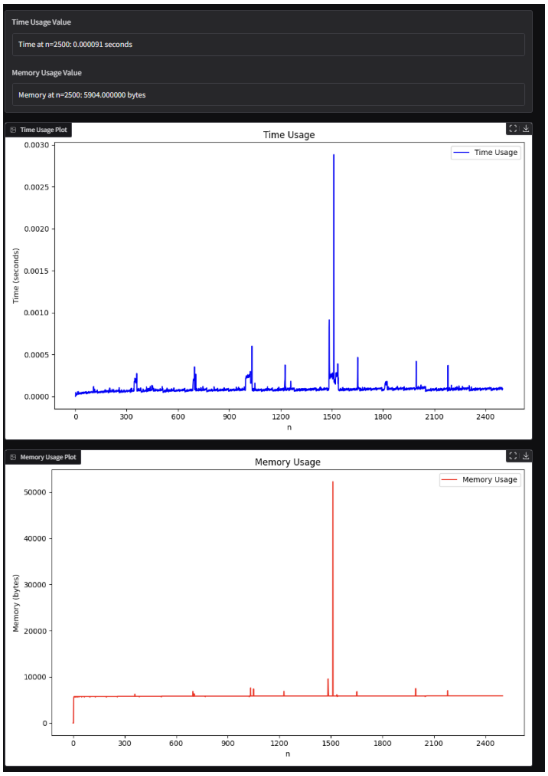

Figure 23: Matrix Exponentiation Algorithm results for n=2500

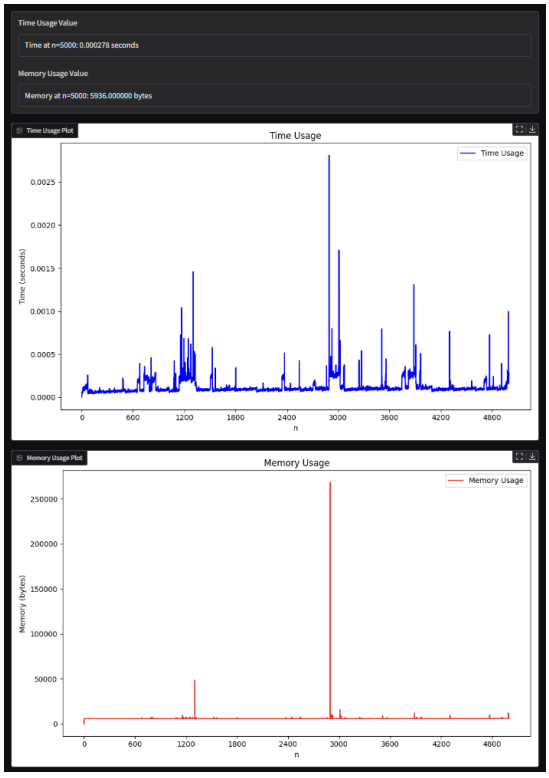

Figure 24: Matrix Exponentiation Algorithm results for n=5000

In the provided figures, the time usage of the Fibonacci computation using matrix exponentiation grows logarithmically with respect to n. This logarithmic growth is expected because the algorithm reduces the problem size by half at each step, resulting in $O(\log n)$ time complexity. The core of the algorithm relies on exponentiating a transformation matrix using a divide-and-conquer approach to compute higher powers efficiently.

The space complexity is $O(\log n)$, as the recursion depth is proportional to the logarithm of n due to the repeated halving of the problem size in each recursive call. Unlike iterative approaches, this method uses a recursion stack that grows with the logarithm of n, leading to slightly higher memory usage compared to iterative methods. However, the overall memory usage remains efficient compared to naive recursive implementations.

6.7 Golden Ratio Algorithm

The Golden Ratio approach can be employed to solve the Fibonacci number problem using the mathematical constant ϕ, also known as the Golden Ratio. This approach leverages the fact that the ratio of successive Fibonacci numbers approaches $\phi = \frac{1+\sqrt{5}}{2}$ as n increases. This allows for an efficient computation of Fibonacci numbers using multiplication and rounding operations.

In this approach, the base cases are precomputed and stored in a list. For $n < 6$, the function directly returns the corresponding Fibonacci number from the sequence. For $n \geq 6$, the function iteratively multiplies the previous Fibonacci value by ϕ and rounds the result to obtain the next Fibonacci number. This process continues until the desired Fibonacci number $F(n)$ is computed.

The Fibonacci number $F(n)$ can be approximated for large n using the following formula:

$$F(n) \approx F(n-1) * \phi$$

```
# Constants
phi = (1 + math.sqrt(5)) / 2
sequence = [0, 1, 1, 2, 3, 5]

def fibonacci_golden_ratio(n):
  if n < 6:
    return sequence[n]
  else:
    indx = 5
    value = sequence[indx]
    while indx < n:
      value = round(value * phi)
```

```
        indx += 1
    return value
```

The complexity of the algorithm is as follows:

Time complexity: $O(n)$

Space complexity: $O(1)$

The second sequence will be used to compute the results for time and space usage.

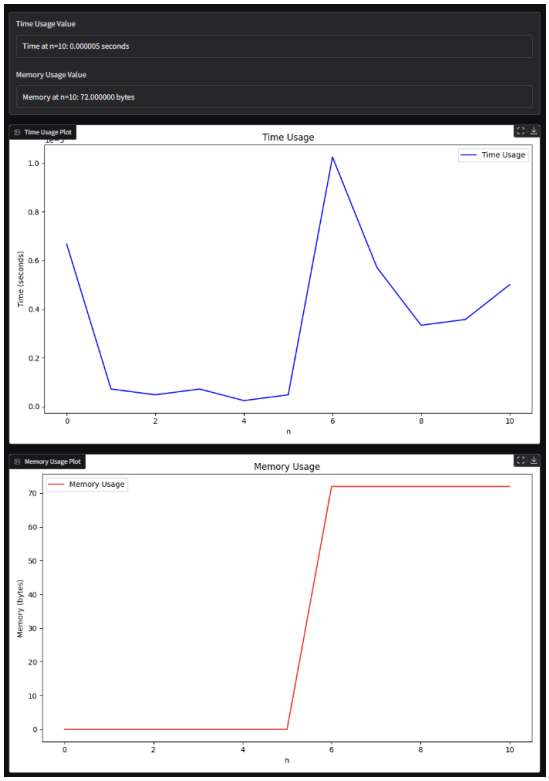

Figure 25: Golden Ratio Algorithm results for n=10

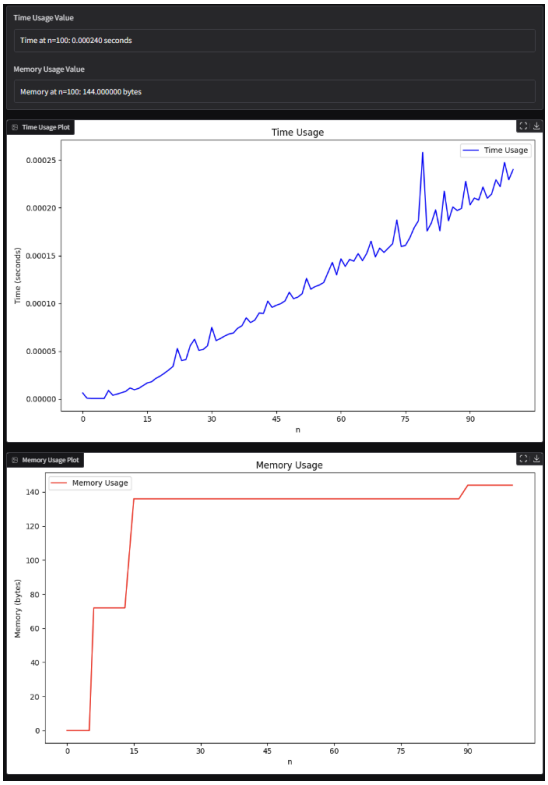

Figure 26: Golden Ratio Algorithm results for n=100

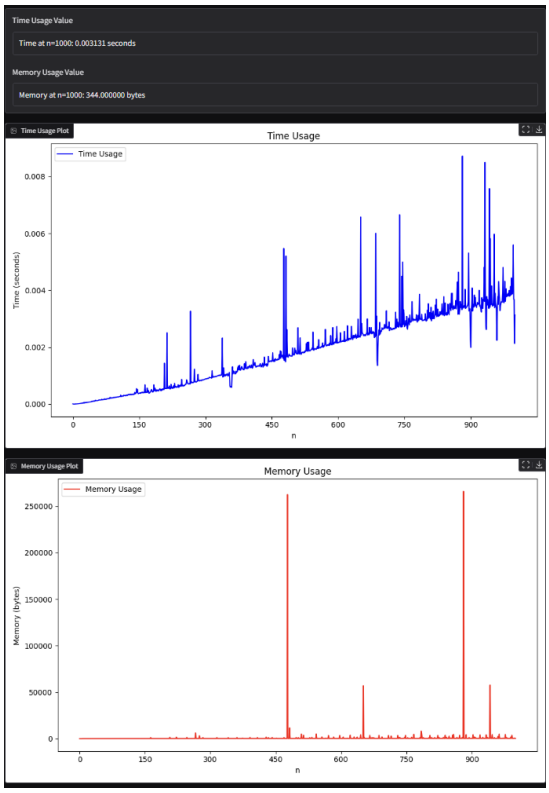

Figure 27: Golden Ratio Algorithm results for n=1000

In the provided figures, the time usage of the Fibonacci computation using the golden ratio method grows linearly with respect to n. This linear growth is expected because the algorithm iterates from a precomputed value up to n, calculating each subsequent Fibonacci number using a multiplication by the golden ratio $\phi = \frac{1+\sqrt{5}}{2}$. The time complexity is $O(n)$, as each Fibonacci number is computed in constant time per iteration.

The memory complexity is constant, or $O(1)$, as the algorithm only uses a fixed number of variables, including the golden ratio constant, the index variable, and the current Fibonacci value. No additional storage for intermediate values or recursion stacks is needed. But still, for $n = 2500$ and $n = 5000$, the values were not computed due to an encountered OverflowError: `cannot convert float infinity to integer`. The `math` library interprets some of the values as infinity, and an attempt to convert this infinite float into an integer triggers the overflow. The issue lies in the `math` library's inability to handle such large values.

6.8 Fast Doubling Algorithm

The Fast Doubling Algorithm is an efficient method for computing Fibonacci numbers using a divide-and-conquer strategy. This algorithm leverages the mathematical identities for Fibonacci numbers, which allow the calculation of $F(2k)$ and $F(2k+1)$ from $F(k)$

and $F(k+1)$ in constant time:

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

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

Using these identities, the algorithm recursively splits the problem into smaller subproblems by halving n, computing the Fibonacci numbers for each half, and combining the results efficiently. This reduces the time complexity to $O(\log n)$.

For even n, the algorithm returns $F(2k)$ and $F(2k+1)$. For odd n, it returns $F(2k+1)$ and $F(2k+2)$.

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

def fibonacci_fast_doubling_helper(n):
  return fibonacci_fast_doubling(n)[0]
```

The complexity of the algorithm is as follows:

Time complexity: $O(\log n)$

Space complexity: $O(\log n)$

The second sequence will be used to compute the results for time and space usage.

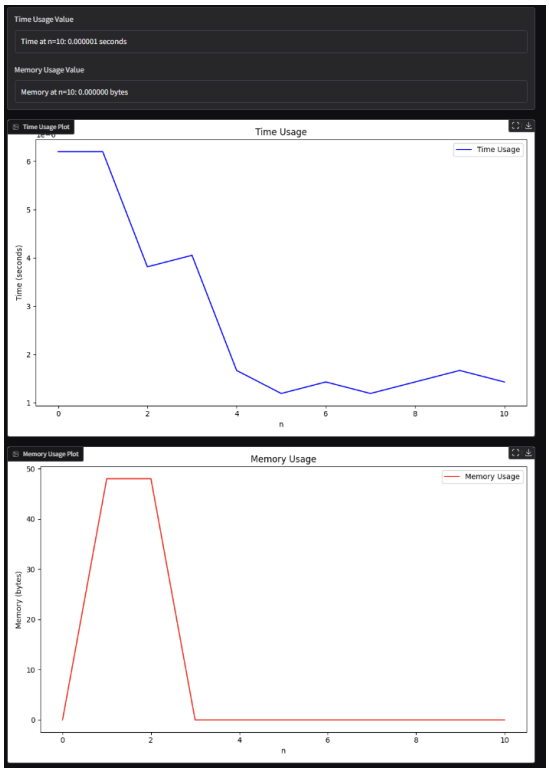

Figure 28: Fast Doubling Algorithm results for n=10

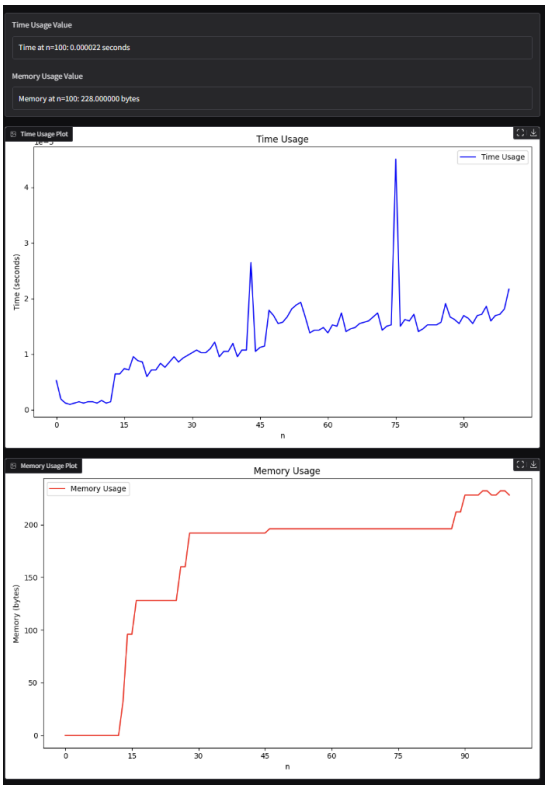

Figure 29: Fast Doubling Algorithm results for n=100

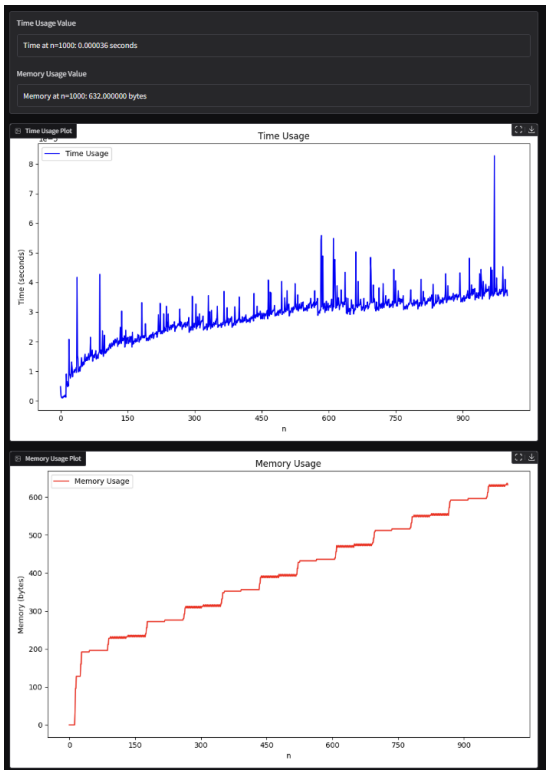

Figure 30: Fast Doubling Algorithm results for n=1000

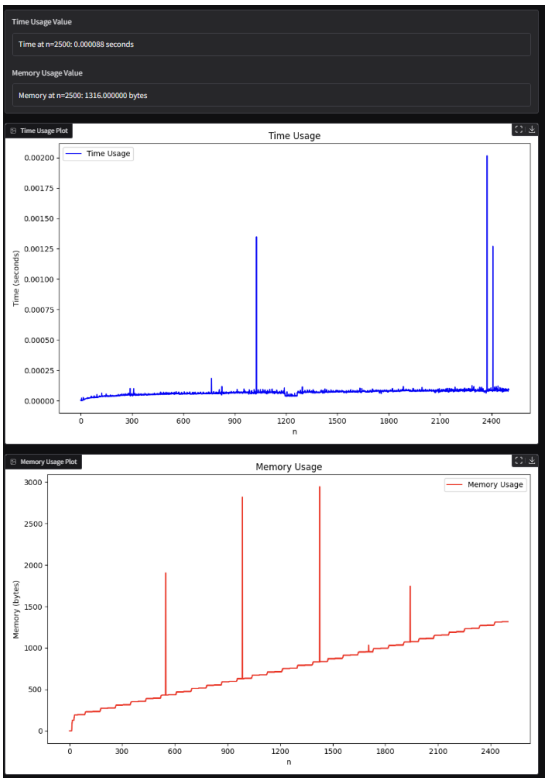

Figure 31: Fast Doubling Algorithm results for n=2500

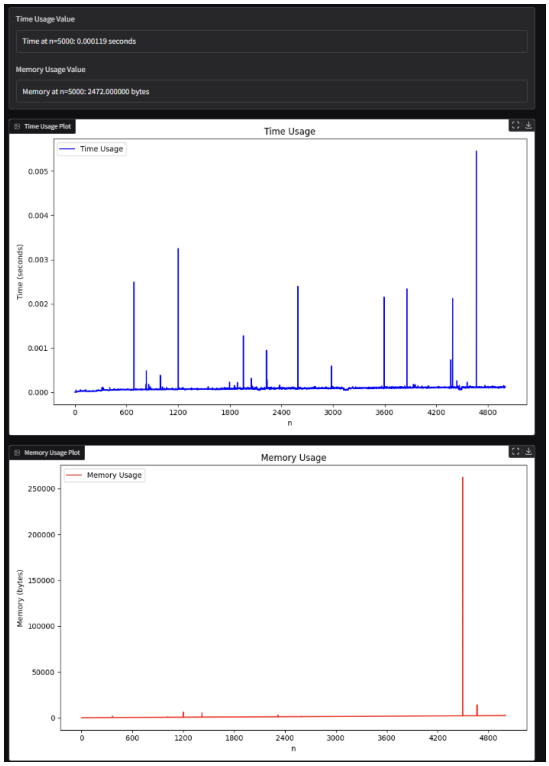

Figure 32: Fast Doubling Algorithm results for n=5000

In the provided figures, the time usage of the Fibonacci computation using the fast doubling method grows logarithmically with respect to n. This logarithmic growth is achieved through a divide-and-conquer approach, where the Fibonacci number at position n is computed by recursively halving the problem size. This results in a time complexity of $O(\log n)$, as each recursive call reduces the problem size by half.

The space complexity is $O(\log n)$ due to the recursion stack depth, which is proportional to the logarithm of n. At each step, intermediate Fibonacci numbers are computed using the formulas:

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

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

This approach efficiently computes Fibonacci numbers without the need to store all intermediate values, resulting in minimal memory usage compared to naive recursive or iterative methods. The fast doubling method is highly efficient for calculating large Fibonacci numbers due to its logarithmic time complexity and space efficiency.

6.9 Binet's Approximation Algorithm

Binet's Approximation Algorithm is a closed-form expression for approximating the n-th Fibonacci number. The algorithm uses the following formula:

$$F(n) \approx \left\lfloor \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n + 0.5 \right\rfloor$$

Where $\phi = \frac{1+\sqrt{5}}{2}$ is the Golden Ratio.

```
def fibonacci_binets_approximation(n):
  if n == 0 or n == 1:
    return n
  else:
    phi = (1 + math.sqrt(5)) / 2
    return math.floor(math.pow(phi, n) / math.sqrt(5) + 0.5)
```

The complexity of the algorithm is as follows:

Time complexity: $O(\log n)$

Space complexity: $O(1)$

The second sequence will be used to compute the results for time and space usage.

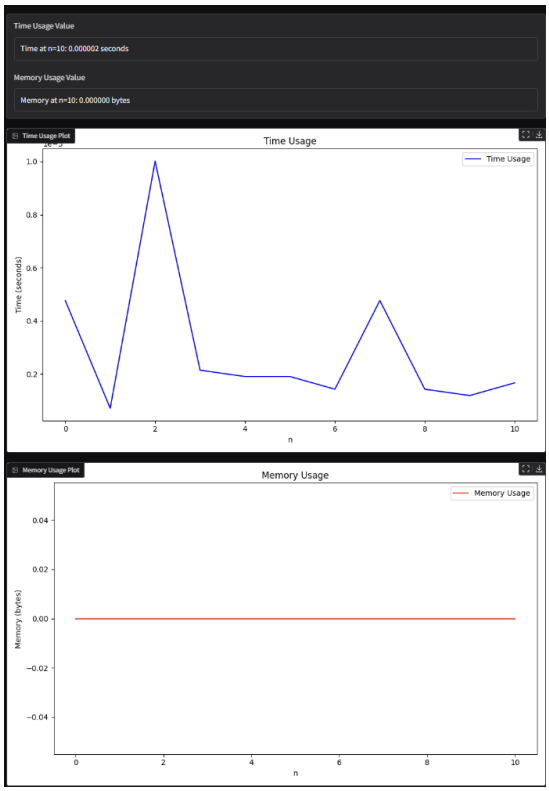

Figure 33: Golden Ratio Algorithm results for n=10

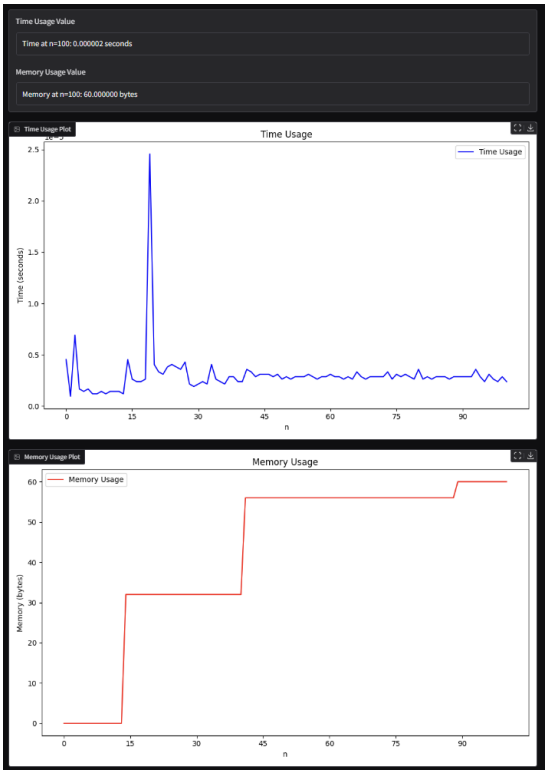

Figure 34: Golden Ratio Algorithm results for n=100

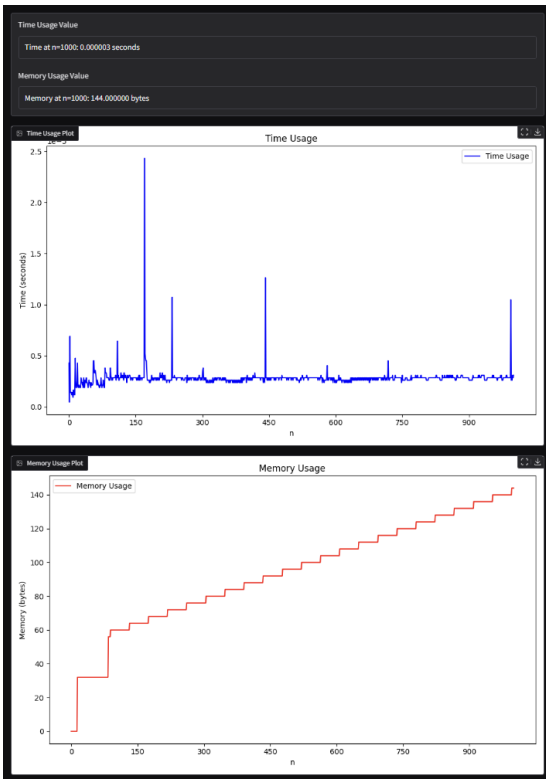

Figure 35: Golden Ratio Algorithm results for n=1000

In the provided figures, the time usage of the Fibonacci computation using Binet's approximation grows logarithmically with respect to n. This logarithmic growth is a result of the use of exponentiation by squaring to compute powers of the golden ratio $\phi = \frac{1+\sqrt{5}}{2}$, which can be done in $O(\log n)$ time.

The memory complexity is constant, or $O(1)$, because the algorithm only uses a fixed number of variables, including the golden ratio constant and temporary variables for arithmetic operations. For n=2500 and n=5000, the values were not computed because of an encountered Overflow Error. The Overflow Error encountered when computing Binet's approximation for large Fibonacci numbers ($n = 2500$ and $n = 5000$) is due to the limitations of the `math` library in handling extremely large numbers. Specifically, the calculation of ϕ^n exceeds the floating-point range, triggering the overflow. The issue lies in the inability of the `math` library to manage such large values.

7 Conclusion

In this paper, several algorithms for computing the n-th Fibonacci term were explored, each with its own advantages, limitations, and performance characteristics.

The naive recursive method, though conceptually straightforward, struggled with both efficiency and overflow for larger inputs due to its exponential time complexity and redundant calculations. This reinforced how brute-force approaches can be unsuitable for computationally intensive tasks.

Dynamic programming methods — Bottom-Up and Top-Down Approaches — addressed the inefficiencies by storing previously computed values, greatly improving runtime. These methods were reliable for moderate input sizes but still susceptible to overflow for very large n without appropriate data types.

The iterative space-optimized approach not only reduced the time complexity to linear but also minimized memory usage to constant space. This efficiency made it practical for larger inputs.

Matrix exponentiation proved to be one of the most efficient algorithms with its logarithmic time complexity. It leveraged mathematical properties for rapid computation.

Binet's formula and the closed-form golden ratio approximation offered an elegant, near-instant computation. However, these methods were limited by the loss of precision and overflow error.

Overall, the choice of algorithm should be guided by the trade-offs between speed, memory usage, numerical stability, and overflow considerations. Future research could explore strategies for avoiding overflow and hybrid algorithms that dynamically select the most efficient approach based on input size.