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

Laboratory work 1:

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

Elaborated:
st. gr. FAF-232 Ba

Verified:

asist. univ. Fiștic Cristofor

Chişinău - 2025

TABLE OF CONTENTS

ALGORITHM ANALYSIS

Objective

Learn and examine various methods for calculating the nth term in the Fibonacci sequence.

Tasks:

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

Theoretical Notes:

Instead of using mathematical methods to analyze complexity, we can use empirical analysis. This approach can be helpful for:

- Getting initial insights into the complexity level of an algorithm.
- Comparing the performance of two or more algorithms that solve the same problem.
- Comparing different versions of the same algorithm.
- Understanding how well an algorithm works on a specific computer.

In empirical analysis, the following steps are usually taken:

1. Define the goal of the analysis.

2. Choose a way to measure efficiency, such as counting how many times a specific operation is performed or measuring the time it takes to run part or all of the algorithm.

3. Decide on the input data properties to analyze, such as the size of the data or specific characteristics.

4. Implement the algorithm in a programming language.

5. Create multiple sets of input data for testing.

6. Run the program for each set of input data.

7. Analyze the results obtained from the tests.

The choice of efficiency measure depends on the goal:

- If the goal is to understand the complexity class or check a theoretical estimate, it’s better to count the number of operations performed.
- If the goal is to see how well the algorithm works in practice, measuring execution time is more suitable.

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: xn= xn-1 + xn-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).

Within this laboratory, we will be analyzing the 4 naïve algorithms empirically.

Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n))

Input Format:

As input, each algorithm will receive two series of numbers that will contain the order of the Fibonacci terms being looked up. The first series will have a more limited scope, (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 six algorithms will be implemented in their naïve form in python and analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending on memory of the device used. The error margin determined will constitute 2.5 seconds as per experimental measurement.

The code can be found here: https://github.com/AlexandraB-C/AA_labs/blob/main/fibonacci_methods.py

Recursive Method:

The recursive approach follows the natural definition of Fibonacci numbers but suffers from severe inefficiencies. It repeatedly recalculates values, leading to exponential time complexity.

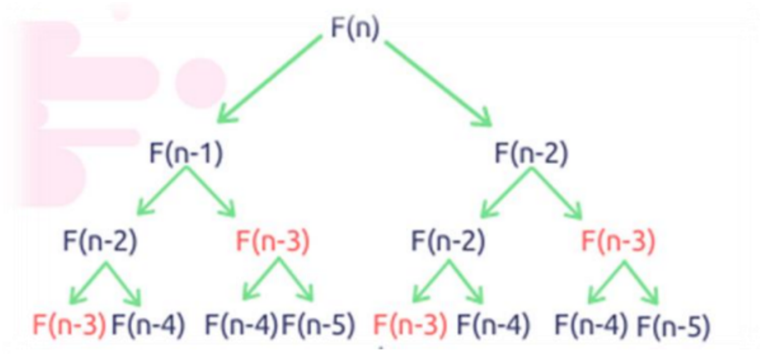


Figure 1 Fibonacci Recursion

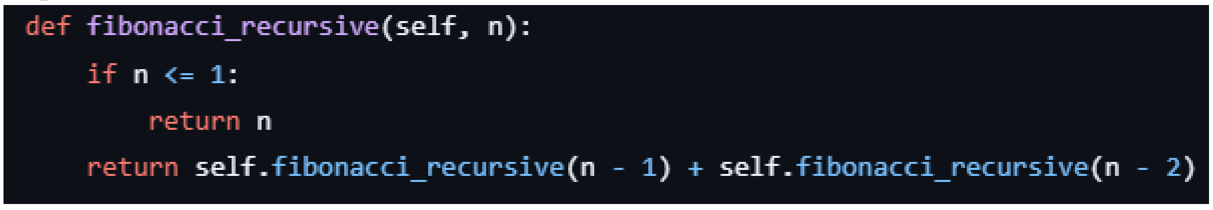
Algorithm Description:

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)
```

However, due to repeated computations of the same values, this approach has an exponential time complexity $O(2^{n})$, making it impractical for large values of *n*.

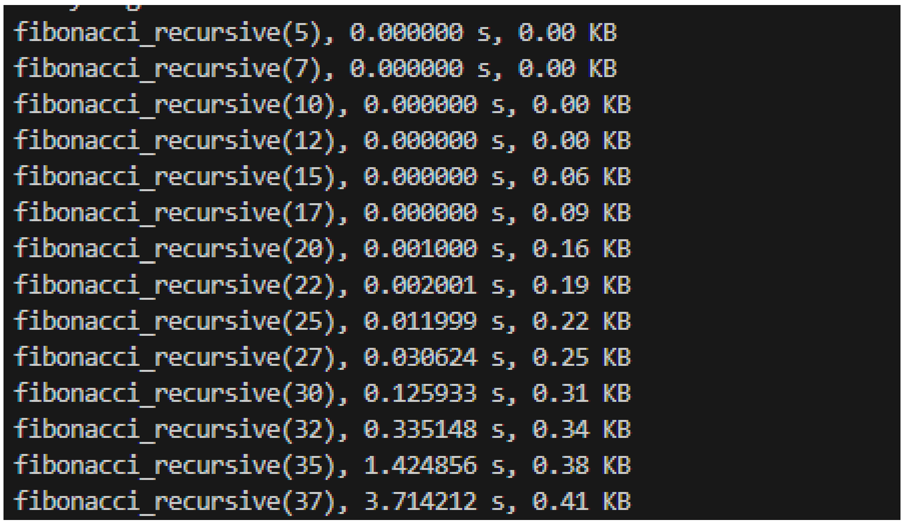
Implementation:



```
def fibonacci_recursive(self, n):
    if n <= 1:
        return n
    return self.fibonacci_recursive(n - 1) + self.fibonacci_recursive(n - 2)
```

Figure 2 Fibonacci recursion in Python

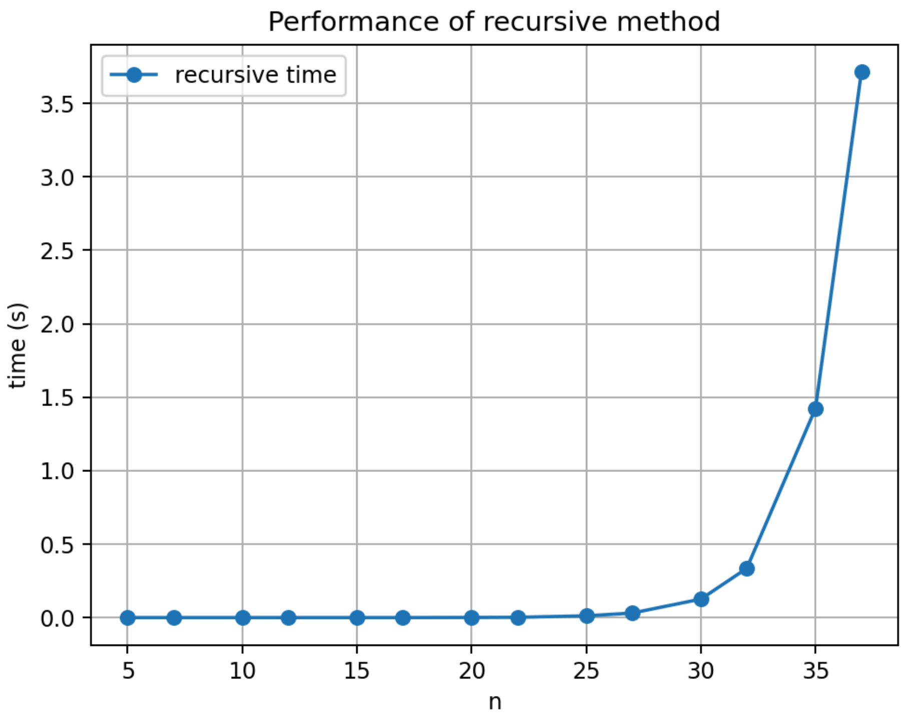
Results:



```
fibonacci_recursive(5), 0.000000 s, 0.00 KB
fibonacci_recursive(7), 0.000000 s, 0.00 KB
fibonacci_recursive(10), 0.000000 s, 0.00 KB
fibonacci_recursive(12), 0.000000 s, 0.00 KB
fibonacci_recursive(15), 0.000000 s, 0.06 KB
fibonacci_recursive(17), 0.000000 s, 0.09 KB
fibonacci_recursive(20), 0.001000 s, 0.16 KB
fibonacci_recursive(22), 0.002001 s, 0.19 KB
fibonacci_recursive(25), 0.011999 s, 0.22 KB
fibonacci_recursive(27), 0.030624 s, 0.25 KB
fibonacci_recursive(30), 0.125933 s, 0.31 KB
fibonacci_recursive(32), 0.335148 s, 0.34 KB
fibonacci_recursive(35), 1.424856 s, 0.38 KB
fibonacci_recursive(37), 3.714212 s, 0.41 KB
```

Figure 3 Results for first set of inputs

The recursive approach is extremely slow, as it redundantly computes the same Fibonacci values multiple times. It performs well for very small *n* but quickly becomes not good for larger values.



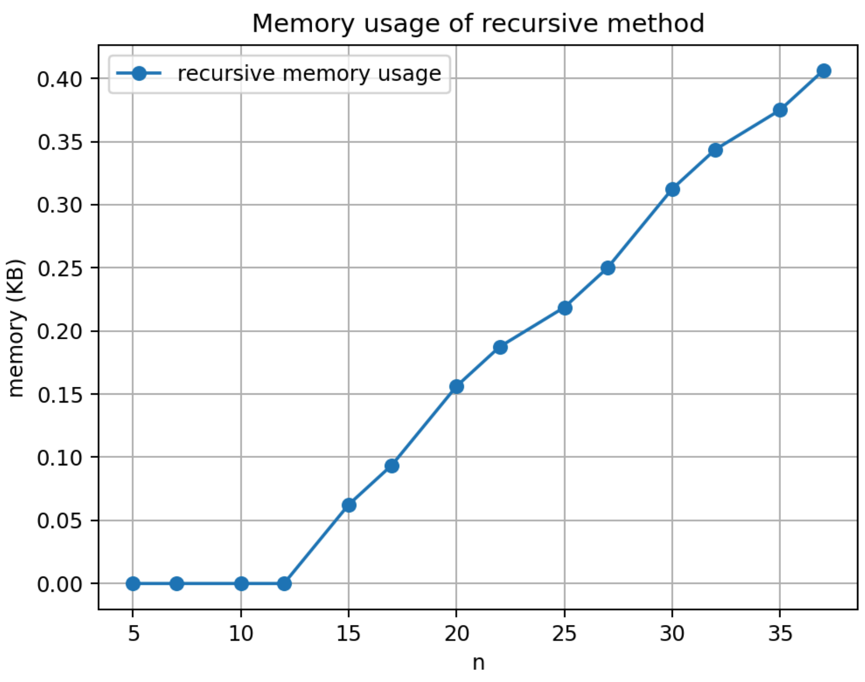


Figure 4 Graph of Recursive Fibonacci Function

The execution time grows exponentially, confirming the $O(2^n)$ complexity. The memory usage is also high due to the recursive call stack depth.

Memoization Method:

Memoization is an optimization technique that stores previously computed results, preventing redundant calculations. This approach significantly improves efficiency while maintaining a recursive structure.

Algorithm Description:

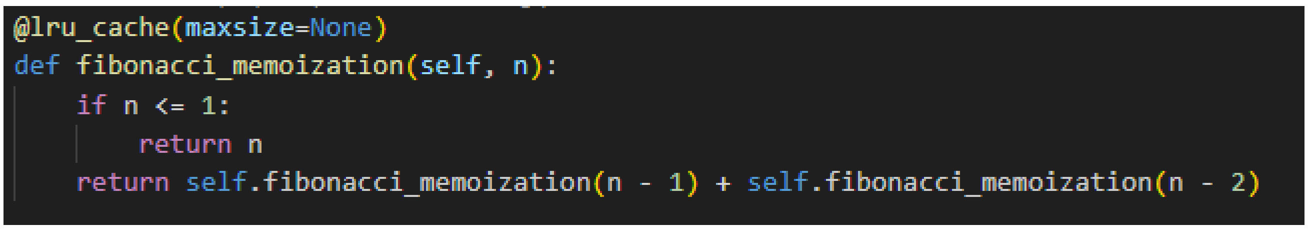
Instead of recomputing values, we store them in a dictionary (or cache) and reuse them when needed. This results in an **O(n) time complexity**, as each Fibonacci number is computed only once.

```
Fibonacci(n, memo):
  if n in memo:
    return memo[n]

  if n <= 1:
    return n

  memo[n] <- Fibonacci(n-1, memo) + Fibonacci(n-2, memo)
  return memo[n]
```

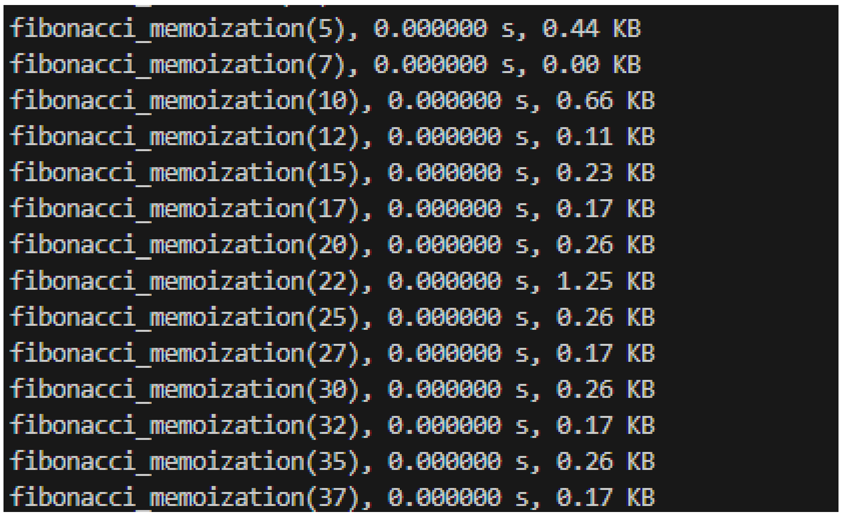
Implementation:



```python
@lru_cache(maxsize=None)
def fibonacci_memoization(self, n):
    if n <= 1:
        return n
    return self.fibonacci_memoization(n - 1) + self.fibonacci_memoization(n - 2)
```

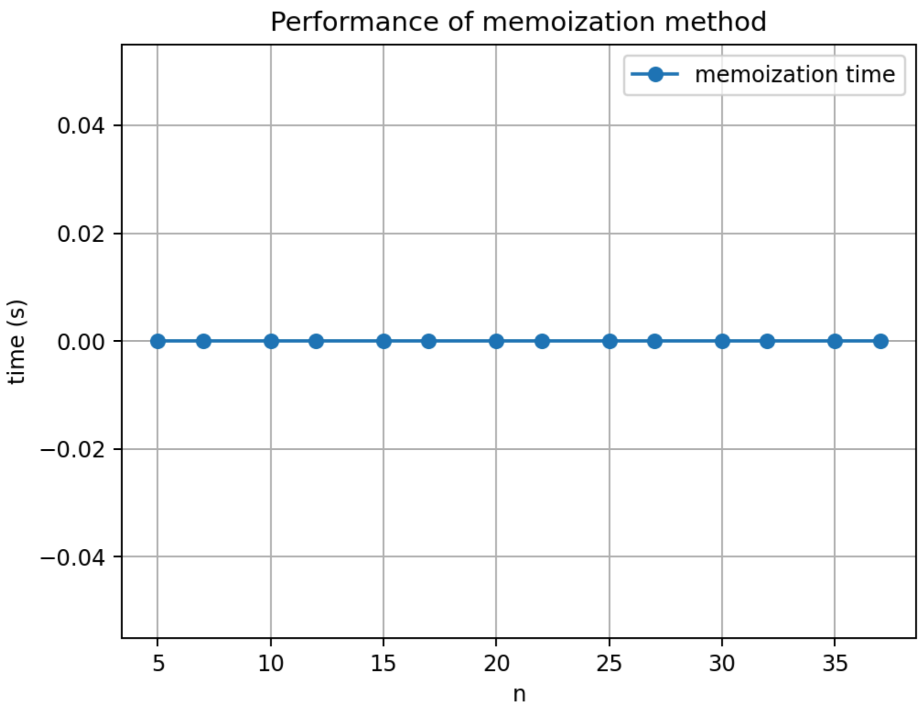
Figure 5 Fibonacci Memoization in Python

Results:



```
fibonacci_memoization(5), 0.000000 s, 0.44 KB
fibonacci_memoization(7), 0.000000 s, 0.00 KB
fibonacci_memoization(10), 0.000000 s, 0.66 KB
fibonacci_memoization(12), 0.000000 s, 0.11 KB
fibonacci_memoization(15), 0.000000 s, 0.23 KB
fibonacci_memoization(17), 0.000000 s, 0.17 KB
fibonacci_memoization(20), 0.000000 s, 0.26 KB
fibonacci_memoization(22), 0.000000 s, 1.25 KB
fibonacci_memoization(25), 0.000000 s, 0.26 KB
fibonacci_memoization(27), 0.000000 s, 0.17 KB
fibonacci_memoization(30), 0.000000 s, 0.26 KB
fibonacci_memoization(32), 0.000000 s, 0.17 KB
fibonacci_memoization(35), 0.000000 s, 0.26 KB
fibonacci_memoization(37), 0.000000 s, 0.17 KB
```

Figure 6 Results for first set of inputs



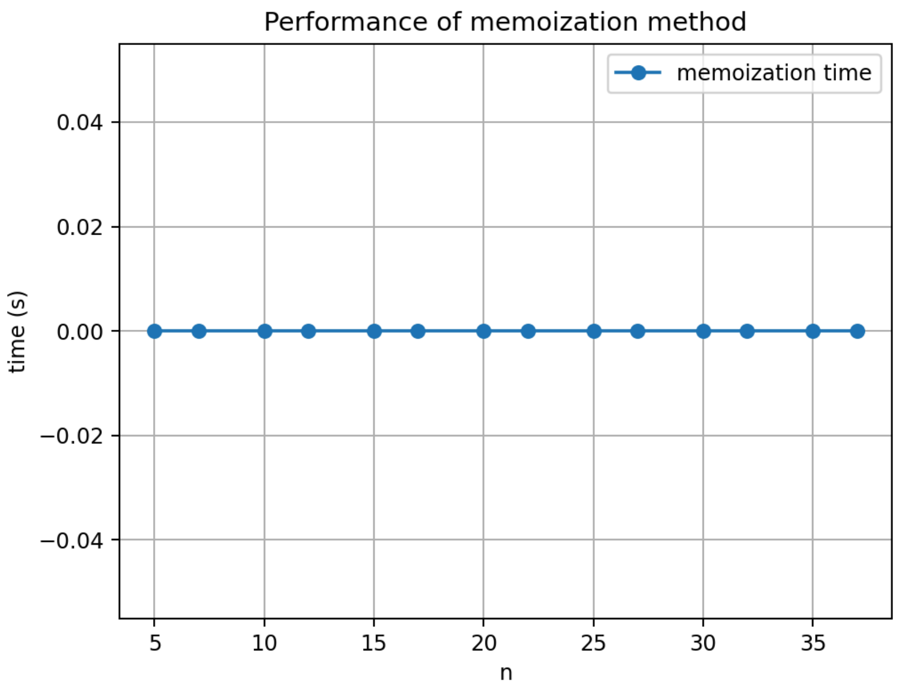


Figure 7 Graph of Memoization Fibonacci Function

Performance: Much faster than naive recursion but slightly slower than iterative methods due to function call overhead.

Memory Usage: O(n) due to the storage of computed values.

Graph Analysis: Linear time complexity, significantly outperforming naive recursion.

Dynamic Programming Method:

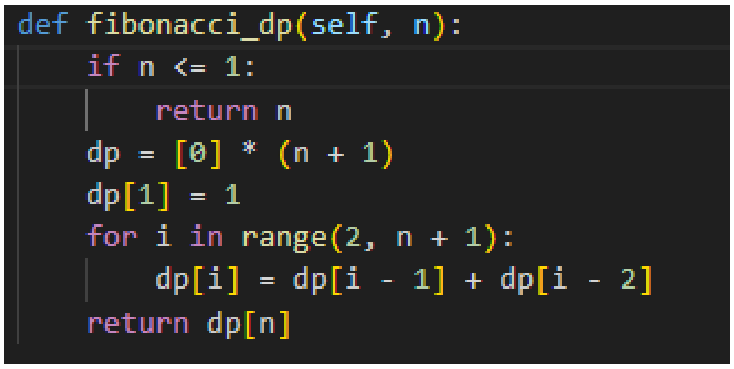
The Dynamic Programming method, similar to the recursive method, takes the straightforward approach of calculating the n-th term. However, instead of calling the function upon itself, from top down it operates based on an array data structure that holds the previously computed terms, eliminating the need to recompute them. So, shortly: The dynamic programming approach eliminates recursion, instead using an array to compute Fibonacci numbers iteratively.

Algorithm Description:

The sequence is built from the ground up, storing previous results in an array to prevent redundant computations. This reduces time complexity to O(n) and avoids the function call overhead present in recursion.

```
Fibonacci(n):
  if n <= 1:
    return n
  dp = array of size n+1
  dp[0] = 0, dp[1] = 1
  for i = 2 to n:
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

Implementation:



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

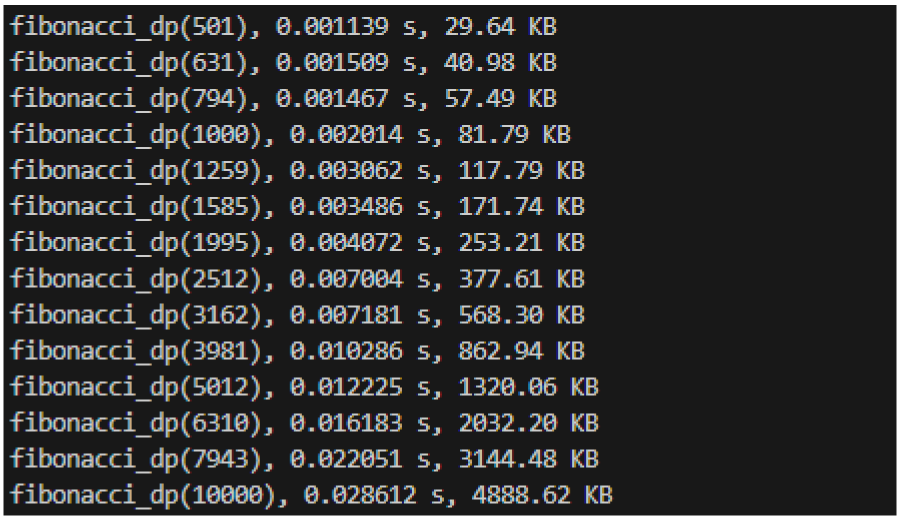
Figure 8 Fibonacci DP in Python

Results:

Performance: Linear time complexity, significantly better than recursion.

Memory Usage: O(n) due to storing all Fibonacci numbers.

Graph Analysis: Linear increase in execution time as nnn grows.



```
fibonacci_dp(501), 0.001139 s, 29.64 KB
fibonacci_dp(631), 0.001509 s, 40.98 KB
fibonacci_dp(794), 0.001467 s, 57.49 KB
fibonacci_dp(1000), 0.002014 s, 81.79 KB
fibonacci_dp(1259), 0.003062 s, 117.79 KB
fibonacci_dp(1585), 0.003486 s, 171.74 KB
fibonacci_dp(1995), 0.004072 s, 253.21 KB
fibonacci_dp(2512), 0.007004 s, 377.61 KB
fibonacci_dp(3162), 0.007181 s, 568.30 KB
fibonacci_dp(3981), 0.010286 s, 862.94 KB
fibonacci_dp(5012), 0.012225 s, 1320.06 KB
fibonacci_dp(6310), 0.016183 s, 2032.20 KB
fibonacci_dp(7943), 0.022051 s, 3144.48 KB
fibonacci_dp(10000), 0.028612 s, 4888.62 KB
```

Figure 9 Fibonacci DP Result

With the Dynamic Programming Method showing excellent results with a time complexity denoted in a corresponding graph of O(n).

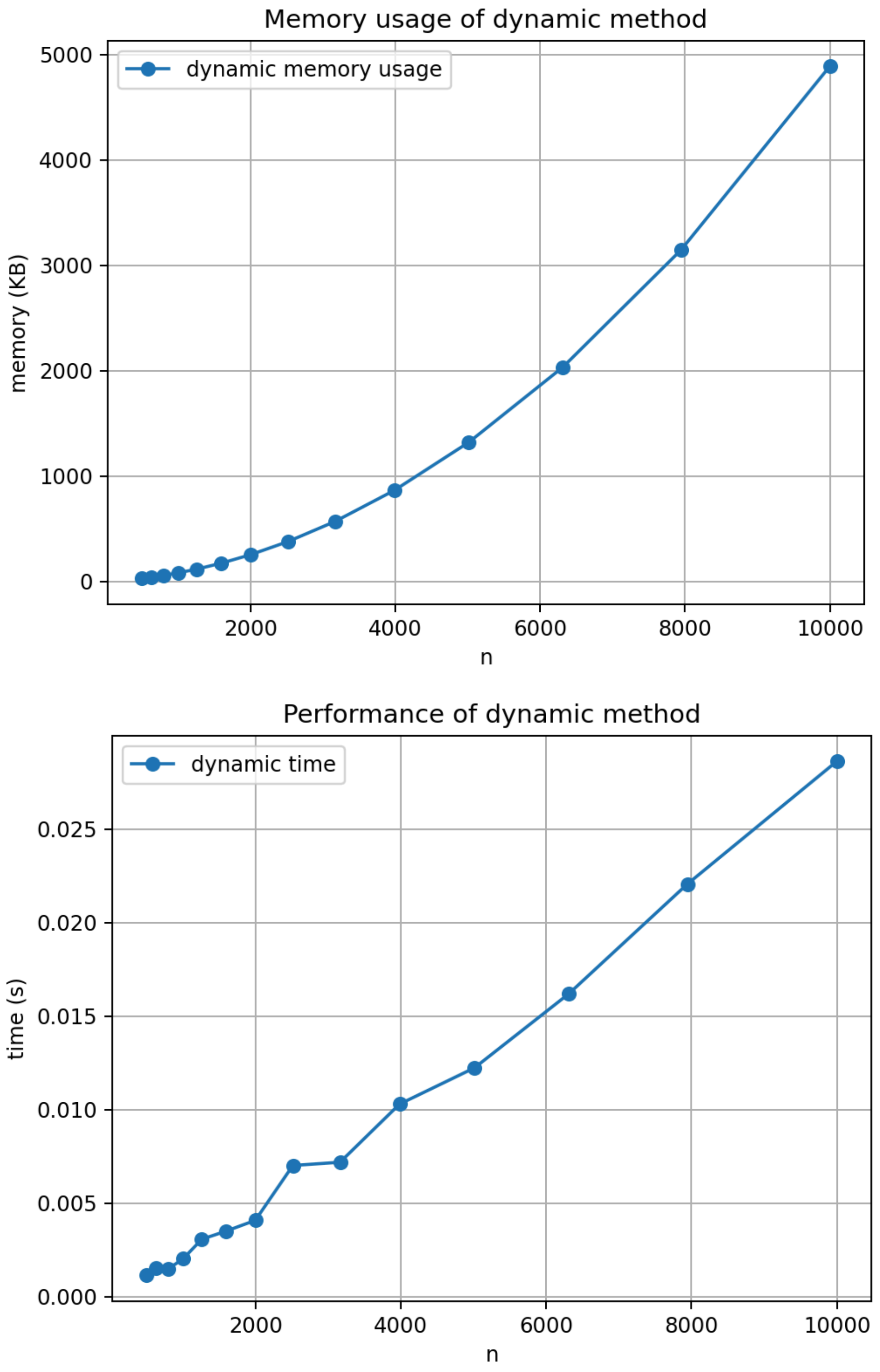


Figure 10 Fibonacci DP Graph

Iterative Method (Space-Optimized):

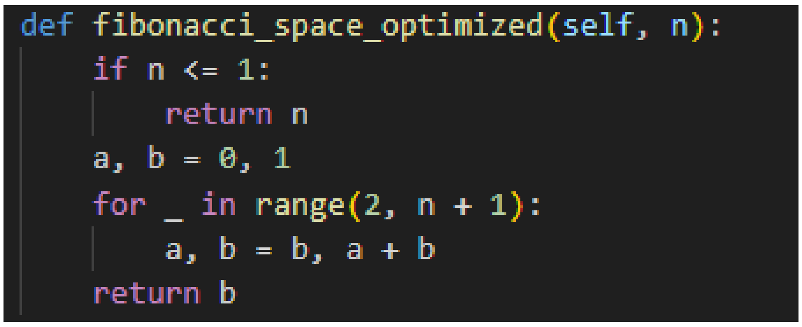
This method improves the standard DP approach by reducing space complexity. Instead of storing all computed values, it only keeps track of the last two Fibonacci numbers.

Algorithm Description:

This approach uses constant space regardless of the input size, making it highly memory-efficient for large n values. The algorithm iterates through Fibonacci numbers while updating only two variables, reducing space complexity to O(1) while maintaining O(n) time complexity.

```
Fibonacci(n):
  if n <= 1:
    return n
  a = 0, b = 1
  for i = 2 to n:
    temp = a + b
    a = b
    b = temp
  return b
```

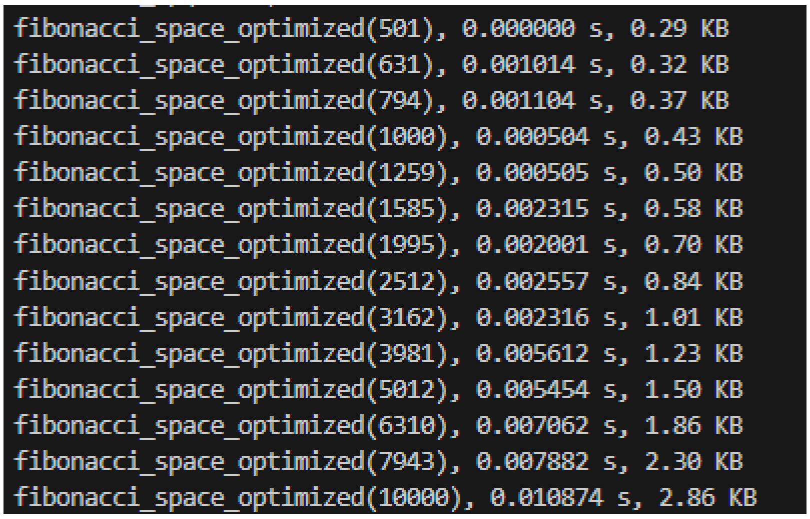
Implementation:



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

Figure 11 Fibonacci Iterative Method in Python

Results:

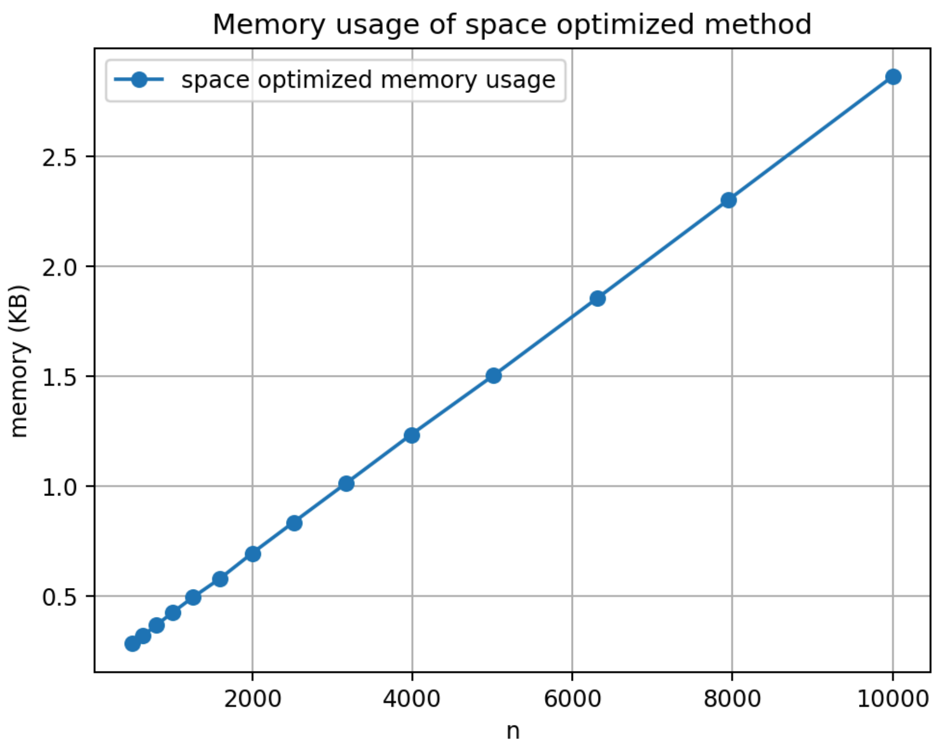


```
fibonacci_space_optimized(501), 0.000000 s, 0.29 KB
fibonacci_space_optimized(631), 0.001014 s, 0.32 KB
fibonacci_space_optimized(794), 0.001104 s, 0.37 KB
fibonacci_space_optimized(1000), 0.000504 s, 0.43 KB
fibonacci_space_optimized(1259), 0.000505 s, 0.50 KB
fibonacci_space_optimized(1585), 0.002315 s, 0.58 KB
fibonacci_space_optimized(1995), 0.002001 s, 0.70 KB
fibonacci_space_optimized(2512), 0.002557 s, 0.84 KB
fibonacci_space_optimized(3162), 0.002316 s, 1.01 KB
fibonacci_space_optimized(3981), 0.005612 s, 1.23 KB
fibonacci_space_optimized(5012), 0.005454 s, 1.50 KB
fibonacci_space_optimized(6310), 0.007062 s, 1.86 KB
fibonacci_space_optimized(7943), 0.007882 s, 2.30 KB
fibonacci_space_optimized(10000), 0.010874 s, 2.86 KB
```

Figure 12 Fibonacci Iterative Method Results

After executing the function for each n Fibonacci term mentioned in the second set of Input Format (larger numbers), we obtained the following results:

The iterative method shows excellent performance for moderate-sized inputs, with execution times growing linearly with n but remaining very manageable.



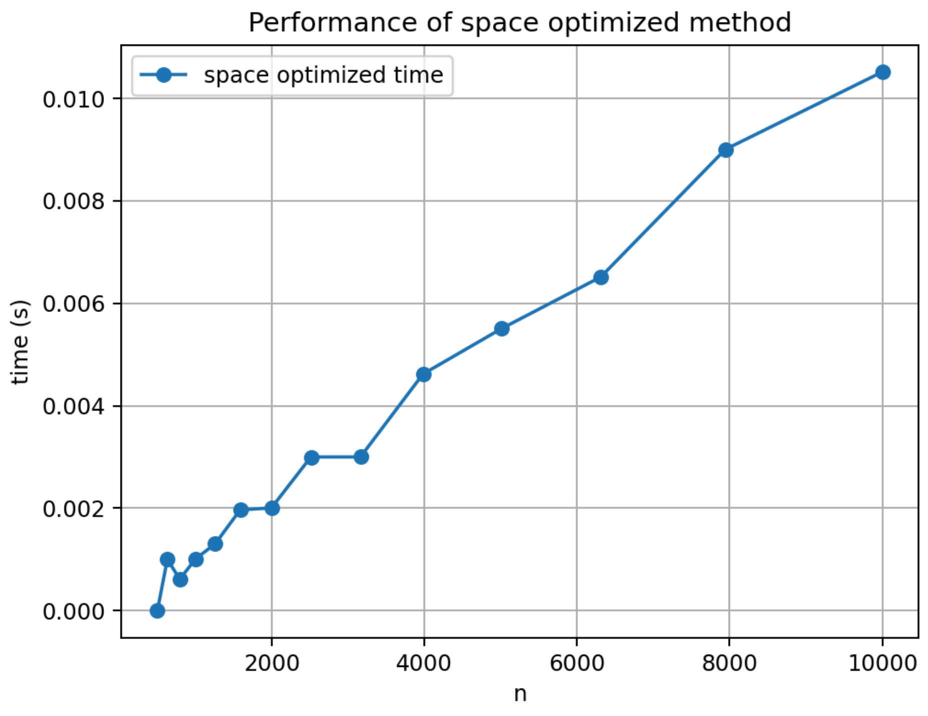


Figure 13 Fibonacci Iterative Method Graph

Performance: As fast as standard DP but uses significantly less memory.

Memory Usage: Minimal; only two variables are maintained (O(1)).

Graph Analysis: Execution time is nearly identical to DP but with lower memory consumption.

The graph shows a clear linear trend, confirming the O(n) time complexity. However, the constant factors are very small, making this method highly efficient in practice. The space complexity remains constant at O(1) regardless of input size.

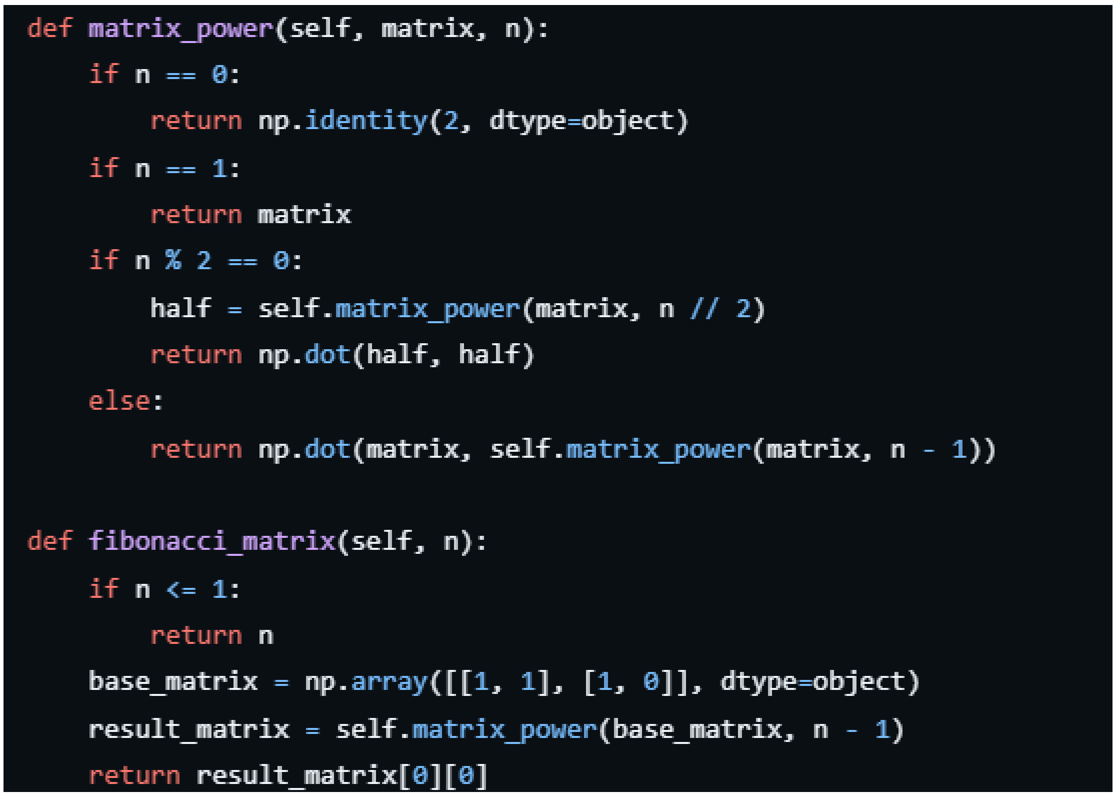
Matrix Power Method:

This approach uses matrix multiplication to compute Fibonacci numbers in O(log n) time, making it highly efficient for large inputs. By computing matrix powers efficiently via exponentiation by squaring, we achieve logarithmic time complexity.

```
MatrixPower(matrix, n):
  if n == 0:
    return identity_matrix
  if n == 1:
    return matrix
  if n is even:
    half = MatrixPower(matrix, n / 2)
    return half * half
  else:
    return matrix * MatrixPower(matrix, n - 1)

Fibonacci(n):
  base_matrix = [[1, 1], [1, 0]]
  result = MatrixPower(base_matrix, n - 1)
  return result[0][0]
```

Implementation:



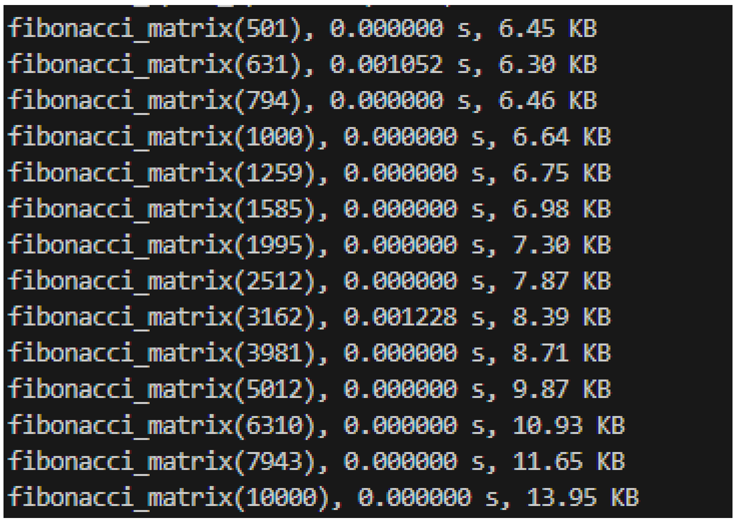
```python
def matrix_power(self, matrix, n):
    if n == 0:
        return np.identity(2, dtype=object)
    if n == 1:
        return matrix
    if n % 2 == 0:
        half = self.matrix_power(matrix, n // 2)
        return np.dot(half, half)
    else:
        return np.dot(matrix, self.matrix_power(matrix, n - 1))

def fibonacci_matrix(self, n):
    if n <= 1:
        return n
    base_matrix = np.array([[1, 1], [1, 0]], dtype=object)
    result_matrix = self.matrix_power(base_matrix, n - 1)
    return result_matrix[0][0]
```

Figure 14 Fibonacci Matrix Power Method in Python

Results:

After the execution of the function for each n Fibonacci term mentioned in the second set of Input Format we obtain the following results:



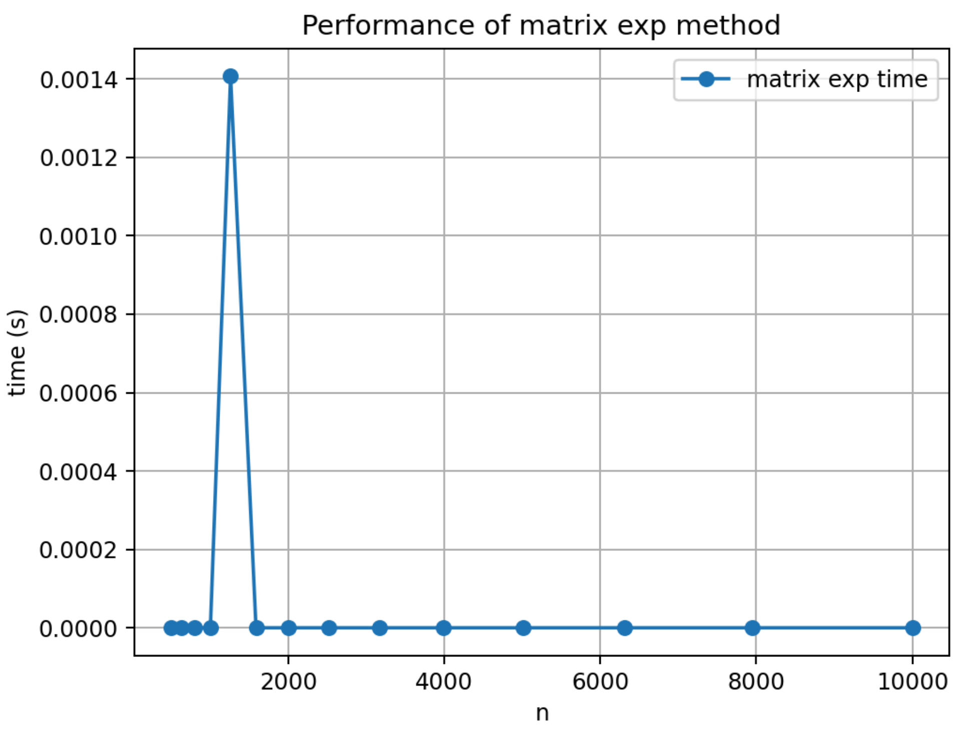
```
fibonacci_matrix(501), 0.000000 s, 6.45 KB
fibonacci_matrix(631), 0.001052 s, 6.30 KB
fibonacci_matrix(794), 0.000000 s, 6.46 KB
fibonacci_matrix(1000), 0.000000 s, 6.64 KB
fibonacci_matrix(1259), 0.000000 s, 6.75 KB
fibonacci_matrix(1585), 0.000000 s, 6.98 KB
fibonacci_matrix(1995), 0.000000 s, 7.30 KB
fibonacci_matrix(2512), 0.000000 s, 7.87 KB
fibonacci_matrix(3162), 0.001228 s, 8.39 KB
fibonacci_matrix(3981), 0.000000 s, 8.71 KB
fibonacci_matrix(5012), 0.000000 s, 9.87 KB
fibonacci_matrix(6310), 0.000000 s, 10.93 KB
fibonacci_matrix(7943), 0.000000 s, 11.65 KB
fibonacci_matrix(10000), 0.000000 s, 13.95 KB
```

Figure 15 Matrix Method Fibonacci Results

Performance: Extremely fast for large values of nnn.

Memory Usage: O(1)O(1)O(1), as only a few matrices are maintained.

Graph Analysis: Execution time grows logarithmically, outperforming DP for large nnn.



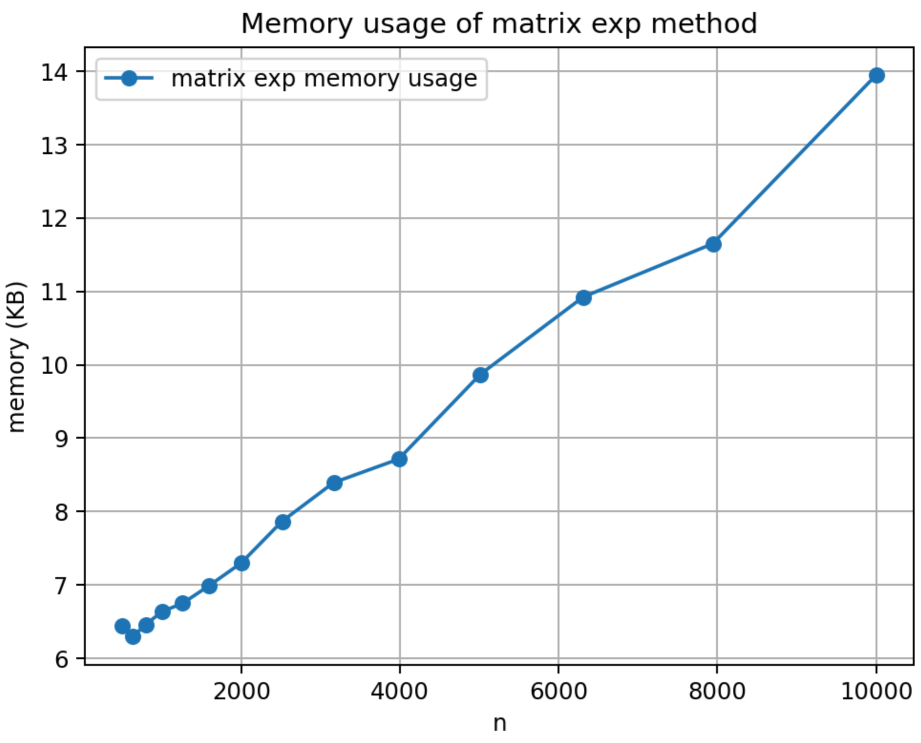


Figure 16 Matrix Method Fibonacci graph

Binet Formula Method:

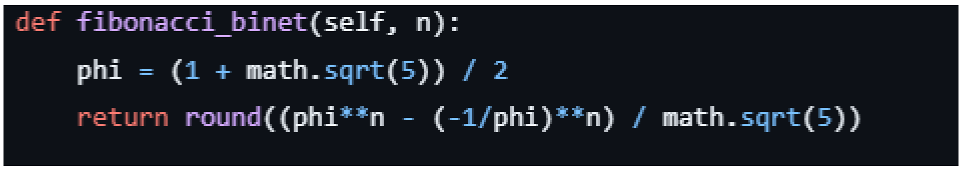
The Binet Formula Method is another unconventional way of calculating the n-th term of the Fibonacci series, as it operates using the Golden Ratio formula, or phi. However, due to its nature of requiring the usage of decimal numbers, at some point, the rounding error of python that accumulates, begins affecting the results significantly. The observation of error starting with around 70-th number making it unusable in practice, despite its speed. So, Binet's formula provides a direct mathematical computation using the golden ratio **ϕ**, offering constant-time complexity.

Algorithm Description:

While theoretically optimal, floating-point precision issues arise for large nnn, making this method unreliable in some cases. The set of operation for the Binet Formula Method can be described in pseudocode as follows:

```
Fibonacci(n):
  phi = (1 + sqrt(5)) / 2
  return round((phi^n - (-1/phi)^n) / sqrt(5))
```

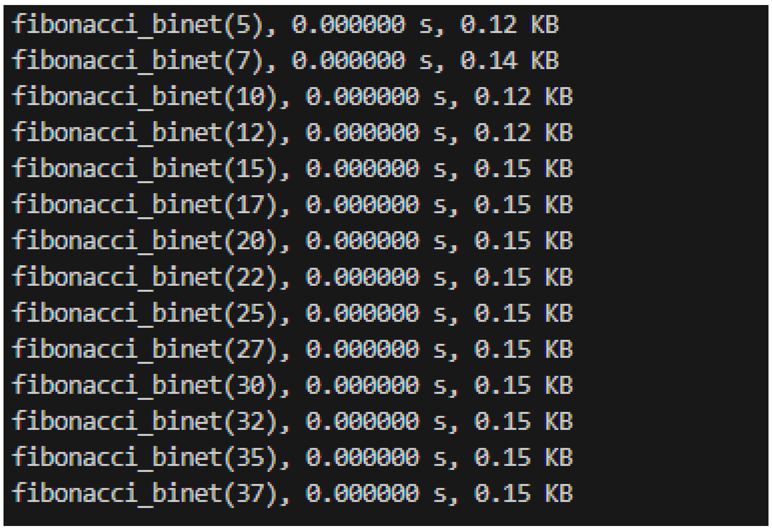
Implementation:



```python
def fibonacci_binet(self, n):
    phi = (1 + math.sqrt(5)) / 2
    return round((phi**n - (-1/phi)**n) / math.sqrt(5))
```

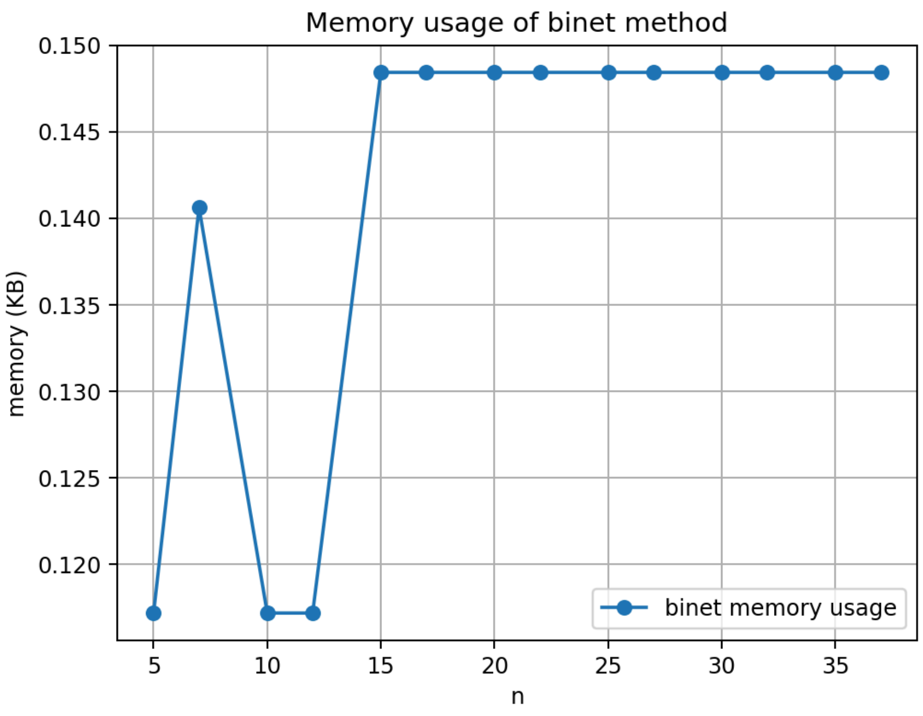
Figure 17 Fibonacci Binet Formula Method in Python

Results:



```
fibonacci_binet(5), 0.000000 s, 0.12 KB
fibonacci_binet(7), 0.000000 s, 0.14 KB
fibonacci_binet(10), 0.000000 s, 0.12 KB
fibonacci_binet(12), 0.000000 s, 0.12 KB
fibonacci_binet(15), 0.000000 s, 0.15 KB
fibonacci_binet(17), 0.000000 s, 0.15 KB
fibonacci_binet(20), 0.000000 s, 0.15 KB
fibonacci_binet(22), 0.000000 s, 0.15 KB
fibonacci_binet(25), 0.000000 s, 0.15 KB
fibonacci_binet(27), 0.000000 s, 0.15 KB
fibonacci_binet(30), 0.000000 s, 0.15 KB
fibonacci_binet(32), 0.000000 s, 0.15 KB
fibonacci_binet(35), 0.000000 s, 0.15 KB
fibonacci_binet(37), 0.000000 s, 0.15 KB
```

Figure 18 Fibonacci Binet Formula Method resul



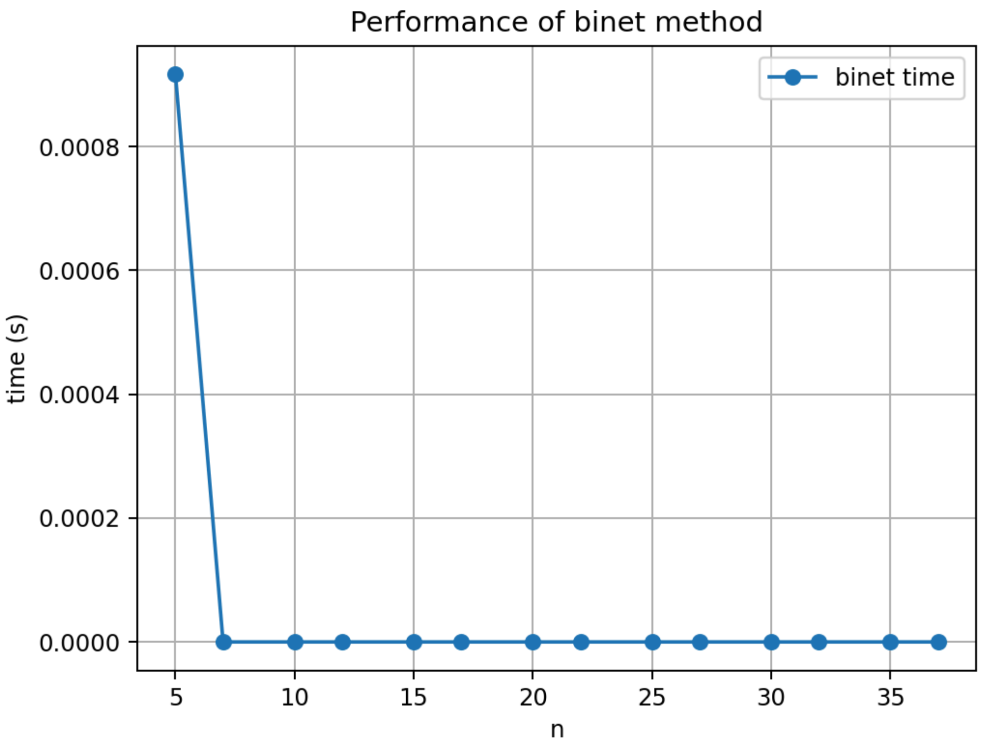


Figure 19 Fibonacci Binet formula Method

The Binet Formula Function is not accurate enough to be considered within the analysed limits and is recommended to be used for Fibonacci terms up to 80. At least in its naïve form in python, as further modification and change of language may extend its usability further.

Performance: Fastest method (O(1)), but precision errors occur for large n.

Memory Usage: O(1), as only a few variables are used.

Graph Analysis: Execution time remains constant, but inaccuracies increase with n.

CONCLUSION

Through this empirical analysis, six different Fibonacci computation methods were tested to measure their efficiency in terms of execution time and memory consumption. The goal was to determine the practical limits of each approach, understand their scalability, and figure out which one is best.

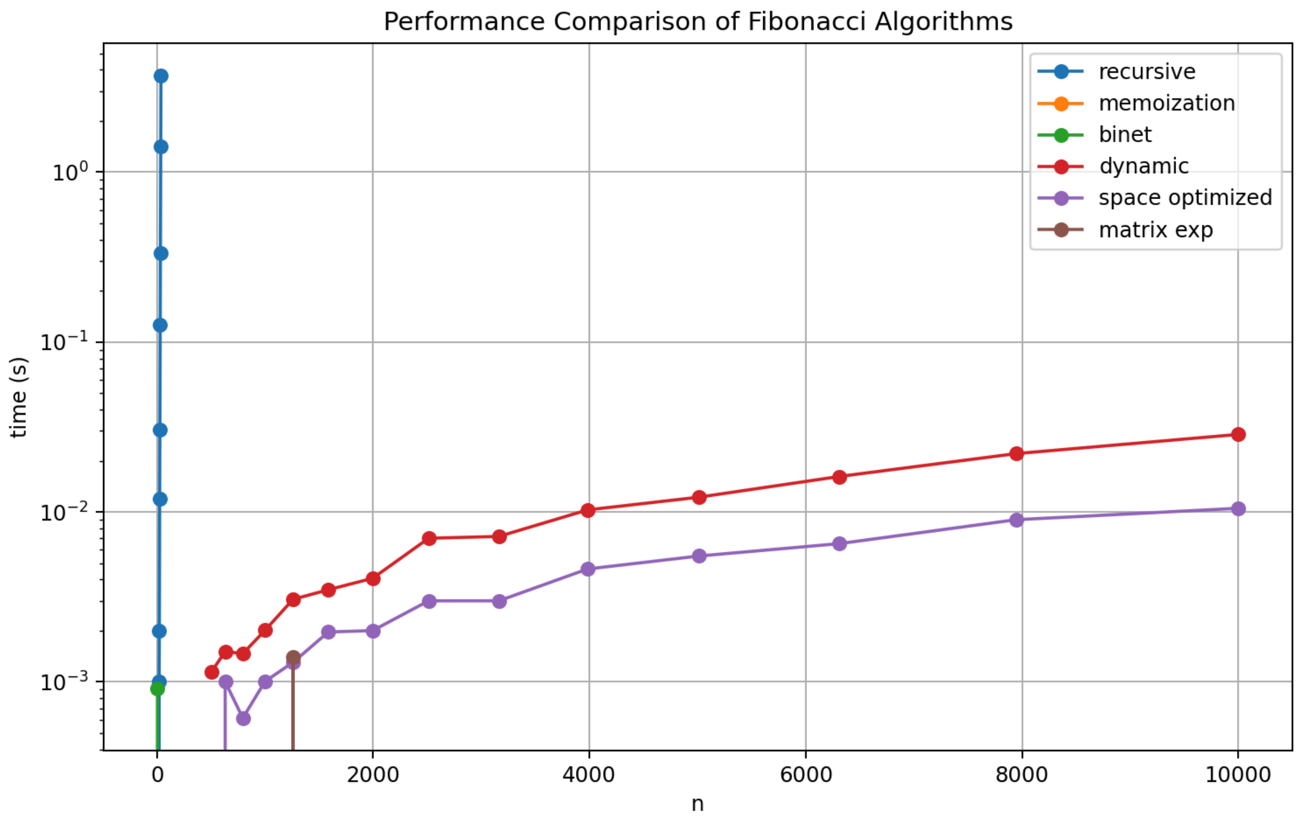
The Recursive method, while the easiest to write, was completely done in terms of performance. As visible from the steep vertical blue line, computation becomes practically impossible beyond n=30. This method has a time complexity of O(2^n), making it academically interesting but completely impractical for real applications.

The Binet formula, for small n, it works fine, but as n increases, errors creep in, making it unreliable for large values.

Dynamic Programming and space-optimized. As seen in the graph, these methods maintain a smooth, predictable execution time, proving their O(n) complexity. The space-optimized variant offers the same performance but uses significantly less memory, making it the better choice.

Matrix Exponentiation clearly shows its superior efficiency, drastically outperforming the previous methods for large n. By leveraging fast matrix exponentiation, it crushes Fibonacci calculations with logarithmic time complexity, making it the most practical choice for large-scale computations.

Thus, Matrix exponentiation emerges as the clear winner for large-scale Fibonacci calculations, combining accuracy with logarithmic performance. The space-optimized approach offers a good compromise when implementation simplicity matters. The recursive method should be avoided entirely for practical applications, and the Binet formula should only be used when approximate results are acceptable.


Performance Comparison of Fibonacci Algorithms
recursive
memoization
binet
dynamic
space optimized
matrix exp
time (s)
10^0
10^{-1}
10^{-2}
10^{-3}
n