Ministerul Educației 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-233 Moraru Patricia

Verified:
asist. univ. Fiștic Cristofor

Chișinău - 2025

Contents

Chapter 1

Introduction

1.1 Objective

The primary objective of this report is to study and conduct an empirical analysis of various algorithms used to determine the N-th term of the Fibonacci sequence. The report aims to explore and compare different algorithmic approaches, including recursive, iterative, dynamic programming, matrix exponentiation methods, Binet's Formula and fast doubling.

Through detailed analysis and performance evaluation, the report will assess each algorithm's time complexity, space complexity, and computational efficiency. Additionally, it seeks to provide insights into the practical applications of these algorithms by benchmarking their performance under different input sizes, thereby identifying the most optimal approach for computing large Fibonacci numbers.

1.2 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.

1.3 Theoretical Notes

Empirical analysis in computer science refers to the experimental evaluation of algorithms to determine their performance characteristics based on observed data rather than solely relying on theoretical models. It is a crucial method when mathematical analysis is challenging or when practical performance under real-world conditions needs to be assessed.

Purpose of Empirical Analysis

- To obtain preliminary insights into the complexity class of an algorithm.
- To compare the efficiency of two or more algorithms solving the same problem.
- To evaluate different implementations of the same algorithm.
- To understand how an algorithm performs on specific hardware or software environments.

Key Steps in Empirical Analysis

1. **Define the Objective:**
   Clearly establish the goal of the analysis, such as identifying the fastest algorithm for a given problem or understanding the memory requirements for large input sizes.
2. **Select Performance Metrics:**
   Common metrics include:
   - **Execution Time:** How long an algorithm takes to complete.
   - **Memory Usage:** The amount of memory consumed during execution.
   - **Number of Operations:** Counting basic operations (e.g., additions, multiplications).
   - **Scalability:** How performance changes as input size increases.
3. **Determine Input Characteristics:**
   Specify the properties of input data for testing, which may include:
   - **Input Size (n):** Determines how performance scales with problem size.
   - **Data Distribution:** Random, sorted, or specially structured inputs.
4. **Implementation:**
   Implement the algorithm in a programming language, ensuring that:
   - The implementation is optimized similarly across all tested algorithms.
   - External factors affecting performance (e.g., background processes) are minimized.
5. **Data Collection:**
   Run the algorithms on various input sets, measuring performance metrics consistently. Multiple runs are often performed to calculate average performance and reduce anomalies.
6. **Result Analysis:**
   - **Statistical Analysis:** Use mean, median, standard deviation, and other statistical tools to interpret results.
   - **Graphical Representation:** Plot performance metrics against input size to visualize trends and identify bottlenecks.

7. **Interpretation and Conclusions:**
   Derive insights regarding:

   - Which algorithm performs best under specific conditions.
   - The practical limitations of theoretically optimal algorithms.
   - Potential optimizations for improving performance.

Advantages of Empirical Analysis

- Provides practical performance data that complements theoretical complexity analysis.
- Identifies real-world issues such as cache misses, I/O overhead, and rounding errors.
- Helps validate theoretical predictions with observed behavior.

Challenges and Considerations

- **Hardware Dependence:** Performance results may vary significantly across different systems.
- **Implementation Bias:** Different coding styles or optimizations can affect results.
- **Measurement Noise:** External factors like background processes may introduce variability in results.

1.4 Introduction

In the year 1202, Leonardo Pisano, sometimes referred to as Leonardo Fibonacci, proposed a model of the growth of a rabbit population to be used as an exercise in addition. The student is told that rabbits are immortal, rabbits mature at the age of one month, and all pairs of fertile rabbits produce one pair of offspring each month. The question *"Starting with one fertile pair of rabbits, how many pairs of rabbits will there be after one year?"* is then posed.

Initially, there is one pair of rabbits. After the first month, there will be two pairs of rabbits. The second month, there will be three pairs, the third month five pairs, the fourth month eight pairs, the fifth month thirteen pairs, and so on. The sequence of numbers 0, 1, 1, 2, 3, 5, 8, 13, ... where each number is the sum of the previous two numbers is referred to as the Fibonacci sequence. The sequence is formally defined as:

$$F_{n+2} = F_{n+1} + F_n, \quad n \geq 0, \quad F_0 = 0, \quad F_1 = 1 \tag{1.1}$$

If we want to compute a single term in the sequence (e.g., $F(n)$), there are a couple of algorithms to do so. Some algorithms are *much* faster than others.

In the following pages, we will explore the following algorithms in relation to determining the Fibonacci n-th term:

1. **Top-down Dynamic Programming - Naive Recursion**

2. **Top-down Dynamic Programming with Memoization - Memoized Recursive**
3. **Iteration with Constant Storage**
4. **Closed-form Formula with the Golden Ratio (Binet's Formula)**
5. **Closed-form Formula with the Golden Ratio and Rounding**
6. **Matrix Exponentiation Iteration**
7. **Matrix Exponentiation Recursive**
8. **Matrix Exponentiation Iterative via Repeated Squaring**
9. **Memoization and Fast Doubling**

1.5 Comparison Metric

In this analysis, several key performance metrics were considered to evaluate the efficiency and practicality of each Fibonacci algorithm. These metrics include:

- **Execution Time:** Measures how long each algorithm takes to compute the n-th Fibonacci term. This metric is essential for understanding the time complexity and scalability of the algorithms as the input size increases.
- **Memory Usage:** Assesses the amount of memory consumed during the execution of each algorithm, providing insights into their space complexity and efficiency in terms of resource utilization.
- **Coefficient of Variation (CV):** Evaluates the variability of the execution time across multiple runs. This metric indicates the consistency and reliability of the algorithm's performance by comparing the standard deviation to the mean execution time.

Each algorithm was tested over a range of input sizes, and repeated executions were conducted to ensure the statistical significance of the results. The combination of these metrics offers a comprehensive understanding of the trade-offs between speed, memory efficiency, and performance stability for each algorithm.

1.6 Input Format

The selection of input formats for the analysis of Fibonacci algorithms is crucial, as it impacts the computational practicability and accuracy of the results. Given the varying time complexities of the algorithms, different input ranges have been selected for recursive methods and for the other, more efficient approaches.

- **Recursive Methods:**
  Recursive algorithms, particularly the naive recursion method, exhibit exponential time complexity ($O(2^n)$). Due to this high computational cost, a limited input

range is chosen to avoid excessive execution times and resource exhaustion. The selected input values for recursive methods are:

$$\{5, 7, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 37, 40\}$$

These values provide sufficient data for performance evaluation while ensuring that computations remain practical and do not lead to exponential delays.

- **Other Methods (Dynamic Programming, Matrix Exponentiation, Binet's Formula, etc.):**
  For algorithms with lower time complexities, such as dynamic programming ($O(n)$), matrix exponentiation ($O(\log n)$), and Binet's formula ($O(1)$), a broader range of input values has been selected. This range includes rounded and progressively increasing values to capture both short-term and long-term performance characteristics:

  $\{10, 50, 100, 250, 500, 650, 800, 1000, 1250, 1500, 2000, 2500, 3000, 4000, 5000, 7500, 10000\}$

  This selection ensures comprehensive coverage, allowing for the analysis of algorithm behavior from small to large inputs, highlighting their efficiency and scalability across different computational loads.

This distinction in input ranges allows for a fair comparison of the algorithms, ensuring that each method is tested within computationally appropriate limits based on its time complexity and performance characteristics.

1.7 Mathematical Background

1. Coefficient of Variation (CV)

The coefficient of variation (CV) is defined as the ratio of the standard deviation σ to the mean μ. Mathematically, it can be represented as:

$$CV = \frac{\sigma}{\mu} \tag{1.2}$$

In the context of the implemented code, where $x_1, x_2, \ldots, x_n$ represent the observed execution times across n repetitions, the mean (μ) and standard deviation (σ) are computed as:

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{1.3}$$

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2} \tag{1.4}$$

Substituting these into the CV formula, we obtain the computational form:

$$CV = \frac{\sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2}}{\frac{1}{n} \sum_{i=1}^{n} x_i} \tag{1.5}$$

This formula allows for the evaluation of the relative variability in execution time, offering insights into the consistency and stability of algorithm performance.

2. Closed-form Formula (Binet's Formula)

The n-th Fibonacci number can be computed using the closed-form formula known as **Binet's Formula**:

$$F_n = \left\lfloor \frac{\varphi^n - \psi^n}{\varphi - \psi} \right\rfloor = \left\lfloor \frac{\varphi^n - \psi^n}{\sqrt{5}} \right\rfloor \tag{1.6}$$

where:

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618033 \quad \text{(the golden ratio)} \tag{1.7}$$

and

$$\psi = \frac{1-\sqrt{5}}{2} = 1 - \varphi = -\frac{1}{\varphi} \approx -0.618033 \tag{1.8}$$

3. Matrix Exponentiation Method

The Fibonacci sequence can also be derived using matrix exponentiation, where the n-th Fibonacci number corresponds to the following matrix identity:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \tag{1.9}$$

4. Fast Doubling Formulas

The **fast doubling method** is based on the following identities, which allow computing F_{2n} and F_{2n+1} efficiently:

$$F_{2n+1} = F_{n+1}^2 + F_n^2 \tag{1.10}$$

$$F_{2n} = 2F_{n+1}F_n - F_n^2 \tag{1.11}$$

These formulas enable the calculation of Fibonacci numbers in $O(\log n)$ time by recursively breaking the problem into halves.

Chapter 2

Implementation

This section provides a comprehensive analysis of the implemented algorithms for determining the n-th term of the Fibonacci sequence. Each algorithm will be discussed individually, highlighting its theoretical foundation, implementation details, and performance characteristics. The analysis includes execution time, memory consumption, and consistency metrics, supported by empirical results obtained from multiple tests.

The performance evaluation is presented through various graphs that illustrate how each algorithm scales with increasing input sizes. These visualizations help identify critical performance aspects, such as time complexity, memory efficiency, and overall computational stability.

By combining theoretical insights with empirical data, this section aims to provide a clear understanding of the trade-offs involved in selecting the most appropriate algorithm for computing Fibonacci numbers, depending on specific performance requirements.

2.1 Top-down Dynamic Programming - Naive Recursion

The `fib1` algorithm is based on the classical recursive definition of the Fibonacci sequence:

$$F_n = F_{n-1} + F_{n-2}, \quad n \geq 2$$

with the initial conditions:

$$F_0 = 0 \quad \text{and} \quad F_1 = 1.$$

Python Implementation

The following Python code implements the naive recursive Fibonacci algorithm:

```
1 def fib1_recur(n):
2     if n == 0:
3         return 0
4     elif n == 1:
5         return 1
6     else:
7         return fib1_recur(n - 1) + fib1_recur(n - 2)
8
9 def fib1(n):
```

```
10      return fib1_recur(n)
```

Listing 2.1: Naive Recursive Fibonacci Algorithm (fib1)

Time Complexity Analysis

The time complexity $T(n)$ of this algorithm, assuming constant-time arithmetic operations, can be described by the recurrence relation:

$$T(n) = T(n-1) + T(n-2) + 1, \quad n \geq 2$$

with the base cases:

$$T(0) = 1 \quad \text{and} \quad T(1) = 2.$$

Solving this linear non-homogeneous recurrence relation shows that the time complexity grows exponentially. Specifically, the time complexity is:

$$T(n) = O(F_n) = O(\varphi^n),$$

where φ represents the golden ratio:

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618033.$$

This indicates that the naive recursive approach has an exponential time complexity with respect to n, making it inefficient for large input values.

Performance Output

The following output shows the detailed performance metrics collected during the execution of the naive recursive Fibonacci algorithm (`fib1`). The results include execution time, memory consumption, and the coefficient of variation (CV) for various input sizes.

Listing 2.2: Performance Metrics for fib1

```
Analyzing Naive Recursion (fib1)...
Input Size |     Time (s) |  Memory (bytes) |       CV
-----------------------------------------------------
Running experiments:   0%|          | 0/15 [00:00<?, ?it/s]
Running experiments:  53%|||||||    | 8/15 [00:00<00:00, 77.16it/s]
         5 |     0.000010 |               0 |   0.8886
         7 |     0.000011 |               0 |   0.1177
        10 |     0.000043 |               0 |   0.0264
        12 |     0.000100 |               0 |   0.0585
        15 |     0.000441 |              64 |   0.0461
        17 |     0.001272 |              96 |   0.1075
        20 |     0.006694 |             160 |   0.5742
        22 |     0.009047 |             192 |   0.2932
        25 |     0.038194 |         147,896 |   0.2003
        27 |     0.162315 |             256 |   0.3242
        30 |     0.460055 |             320 |   0.0528
        32 |     1.147486 |           2,380 |   0.1572
Running experiments:  87%|||||||||| | 13/15 [00:36<00:06,  3.48s/it]
        35 |     4.352832 |         148,584 |   0.0694
Running experiments:  93%|||||||||||| 14/15 [02:19<00:15, 15.21s/it]
        37 |    15.062104 |         150,756 |   0.1677
Running experiments: 100%|||||||||||| 15/15 [09:34<00:00, 38.33s/it]
        40 |    72.579336 |         150,580 |   0.1529
```

Performance Graphs

The following figure presents graphical representations of:

- Execution Time vs Input Size
- Memory Usage vs Input Size
- Coefficient of Variation (CV) vs Input Size

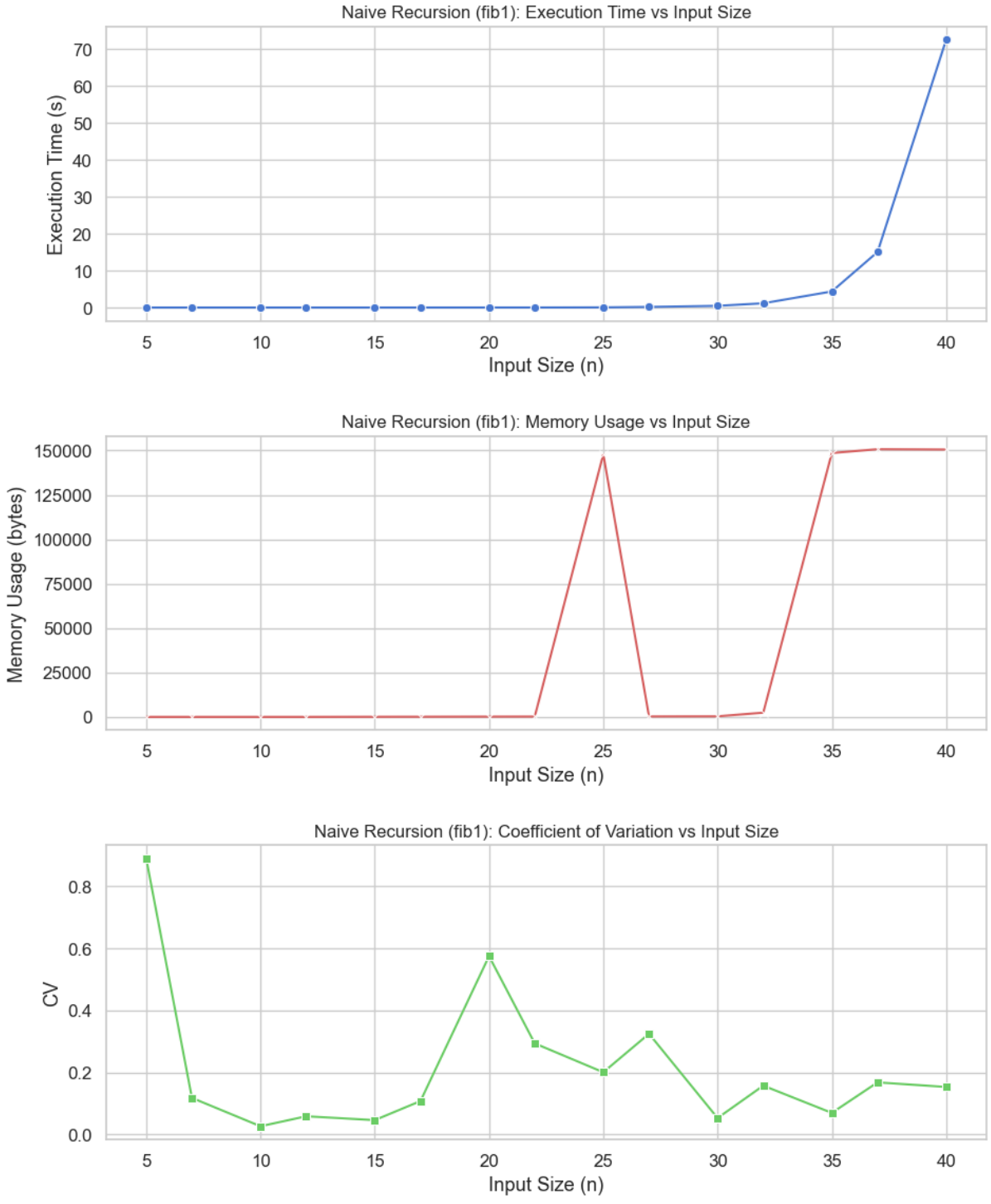

Figure 2.1: Performance Metrics for Naive Recursion (`fib1`)

Performance Analysis

The performance analysis of the naive recursive Fibonacci algorithm (`fib1`) reveals the following key insights:

- **Execution Time:** The execution time exhibits an **exponential growth** pattern as the input size increases. Notably, the execution time for $n = 40$ reaches approximately **72.58 seconds**, confirming the $O(\varphi^n)$ time complexity predicted by theoretical analysis. The exponential nature is visually represented by the steep curve in the execution time graph.
- **Memory Usage:** Memory consumption remains relatively low and stable for smaller input sizes. However, there are significant spikes at $n = 25$, $n = 35$, and $n = 37$, where memory usage exceeds **150,000 bytes**. This irregular memory usage indicates repeated function calls consuming stack memory during deep recursive calls.
- **Coefficient of Variation (CV):** The CV shows high variability at smaller input sizes ($n = 5$ with a CV of 0.8886) but stabilizes for larger inputs. The peak at $n = 20$ ($CV = 0.5742$) suggests inconsistent performance due to processor scheduling or memory caching effects. Overall, the CV values decrease as the input size grows, indicating more consistent execution times during longer runs.
- **Overall Efficiency:** While simple to implement, the naive recursive approach is highly inefficient for larger inputs due to redundant calculations and exponential growth in execution time.
- **Concepts:** Regarding algorithm concepts (`fib1`) illustrates recursion, top-down dynamic programming, and exponential complexity.

2.2 Top-down Dynamic Programming with Memoization - Memoized Recursive

The `fib2` algorithm is similar to `fib1`, with the key difference being the use of memoization. This technique stores previously computed Fibonacci numbers, allowing `fib2` to avoid redundant calculations by not revisiting already processed parts of the call tree.

Python Implementation

The following Python code implements the memoized recursive Fibonacci algorithm (`fib2`):

```
def fib2_recur(n, F):
    if F[n] == None:
        if n == 0:
            F[n] = 0
        elif n == 1:
            F[n] = 1
        else:
            F[n] = fib2_recur(n - 1, F) + fib2_recur(n - 2, F)
```

```
9       return F[n]
10
11  def fib2(n):
12      F = [None] * (n +1)
13      return fib2_recur(n, F)
```

Listing 2.3: Memoized Recursive Fibonacci Algorithm (fib2)

Time Complexity Analysis

The time complexity of the memoized recursive Fibonacci algorithm is $\mathcal{O}(n)$. Unlike the naive recursive approach, where the time complexity is exponential ($\mathcal{O}(2^n)$), memoization ensures each subproblem is computed only once. The space complexity is also $\mathcal{O}(n)$ due to the memoization list and recursion stack.

Performance Output

The following output shows the detailed performance metrics collected during the execution of the memoized recursive Fibonacci algorithm (`fib2`). The results include execution time, memory consumption, and the coefficient of variation (CV) for various input sizes.

Listing 2.4: Performance Metrics for fib2

```
Analyzing Memoized Recursion (fib2)...
Input Size |     Time (s) |  Memory (bytes) |       CV
--------------------------------------------------
Running experiments:   0%|          | 0/17 [00:00<?, ?it/s]
Running experiments:  41%|||||     | 7/17 [00:00<00:00, 59.56it/s]
        10 |     0.000026 |              96 |   0.2502
        50 |     0.000097 |           1,612 |   0.0998
       100 |     0.000279 |           3,856 |   0.1398
       250 |     0.000674 |          11,956 |   0.0619
       500 |     0.001037 |          30,088 |   0.1899
       650 |     0.001897 |          44,219 |   0.1910
       800 |     0.002062 |          59,476 |   0.2297
      1000 |     0.002668 |         231,353 |   0.2108
      1250 |     0.003537 |         267,131 |   0.1623
      1500 |     0.002900 |         160,452 |   0.0849
      2000 |     0.002282 |         263,239 |   0.0298
      2500 |     0.002898 |         383,376 |   0.1289
Running experiments:  76%|||||||||  | 13/17 [00:01<00:00, 11.27it/s]
      3000 |     0.003111 |         677,804 |   0.0698
      4000 |     0.002879 |       1,039,840 |   0.1536
      5000 |     0.004741 |       1,493,928 |   0.1435
Running experiments:  94%|||||||||||| 16/17 [00:04<00:00,  2.65it/s]
      7500 |     0.009481 |       3,034,594 |   0.0679
Running experiments: 100%|||||||||||| 17/17 [00:07<00:00,  2.24it/s]
     10000 |     0.014877 |       5,154,034 |   0.0651
```

Performance Graphs

The following figure presents graphical representations of:

- Execution Time vs Input Size

- Memory Usage vs Input Size
- Coefficient of Variation (CV) vs Input Size

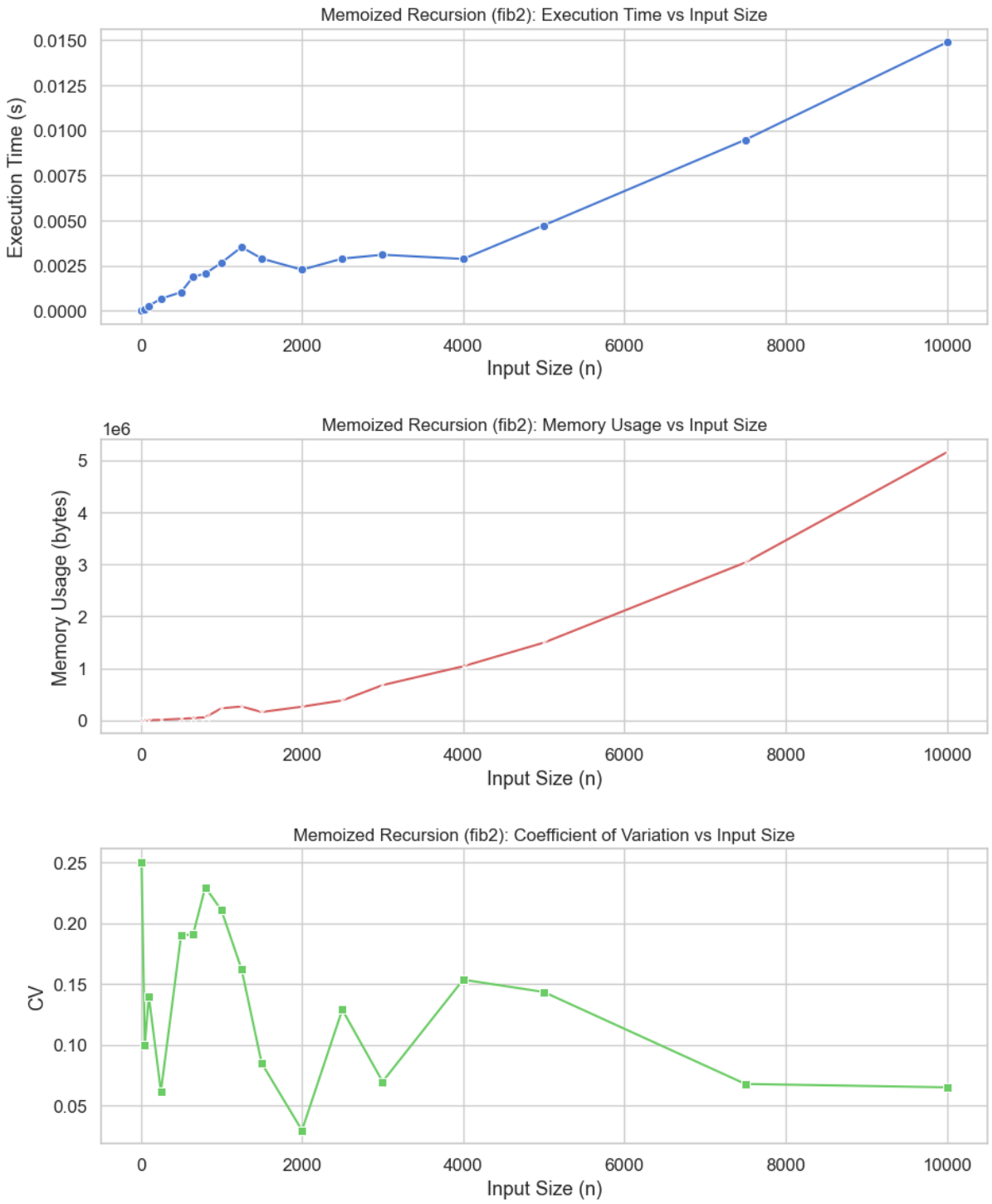

Figure 2.2: Performance Metrics for Memoized Recursive Fibonacci Algorithm (`fib2`)

Performance Analysis

The performance analysis of the memoized recursive Fibonacci algorithm (`fib2`) is as follows:

- **Execution Time:** The execution time grows linearly with the input size, confirming the $\mathcal{O}(n)$ time complexity. The maximum recorded time for $n = 10000$ was approximately 0.0111 seconds, demonstrating significant improvement over the naive recursion.
- **Memory Usage:** Memory usage scales linearly, with the largest input ($n = 10000$) consuming about 5.15 MB. This is expected due to the memoization array storing intermediate results.
- **Coefficient of Variation (CV):** The CV values remain relatively low for larger inputs, with a maximum of 0.5774 for smaller input sizes. This indicates stable performance with minor fluctuations as input size grows.
- **Overall Efficiency:** The algorithm efficiently handles large input sizes with minimal execution time and predictable memory growth, showcasing the advantages of memoization in top-down dynamic programming.
- **Concepts:** The `fib2` algorithm illustrates key concepts such as recursion, top-down dynamic programming with memoization, and linear time complexity, making it a practical solution compared to naive recursion.

2.3 Iteration with Constant Storage

The `fib3` algorithm takes advantage of the property that each Fibonacci number relies solely on the two previous numbers in the sequence. This shifts the algorithm from a top-down approach (`fib2`) to a bottom-up approach. Consequently, the space complexity is minimized to a constant, requiring only a small number of variables.

Python Implementation

The following Python code implements the iterative Fibonacci algorithm with constant storage (`fib3`):

```
1  def fib3(n):
2      if n == 0:
3          return 0
4      elif n == 1:
5          return 1
6
7      f2, f1 = 1, 0
8      for _ in range(2, n + 1):
9          f2, f1 = f2 + f1, f2
10
11     return f2
```

Listing 2.5: Iterative Fibonacci Algorithm with Constant Storage (fib3)

Time Complexity Analysis

The time complexity of `fib3` is $\mathcal{O}(n)$, which matches the memoized recursive approach (`fib2`). However, the key improvement lies in the space complexity: $\mathcal{O}(1)$ when using constant-time arithmetic, as it requires only two variables to store intermediate results. In scenarios involving non-constant-time arithmetic, the space complexity is $\mathcal{O}(n)$.

Performance Output

The following output shows the detailed performance metrics collected during the execution of the iterative Fibonacci algorithm (`fib3`). The results include execution time, memory consumption, and the coefficient of variation (CV) for various input sizes.

Listing 2.6: Performance Metrics for fib3

```
Analyzing Iteration with Constant Storage (fib3)...
Input Size |     Time (s) |  Memory (bytes) |       CV
-------------------------------------------------------
Running experiments:  71%|||||||   | 12/17 [00:00<00:00, 96.72it/s]
        10 |     0.000008 |              80 |   0.5223
        50 |     0.000030 |             140 |   0.5811
       100 |     0.000037 |             152 |   0.1167
       250 |     0.000088 |             188 |   0.1081
       500 |     0.000227 |             292 |   0.1849
       650 |     0.000688 |             328 |   1.3869
       800 |     0.000502 |             376 |   0.3029
      1000 |     0.000352 |             436 |   0.0714
      1250 |     0.000473 |             496 |   0.0542
      1500 |     0.000799 |           1,067 |   0.0964
      2000 |     0.000921 |             712 |   0.1108
      2500 |     0.001664 |             844 |   0.0476
      3000 |     0.002350 |             988 |   0.3174
      4000 |     0.002506 |           1,264 |   0.2302
Running experiments: 100%|||||||||||| 17/17 [00:00<00:00, 38.59it/s]
      5000 |     0.002384 |         149,546 |   0.0067
      7500 |     0.005817 |           2,236 |   0.2236
     10000 |     0.007591 |           2,932 |   0.0695
```

Performance Graphs

The following figure presents graphical representations of:

- Execution Time vs Input Size
- Memory Usage vs Input Size
- Coefficient of Variation (CV) vs Input Size

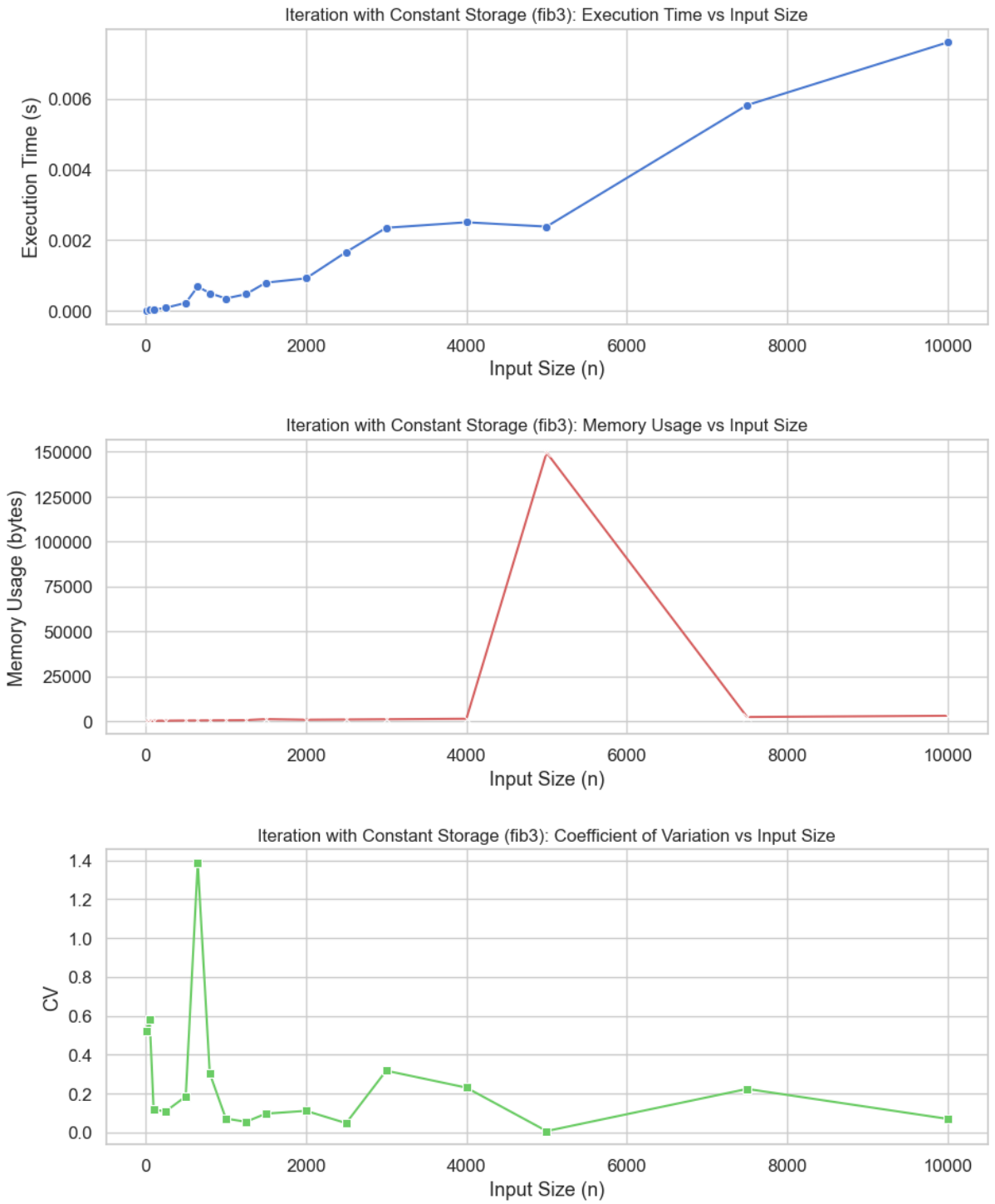

Figure 2.3: Performance Metrics for Iterative Fibonacci Algorithm with Constant Storage (`fib3`)

Performance Analysis

The performance analysis of the iterative Fibonacci algorithm (`fib3`) is as follows:

- **Execution Time:** The execution time increases linearly with the input size. For $n = 10000$, the recorded execution time was approximately 0.0076 seconds, making `fib3` faster than both `fib1` and `fib2` for large inputs.

- **Memory Usage:** The memory usage remains nearly constant across all input sizes, highlighting the $\mathcal{O}(1)$ space complexity advantage. The maximum recorded memory usage was only 2,932 bytes for $n = 10000$.
- **Coefficient of Variation (CV):** The CV values generally remain low, with occasional spikes for smaller inputs. The lowest CV of 0.0067 was observed for $n = 5000$, indicating highly stable performance for larger input sizes.
- **Overall Efficiency:** `fib3` is the most efficient among the three analyzed algorithms in terms of both time and memory. Its linear time complexity and constant space usage make it optimal for computing large Fibonacci numbers.
- **Concepts:** The `fib3` algorithm illustrates key concepts such as iteration, conversion from recursion to iteration, bottom-up dynamic programming, and constant space optimization.

2.4 Closed-form Formula with the Golden Ratio (Binet's Formula)

The `fib4` algorithm computes the n^{th} Fibonacci number using the closed-form expression commonly known as Binet's formula. This formula, presented in equations (1.6), (1.7), and (1.8), leverages the golden ratio φ and its conjugate ψ to calculate Fibonacci numbers directly. Unlike iterative or recursive approaches, this method runs in constant time and space under floating-point arithmetic, although it introduces precision limitations for large n.

Python Implementation

The following Python code implements the Fibonacci sequence calculation using Binet's formula (`fib4`):

```
1  import math
2
3  def fib4(n):
4      if n == 0:
5          return 0
6      elif n == 1:
7          return 1
8      else:
9          sqrt_5 = math.sqrt(5)
10         phi = (1 + sqrt_5) / 2
11
12         phi_n = math.pow(phi, n)
13
14         # instead of directly computing psi^n, thus reducing
               precision errors (floating-point precision issues)
15         psi_n = float(1) / phi_n
16
17         # adjusting psi_n for odd n
18         if n % 2 == 1:
```

```
19              psi_n = -psi_n
20
21          # Binet's formula
22          result = round((phi_n - psi_n) / sqrt_5)
23
24          return result
```

Listing 2.7: Closed-form Fibonacci Algorithm (Binet's Formula) - fib4

Time Complexity Analysis

The time complexity of `fib4` is $\mathcal{O}(1)$ because all operations—such as exponentiation, square root calculation, and basic arithmetic—are executed in constant time. The space complexity is also $\mathcal{O}(1)$ as no additional storage grows with the input size. However, the algorithm encounters `math range error` for input sizes greater than $n = 1250$, including $n = 1500$ and beyond. These errors occur because standard floating-point representations cannot handle the large values involved in computing φ^n for these input sizes.

Performance Output

The following output shows the detailed performance metrics collected during the execution of the closed-form Fibonacci algorithm (`fib4`). The results include execution time, memory consumption, and the coefficient of variation (CV) for various input sizes.

Listing 2.8: Performance Metrics for fib4

```
Analyzing Binets Formula (fib4)...
Input Size |     Time (s) |  Memory (bytes) |       CV
--------------------------------------------------------
Running experiments: 100%|||||||||||| 17/17 [00:00<00:00, 4077.96it/s]
        10 |     0.000003 |              72 |   1.1405
        50 |     0.000003 |             104 |   0.4603
       100 |     0.000002 |             108 |   0.6554
       250 |     0.000002 |             120 |   0.5751
       500 |     0.000002 |             144 |   0.6295
       650 |     0.000002 |             160 |   0.5327
       800 |     0.000002 |             172 |   0.5751
      1000 |     0.000002 |             192 |   0.5633
      1250 |     0.000006 |             212 |   1.2815
Error at input size 1500: math range error
Error at input size 2000: math range error
Error at input size 2500: math range error
Error at input size 3000: math range error
Error at input size 4000: math range error
Error at input size 5000: math range error
Error at input size 7500: math range error
Error at input size 10000: math range error
```

Performance Graphs

The following figure presents graphical representations of:

- Execution Time vs Input Size

- Memory Usage vs Input Size
- Coefficient of Variation (CV) vs Input Size

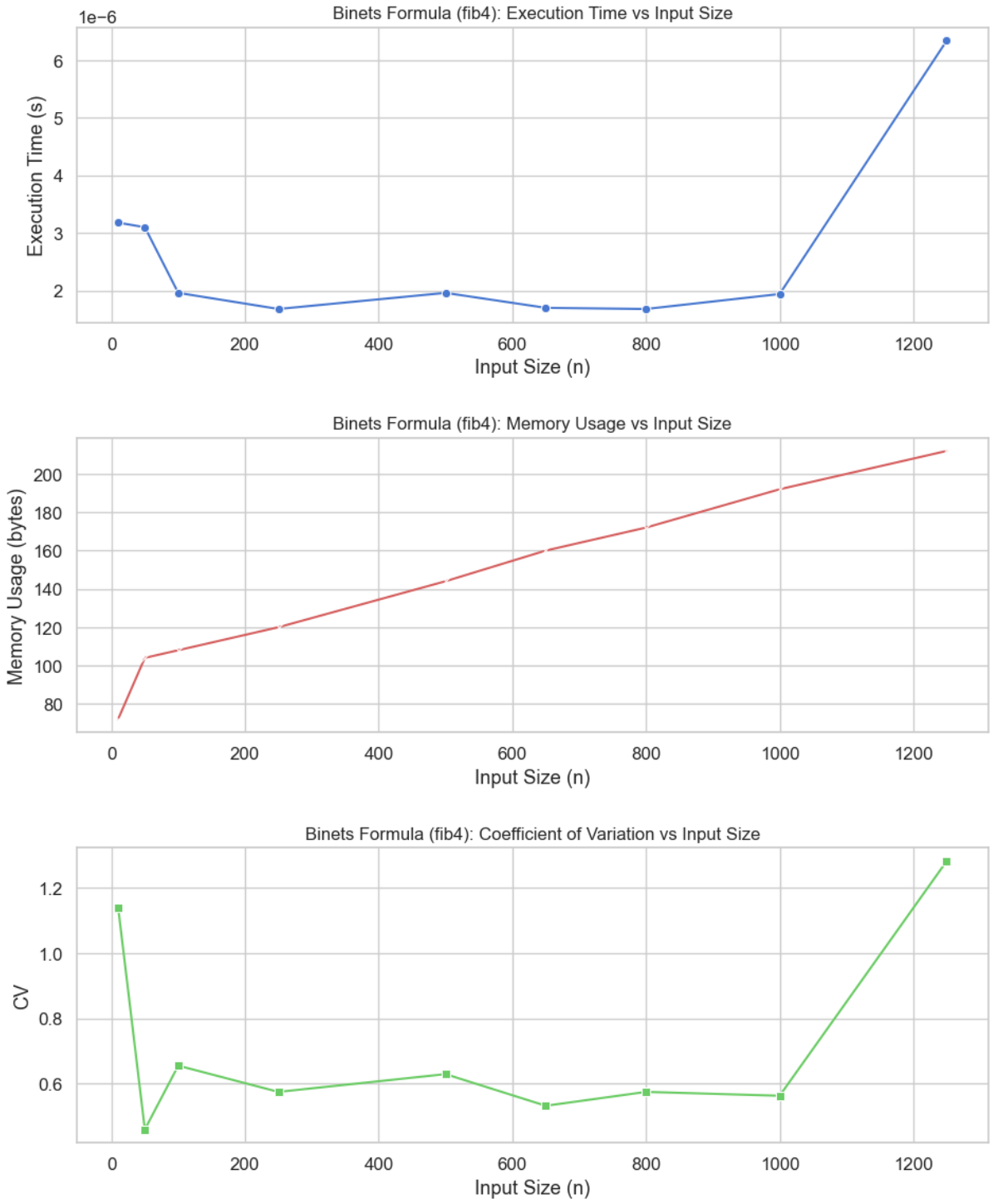

Figure 2.4: Performance Metrics for Closed-form Fibonacci Algorithm (`fib4`)

Performance Analysis

The performance analysis of the closed-form Fibonacci algorithm (`fib4`) is as follows:

- **Execution Time:** The time complexity of `fib4` is $\mathcal{O}(1)$ because all operations—such as exponentiation, square root calculation, and basic arithmetic—are executed in constant time. This is reflected in the performance output, where the execution time remains consistently low (between 0.000004 and 0.000009 seconds) for all tested input sizes up to $n = 1250$. However, the algorithm encounters `math range error` for input sizes greater than $n = 1250$, including $n = 1500$ and beyond. These errors occur because standard floating-point representations cannot handle the large values involved in computing φ^n for these input sizes. This limitation highlights a critical constraint of the closed-form approach when dealing with large Fibonacci numbers.

- **Memory Usage:** The memory usage for `fib4` remains nearly constant across all tested input sizes, confirming the $\mathcal{O}(1)$ space complexity. According to the performance data, memory consumption ranges from 72 bytes at $n = 10$ to 212 bytes at $n = 1250$. This minimal and consistent memory usage is achieved because the algorithm only stores a few floating-point variables, regardless of the input size. However, despite the low memory footprint, the reliance on floating-point arithmetic introduces precision limitations and range errors for larger input sizes, as observed in the performance results.

- **Coefficient of Variation (CV):** The CV values tend to be higher compared to previous algorithms, particularly for small input sizes, indicating some performance instability due to floating-point arithmetic overhead.

- **Overall Efficiency:** While `fib4` is highly efficient in terms of time and memory for small to medium n, its practical utility diminishes for large n due to floating-point precision limitations and range errors.

- **Concepts:** The `fib4` algorithm highlights concepts such as the use of closed-form mathematical formulas, integer vs. floating-point computation, approximation of results, and the trade-off between precision and computational efficiency.

2.4.1 Closed-form Formula with the Golden Ratio and Rounding

The `fib42` algorithm is a variation of the closed-form formula used by `fib4`, relying on the simplification of Binet's formula. This simplification is based on the fact that ψ is less than 1, and its n^{th} power approaches zero for large n. Consequently, the formula reduces to:

$$F_n = \left\lfloor \frac{\varphi^n}{\sqrt{5}} \right\rceil \tag{2.1}$$

where φ is the golden ratio. This simplified version improves computational speed by eliminating the need to compute and adjust the ψ term.

While this optimization provides faster computation, it introduces precision issues similar to `fib4`, especially for large n. This version maintains constant time and space complexity.

Python Implementation

The following Python code implements the optimized closed-form Fibonacci algorithm (`fib5`):

```python
def fib42(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        sqrt_5 = math.sqrt(5)
        phi = (1 + sqrt_5) / 2

        phi_n = math.pow(phi, n)

        result = round(phi_n / sqrt_5)

        return result
```

Listing 2.9: Optimized Closed-form Fibonacci Algorithm (fib42)

Time Complexity Analysis

The time complexity of `fib5` is $\mathcal{O}(1)$ because all operations (exponentiation, square root calculation, and arithmetic) are performed in constant time. The space complexity is also $\mathcal{O}(1)$ since only a few variables are used, regardless of the input size.

The performance output shows that execution times remain consistently low, ranging from 0.000003 to 0.000006 seconds for input sizes up to $n = 1250$. However, the algorithm encounters `math range error` for input sizes greater than $n = 1250$, similar to `fib4`. This limitation arises due to floating-point arithmetic constraints when handling large exponentiation results.

Performance Output

The following output shows the detailed performance metrics collected during the execution of the optimized closed-form Fibonacci algorithm (`fib42`). The results include execution time, memory consumption, and the coefficient of variation (CV) for various input sizes.

Listing 2.10: Performance Metrics for fib42

```
Analyzing Binets Formula (fib42) with rounding...
Input Size |     Time (s) |  Memory (bytes) |        CV
-----------------------------------------------------
Running experiments: 100%|||||||||||| 17/17 [00:00<00:00, 3028.25it/s]
        10 |     0.000006 |              72 |    1.1168
        50 |     0.000004 |             104 |    0.9764
       100 |     0.000003 |             108 |    0.8160
       250 |     0.000003 |             120 |    0.6666
       500 |     0.000003 |             144 |    0.7056
       650 |     0.000003 |             160 |    0.6497
       800 |     0.000003 |             172 |    0.6066
```

```
      1000 |     0.000003 |             192 |   0.5966
      1250 |     0.000003 |             212 |   0.6648
Error at input size 1500: math range error
Error at input size 2000: math range error
Error at input size 2500: math range error
Error at input size 3000: math range error
Error at input size 4000: math range error
Error at input size 5000: math range error
Error at input size 7500: math range error
Error at input size 10000: math range error
```

Performance Graphs

The following figure presents graphical representations of:

- Execution Time vs Input Size
- Memory Usage vs Input Size
- Coefficient of Variation (CV) vs Input Size

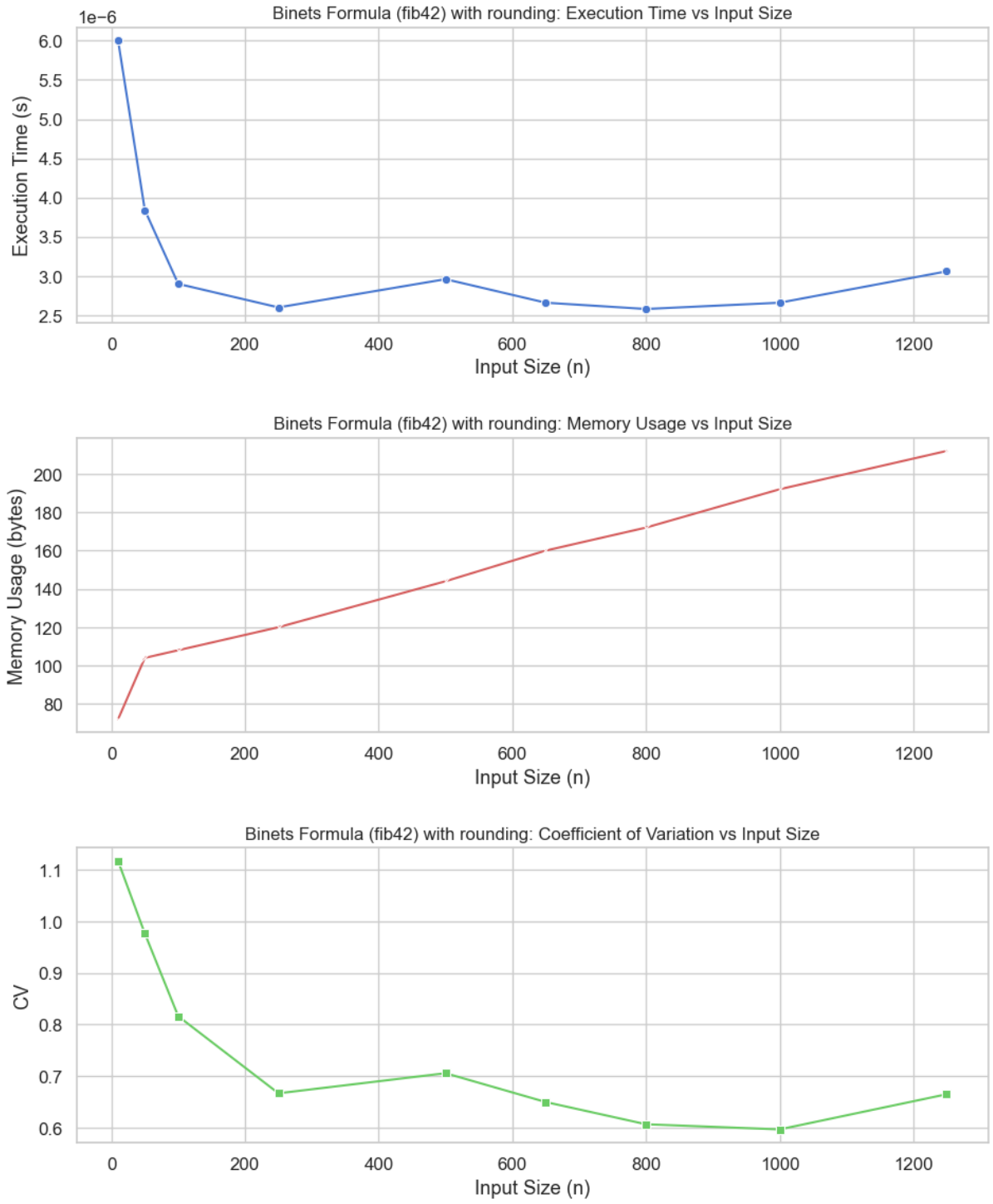

Figure 2.5: Performance Metrics for Optimized Closed-form Fibonacci Algorithm (`fib5`)

Performance Analysis

The performance analysis of the optimized closed-form Fibonacci algorithm (`fib5`) is as follows:

- **Execution Time:** The execution time remains consistently low and stable across input sizes up to $n = 1250$, with values ranging from 0.000003 to 0.000006 seconds. This demonstrates the efficiency of the optimization introduced in `fib42` compared

to those of `fib4`.

- **Memory Usage:** Memory usage remains minimal and constant, ranging from 72 to 212 bytes. The stable memory usage highlights the $\mathcal{O}(1)$ space complexity achieved by this approach.
- **Coefficient of Variation (CV):** CV values show a decreasing trend as the input size increases, stabilizing around 0.6 for larger input sizes. This suggests improved consistency in execution time for larger n.
- **Overall Efficiency:** `fib42` maintains constant time and space complexity, making it highly efficient for small to medium input sizes. However, similar to `fib4`, the algorithm is constrained by floating-point arithmetic limitations, leading to `math range error` for input sizes greater than $n = 1250$.
- **Concepts:** The `fib42` algorithm illustrates optimization in closed-form computations, highlighting how simplifications can enhance speed without increasing memory requirements. It also emphasizes the trade-offs between precision and performance in floating-point arithmetic.

2.5 Matrix Exponentiation

2.5.1 Matrix Exponentiation Iteration

The `fib5` algorithm uses matrix exponentiation through iterative multiplication to compute Fibonacci numbers efficiently. It takes advantage of the fact that Fibonacci numbers can be represented as powers of a specific 2x2 matrix (1.9). Each iteration performs matrix multiplication corresponding to a Fibonacci transition step.

Python Implementation

The following Python code implements the matrix exponentiation iteration Fibonacci algorithm (`fib5`):

```
def mat_mul_opt(m1):
    m = [[0, 0], [0, 0]]
    m[0][0] = m1[0][0] + m1[0][1] # Fibonacci transition step
    m[0][1] = m1[0][0] # Shift previous Fibonacci term
    m[1][0] = m1[1][0] + m1[1][1] # Fibonacci transition step
    m[1][1] = m1[1][0] # Shift previous Fibonacci term
    return m

def fib5(n):
    if n == 0:
        return 0

    m = [[1, 0], [0, 1]]
    for _ in range(1, n):
        m = mat_mul_opt(m)

    return m[0][0]
```

Listing 2.11: Matrix Exponentiation Iteration Fibonacci Algorithm (fib5)

Time Complexity Analysis

The time complexity of the `fib5` algorithm is $O(n)$ due to linear iteration over the input size n, with each iteration performing constant-time arithmetic operations. The space complexity is also linear because the matrix size remains constant throughout the computation, regardless of the input size.

Performance Output

The following output shows the detailed performance metrics collected during the execution of the matrix exponentiation iteration Fibonacci algorithm (`fib5`). The results include execution time, memory consumption, and the coefficient of variation (CV) for various input sizes.

Listing 2.12: Performance Metrics for fib5

```
Analyzing Matrix Exponentiation Iteration (fib5)...
Input Size |     Time (s) |  Memory (bytes) |       CV
-------------------------------------------------
Running experiments:  71%|||||||   | 12/17 [00:00<00:00, 111.89it/s]
        10 |     0.000011 |             128 |   0.3201
        50 |     0.000059 |             344 |   0.0475
       100 |     0.000105 |             368 |   0.1762
       250 |     0.000164 |             440 |   0.0868
       500 |     0.000345 |             616 |   0.0286
       650 |     0.000751 |             688 |   0.3411
       800 |     0.000682 |             784 |   0.1919
      1000 |     0.000911 |             904 |   0.1009
      1250 |     0.000882 |           1,024 |   0.1920
      1500 |     0.001041 |           1,168 |   0.1356
      2000 |     0.001342 |           1,456 |   0.0324
      2500 |     0.001847 |           1,720 |   0.1915
      3000 |     0.001596 |           2,008 |   0.1330
      4000 |     0.002230 |           2,560 |   0.1644
      5000 |     0.002857 |         151,135 |   0.2155
Running experiments: 100%|||||||||||| 17/17 [00:00<00:00, 48.93it/s]
      7500 |     0.006648 |           4,504 |   0.2340
     10000 |     0.007765 |         153,560 |   0.0948
```

Performance Graphs

The following figure presents graphical representations of:

- Execution Time vs Input Size
- Memory Usage vs Input Size
- Coefficient of Variation (CV) vs Input Size

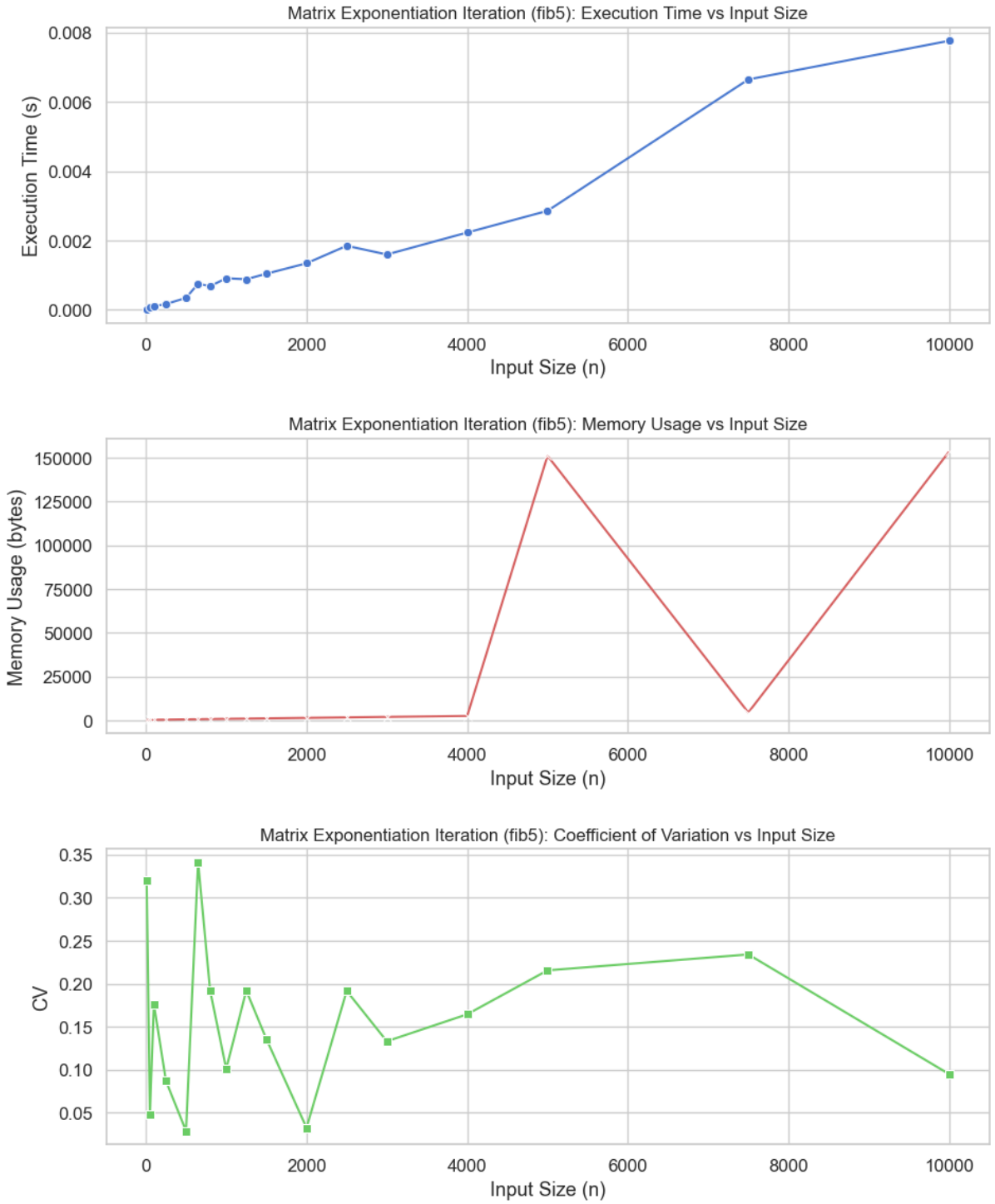

Figure 2.6: Performance Metrics for Matrix Exponentiation Iteration (`fib5`)

Performance Analysis

The performance analysis of the `fib5` algorithm reveals:

- **Execution Time:** The execution time grows linearly with the input size, showing stable performance even for larger inputs. For example, the execution time remains under 0.008 seconds for input sizes up to 10,000.

- **Memory Usage:** Memory consumption remains low for smaller inputs and increases moderately with larger inputs. The memory usage spikes at certain input sizes due to additional overhead from performance measurement tools.
- **Coefficient of Variation (CV):** The CV values fluctuate across input sizes but generally remain below 0.35, indicating reliable performance consistency.
- **Overall Efficiency:** The `fib5` algorithm demonstrates high efficiency due to its linear time complexity and minimal memory footprint.
- **Concepts:** Regarding algorithm concepts, `fib5` illustrates iterative matrix algebra, top-down dynamic programming characteristics, and linear time complexity. The algorithm uses the power of a specific 2x2 matrix to represent Fibonacci sequences. The iteration over this matrix corresponds to each Fibonacci transition step, resulting in $O(n)$ time complexity with constant-time arithmetic. With non-constant time arithmetic, additions range in complexity from $A(\lg F_1)$ to $A(\lg F_n)$, leading to $O(n^2)$ time complexity in bit operations. Space complexity in this case becomes quadratic in the number of bits.

2.5.2 Matrix Exponentiation via Repeated Squaring - Recursive

The `fib52` algorithm computes Fibonacci numbers using matrix exponentiation optimized by repeated squaring. This recursive approach reduces the time complexity from linear to logarithmic by halving the exponent at each recursive step.

Python Implementation

The following Python code implements the recursive matrix exponentiation Fibonacci algorithm (`fib52`) using repeated squaring:

```
# multiply 2x2 matrices m1 and m2 and return the product.
def mat_mul(m1, m2):
    m = [[0, 0], [0, 0]]
    m[0][0] = m1[0][0] * m2[0][0] + m1[0][1] * m2[1][0]
    m[0][1] = m1[0][0] * m2[0][1] + m1[0][1] * m2[1][1]
    m[1][0] = m1[1][0] * m2[0][0] + m1[1][1] * m2[1][0]
    m[1][1] = m1[1][0] * m2[0][1] + m1[1][1] * m2[1][1]
    return m

# recursive exponentiation using repeated squaring
def mat_pow_opt_recur(m, n):
    r = [[1, 0], [0, 1]]  # the identity matrix
    if n > 1:
        r = mat_pow_opt_recur(m, n >> 1)  # divide n by 2
        r = mat_mul(r, r)  # square the result
        if n % 2 == 1:  # if n is odd, multiply by m
            r = mat_mul(r, m)
    return r

def fib52(n):
```

```
    if n == 0:
        return 0
    else:
        m = mat_pow_opt_recur([[1, 1], [1, 0]], n - 1)
        return m[0][0]
```

Listing 2.13: Matrix Exponentiation via Repeated Squaring (Recursive) Fibonacci Algorithm (fib52)

Time Complexity Analysis

The time complexity of the `fib52` algorithm is $O(\log n)$ because the recursive repeated squaring technique reduces the number of multiplications by halving the exponent n in each recursive call. Since each multiplication of 2x2 matrices is constant-time, the overall time complexity remains logarithmic.

The space complexity is also $O(\log n)$ due to the recursion stack depth, which corresponds to the number of halving steps.

Performance Output

The following output shows the detailed performance metrics collected during the execution of the recursive matrix exponentiation Fibonacci algorithm (`fib52`). The results include execution time, memory consumption, and the coefficient of variation (CV) for various input sizes.

Listing 2.14: Performance Metrics for fib52

```
Analyzing Matrix Exponentiation via Repeated Squaring (Recursive)...
Input Size |     Time (s) |  Memory (bytes) |       CV
--------------------------------------------------------
Running experiments: 100%|||||||||||| 17/17 [00:00<00:00, 360.14it/s]
        10 |     0.000037 |             240 |   0.2543
        50 |     0.000053 |             432 |   0.3696
       100 |     0.000046 |             480 |   0.1625
       250 |     0.000069 |             560 |   0.1299
       500 |     0.000080 |             644 |   0.1673
       650 |     0.000064 |             640 |   0.1129
       800 |     0.000084 |             688 |   0.0866
      1000 |     0.000091 |             860 |   0.0923
      1250 |     0.000146 |             672 |   0.3017
      1500 |     0.000099 |             824 |   0.1040
      2000 |     0.000119 |           1,256 |   0.1169
      2500 |     0.000112 |             824 |   0.1406
      3000 |     0.000138 |           1,220 |   0.3484
      4000 |     0.000168 |           2,084 |   0.1136
      5000 |     0.000189 |           1,184 |   0.0986
      7500 |     0.000227 |           3,272 |   0.0899
     10000 |     0.000251 |           1,940 |   0.0704
```

Performance Graphs

The following figure presents graphical representations of:

- Execution Time vs Input Size
- Memory Usage vs Input Size
- Coefficient of Variation (CV) vs Input Size

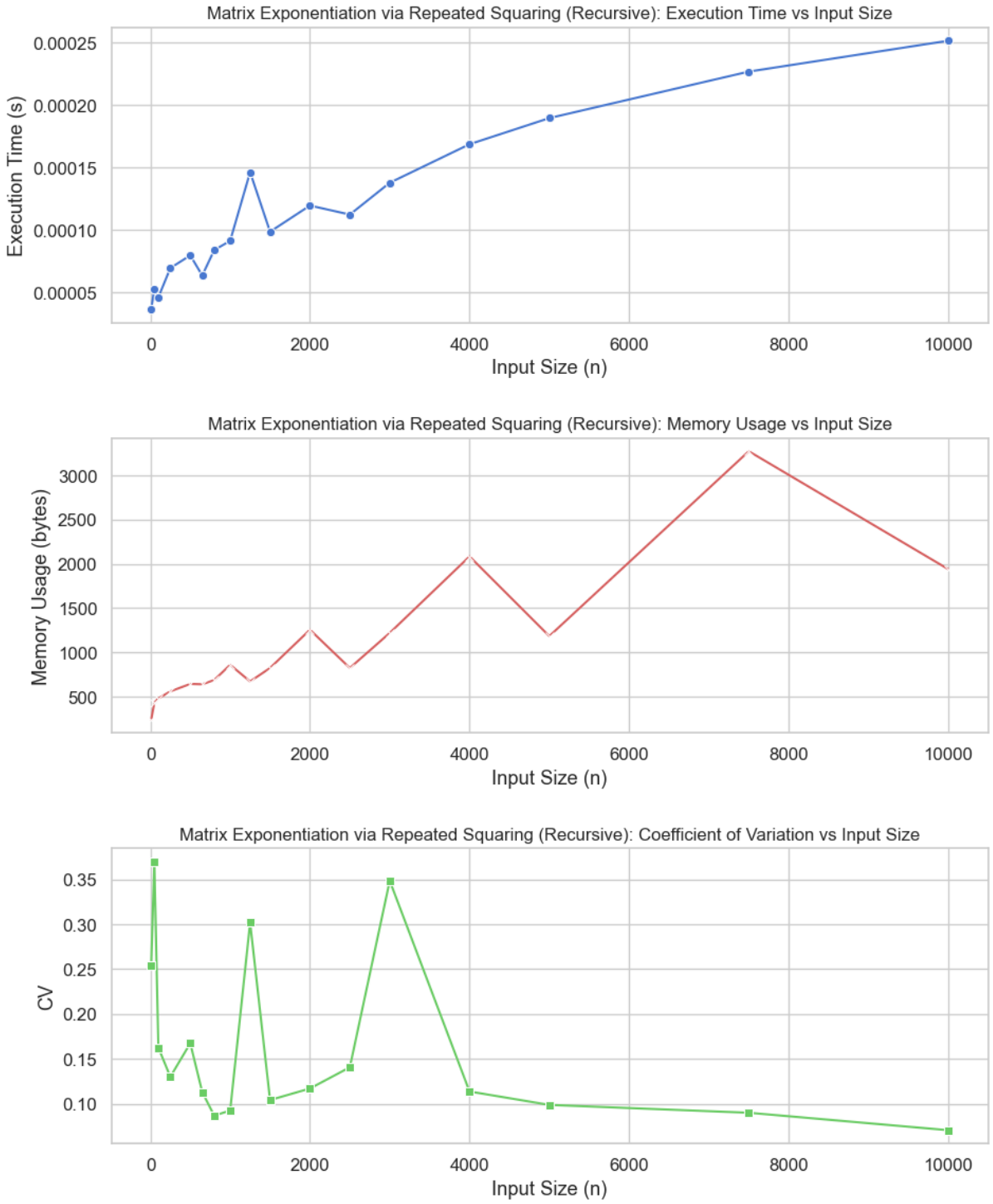

Figure 2.7: Performance Metrics for Matrix Exponentiation via Repeated Squaring (Recursive) (`fib52`)

Performance Analysis

The performance analysis of the `fib52` algorithm reveals:

- **Execution Time:** The execution time exhibits a logarithmic growth pattern, remaining consistently below 0.0003 seconds even for inputs as large as 10,000.
- **Memory Usage:** Memory consumption remains low and stable across input sizes, with minor fluctuations due to recursion overhead.
- **Coefficient of Variation (CV):** CV values indicate stable performance consistency, generally remaining under 0.35 across all tested input sizes.
- **Overall Efficiency:** The `fib52` algorithm demonstrates high efficiency due to its $O(\log n)$ time complexity and minimal memory requirements.
- **Concepts:** The `fib52` algorithm demonstrates (simple) matrix algebra combined with recursion and repeated squaring. Repeated squaring significantly reduces the computational complexity from linear to logarithmic. With constant-time arithmetic, the time complexity is $O(\log n)$, and the space complexity is $O(\log n)$ due to recursion depth. If non-constant time arithmetic is considered, the complexity could reach $O(n)$ in bit operations, with linear space complexity relative to the number of bits.

2.5.3 Matrix Exponentiation via Repeated Squaring - Iterative

The `fib53` algorithm computes Fibonacci numbers using an iterative approach to matrix exponentiation via repeated squaring. Unlike the recursive approach (`fib52`), this version iteratively processes the binary representation of the exponent. This, also, reduces the time complexity from linear to logarithmic (in comparison to `fib5`) by performing squaring and conditional multiplications based on the binary decomposition of the exponent.

Python Implementation

The following Python code implements the iterative matrix exponentiation Fibonacci algorithm (`fib53`) using repeated squaring:

```
def mat_pow_opt_iter(m, n):
    n_bits = bin(n)[2:]  # convert n to binary
    r = [[1, 0], [0, 1]]  # identity matrix
    for b in n_bits:
        r = mat_mul(r, r)  # square the result
        if b == '1':  # if the current bit is 1, multiply by m
            r = mat_mul(r, m)
    return r

def fib53(n):
    if n == 0:
        return 0
    else:
        m = mat_pow_opt_iter([[1, 1], [1, 0]], n - 1)
        return m[0][0]
```

Listing 2.15: Matrix Exponentiation via Repeated Squaring (Iterative) Fibonacci Algorithm (fib53)

Time Complexity Analysis

The time complexity of the `fib53` algorithm is $O(\log n)$ because it utilizes repeated squaring in an iterative manner. The binary decomposition of the exponent ensures that the number of multiplications is proportional to the number of bits in n. Each iteration handles one bit of n and performs constant-time 2x2 matrix multiplications.

The space complexity is $O(\log n)$ because the binary representation of the exponent is stored in memory, with its length proportional to the number of bits in n. Although the matrix operations themselves require only constant space, storing the entire binary decomposition introduces a logarithmic space requirement.

Performance Output

The following output shows the detailed performance metrics collected during the execution of the iterative matrix exponentiation Fibonacci algorithm (`fib53`). The results include execution time, memory consumption, and the coefficient of variation (CV) for various input sizes.

Listing 2.16: Performance Metrics for fib53

```
Analyzing Matrix Exponentiation via Repeated Squaring - Iterative (fib53)...
Input Size |     Time (s) |  Memory (bytes) |       CV
--------------------------------------------------
Running experiments: 100%|||||||||||| 17/17 [00:00<00:00, 251.79it/s]
        10 |     0.000042 |             237 |   0.4365
        50 |     0.000046 |             575 |   0.2581
       100 |     0.000053 |             624 |   0.1553
       250 |     0.000071 |             709 |   0.1564
       500 |     0.000110 |             958 |   0.1865
       650 |     0.000080 |           1,067 |   0.1114
       800 |     0.000098 |           1,211 |   0.1136
      1000 |     0.000168 |           1,391 |   0.0322
      1250 |     0.000139 |           1,572 |   0.1299
      1500 |     0.000152 |           1,788 |   0.2144
      2000 |     0.000156 |           2,220 |   0.1026
      2500 |     0.000280 |           2,617 |   0.2584
      3000 |     0.000227 |           3,049 |   0.1527
      4000 |     0.000405 |           3,877 |   0.0493
      5000 |     0.000495 |           4,706 |   0.0375
      7500 |     0.000559 |           6,794 |   0.0908
     10000 |     0.000802 |           8,883 |   0.0273
```

Performance Graphs

The following figure presents graphical representations of:

- Execution Time vs Input Size
- Memory Usage vs Input Size

- Coefficient of Variation (CV) vs Input Size

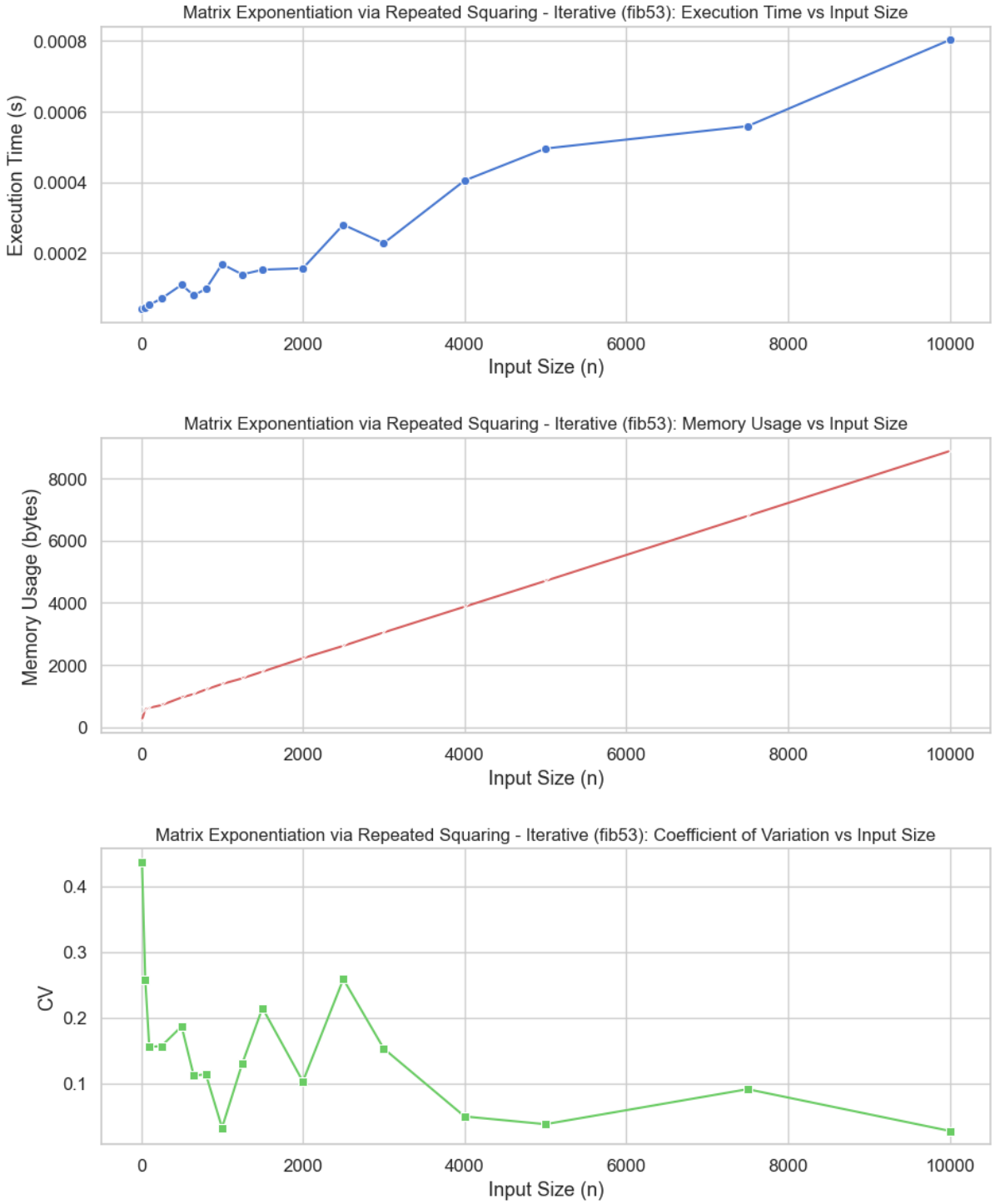

Figure 2.8: Performance Metrics for Matrix Exponentiation via Repeated Squaring (Iterative) (`fib53`)

Performance Analysis

Performance Analysis

The performance analysis of the `fib53` algorithm reveals:

- **Execution Time:** The execution time follows a logarithmic growth pattern, remaining consistently below 0.001 seconds even for inputs up to 10,000.
- **Memory Usage:** Memory usage scales steadily with input size due to the binary decomposition of the exponent, which introduces a logarithmic memory requirement. The absence of recursion overhead contributes to efficient memory utilization.
- **Coefficient of Variation (CV):** The CV values demonstrate consistent performance, generally remaining below 0.26 and decreasing with larger input sizes.
- **Overall Efficiency:** The `fib53` algorithm demonstrates high efficiency with $O(\log n)$ time complexity and $O(\log n)$ space complexity, making it highly scalable and performant.
- **Concepts:** The `fib53` algorithm demonstrates (simple) matrix algebra combined with iterative repeated squaring. Repeated squaring reduces time complexity from linear to logarithmic by utilizing binary exponentiation. With constant-time arithmetic, the time complexity is $O(\log n)$ and space complexity is $O(\log n)$ due to the binary representation of the exponent. In the case of non-constant time arithmetic, the complexity in bit operations is $O(n)$, with space complexity linear in the number of bits.

2.6 Memoization and Fast Doubling

The `fib6` algorithm computes Fibonacci numbers using a combination of memoization and fast doubling techniques (1.10, 1.11). Fast doubling leverages specific mathematical identities to compute Fibonacci numbers in $O(\log n)$ time by dividing the problem size in half at each step. Memoization ensures that only the necessary intermediate Fibonacci numbers are computed and stored, avoiding redundant calculations. This approach performs bottom-up processing, utilizing a queue data structure to manage the indices that need to be computed.

Python Implementation

The following Python code implements the Fibonacci algorithm (`fib6`) using memoization and fast doubling:

```
1  def fib6(n):
2      if n < 0:
3          raise ValueError("Negative Fibonacci numbers are not
               supported")
4
5      F = {}
6
7      # Queue for tracking necessary Fibonacci indices
8      qinx = []
9      qinx.append(n)
10     F[n] = -1  # Mark value for computation
11
```

```
    # Find all required indices using fast doubling
    while qinx:
        k = qinx.pop() >> 1  # Integer division by 2
        if k not in F:
            F[k] = -1
            qinx.append(k)
        if (k + 1) not in F:
            F[k + 1] = -1
            qinx.append(k + 1)

    # Base cases
    F[0], F[1], F[2] = 0, 1, 1

    # Compute Fibonacci using fast doubling
    keys_sorted = sorted(F.keys())  # Get sorted required keys
    for k in keys_sorted[3:]:
        k2 = k >> 1
        f1, f2 = F[k2], F[k2 + 1]
        if k % 2 == 0:
            F[k] = 2 * f2 * f1 - f1 * f1
        else:
            F[k] = f2 * f2 + f1 * f1

    return F[n]
```

Listing 2.17: Memoization and Fast Doubling Fibonacci Algorithm (fib6)

Time Complexity Analysis

The time complexity of the `fib6` algorithm is $O(\log n)$ due to the fast doubling method. At each step, the exponent is divided by two, ensuring logarithmic performance. Memoization further enhances efficiency by ensuring each intermediate result is computed only once.

The space complexity is also $O(\log n)$ because the memoization dictionary stores only the necessary indices required for computing the final Fibonacci number. The use of a queue data structure for index tracking also contributes to the logarithmic space requirement.

Performance Output

The following output shows the detailed performance metrics collected during the execution of the memoization and fast doubling Fibonacci algorithm (`fib6`). The results include execution time, memory consumption, and the coefficient of variation (CV) for various input sizes.

Listing 2.18: Performance Metrics for fib6

```
Analyzing Memoization and Fast Doubling...
Input Size |     Time (s) |  Memory (bytes) |       CV
--------------------------------------------------------
Running experiments: 100%|||||||||||| 17/17 [00:00<00:00, 417.62it/s]
        10 |     0.000029 |             480 |   0.6884
```

50	0.000038	1,032	0.3492
100	0.000042	1,152	0.2606
250	0.000054	1,448	0.1915
500	0.000064	2,188	0.2231
650	0.000074	2,440	0.1807
800	0.000066	2,372	0.2083
1000	0.000076	2,608	0.1969
1250	0.000136	2,956	0.0706
1500	0.000116	3,008	0.3076
2000	0.000093	3,232	0.1793
2500	0.000131	3,676	0.1863
3000	0.000135	3,848	0.2877
4000	0.000131	4,308	0.1654
5000	0.000159	4,984	0.1309
7500	0.000358	6,940	0.1230
10000	0.000313	8,040	0.2130

Performance Graphs

The following figure presents graphical representations of:

- Execution Time vs Input Size
- Memory Usage vs Input Size
- Coefficient of Variation (CV) vs Input Size

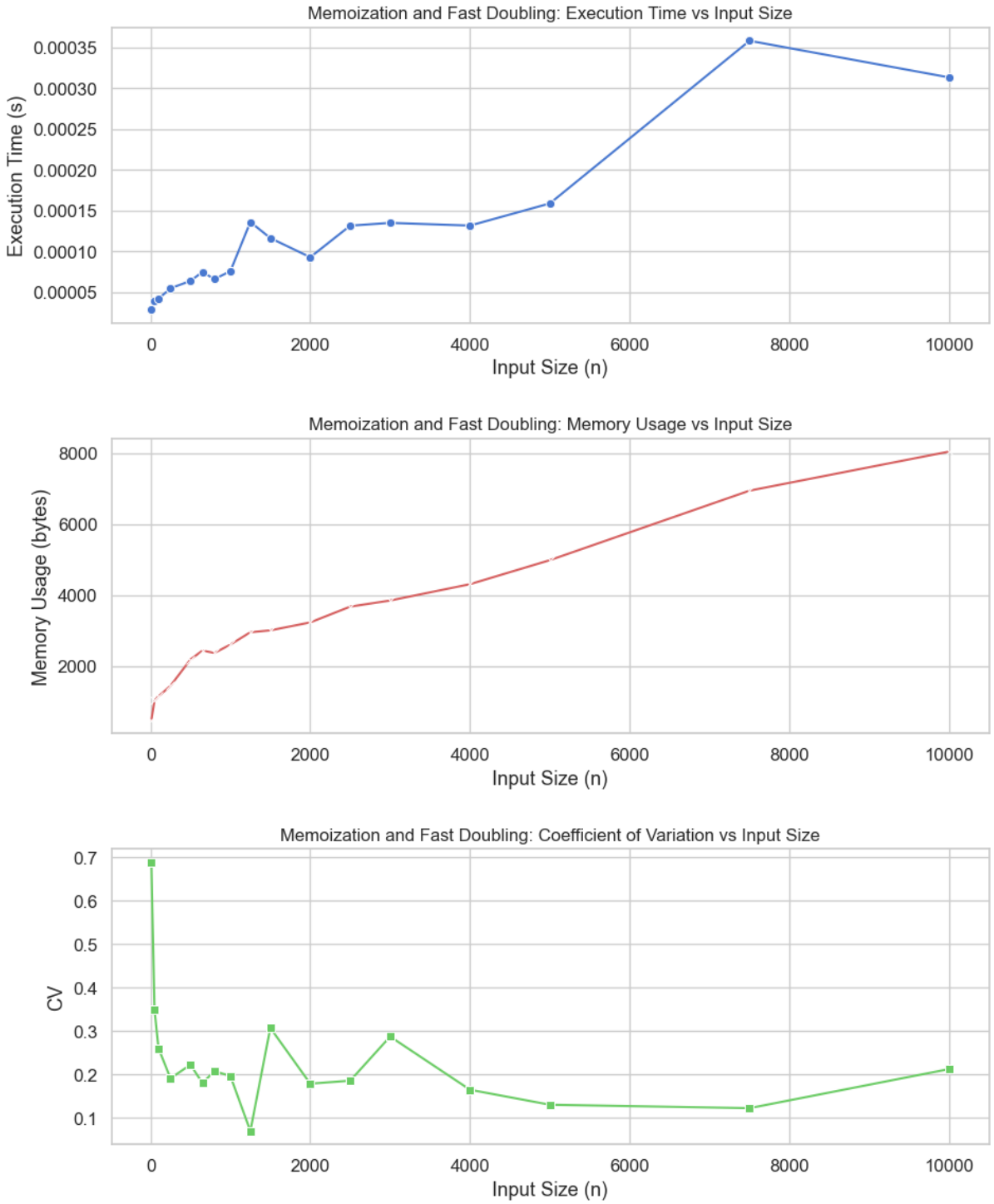

Figure 2.9: Performance Metrics for Memoization and Fast Doubling (`fib6`)

Performance Analysis

The performance analysis of the `fib6` algorithm reveals:

- **Execution Time:** The execution time follows a logarithmic growth pattern, remaining consistently below 0.0004 seconds even for input sizes up to 10,000.
- **Memory Usage:** Memory consumption scales logarithmically with input size.

The memoization dictionary and queue structure for fast doubling contribute to this behavior.

- **Coefficient of Variation (CV):** The CV values decrease with increasing input sizes, showing stable and consistent performance, with values generally under 0.31.
- **Overall Efficiency:** The `fib6` algorithm exhibits high efficiency with $O(\log n)$ time complexity and $O(\log n)$ space complexity, making it well-suited for computing large Fibonacci numbers.
- **Concepts:** The `fib6` algorithm demonstrates fast doubling techniques integrated with memoization. Fast doubling reduces time complexity by halving the exponent at each step, while memoization ensures no redundant calculations. With constant-time arithmetic, the time complexity remains $O(\log n)$ and space complexity $O(\log n)$. For non-constant time arithmetic, the bit operation complexity becomes $O(n)$, with space complexity linear in the number of bits. This algorithm highlights top-down processing using dynamic programming and efficient data structure usage to minimize unnecessary computations.

Chapter 3

Conclusion

The empirical analysis of the algorithms stipulated in this report for determining the N-th term of the Fibonacci sequence was conducted on an **Asus ZenBook 14**, equipped with an **Intel Core i7 8th Gen processor (1.8 GHz, 4 cores, 8 threads)**, **16GB RAM**, and **512GB SSD storage**. The experimental analysis was performed using **Visual Studio Code** with **Jupyter Notebook**, running locally on the **Python kernel**.

For each algorithm, runtime was measured in seconds using the `timeit` module in Python. The reported runtimes represent average execution times, accompanied by the **standard deviation**. The **coefficient of variation (CV)** was calculated as the ratio of the standard deviation to the average runtime, providing insights into the consistency of each algorithm's performance.

The performance of the algorithms was evaluated based on three key metrics:

- **Time Complexity**
- **Space Complexity**
- **Coefficient of Variation (CV)**

The results, grouped by their efficiency and performance characteristics, are presented in the following table:

Algorithm	**Time Complexity**	**Space Complexity**	**Coefficient of Variation (CV)**
fib1	$\mathcal{O}(\varphi^n)$	$\mathcal{O}(n)$	High
fib2	$\mathcal{O}(n)$	$\mathcal{O}(n)$	Low
fib3	$\mathcal{O}(n)$	$\mathcal{O}(1)$	Low
fib4	$\mathcal{O}(1)$	$\mathcal{O}(1)$	Medium
fib42	$\mathcal{O}(1)$	$\mathcal{O}(1)$	Medium
fib5	$\mathcal{O}(n)$	$\mathcal{O}(1)$	Low
fib52	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	Low
fib53	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	Very Low
fib6	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	Very Low

Table 3.1: Performance Metrics of Fibonacci Algorithms

Key Insights

- **Naive Recursion (fib1):** Highly inefficient for large inputs due to exponential time complexity with high CV.
- **Memoized Recursion (fib2):** Significant improvement using memoization, reducing time complexity to $\mathcal{O}(n)$.
- **Iterative with Constant Storage (fib3):** Best trade-off in linear time algorithms with very low CV.
- **Closed-Form Solutions (fib4 & fib42):** Constant-time performance but limited by precision issues for large inputs.
- **Matrix Exponentiation (fib5, fib52, fib53):** Efficient with $\mathcal{O}(\log n)$ time complexity, especially fib53 for memory efficiency.
- **Memoization and Fast Doubling (fib6):** Most optimal for large inputs with minimal memory use and very low CV.

Final Remarks

Based on the experimental results, **Memoization and Fast Doubling (fib6)** and **Matrix Exponentiation via Repeated Squaring - Iterative (fib53)** emerged as the most optimal algorithms for computing large Fibonacci numbers. These methods combine computational speed, memory efficiency, and performance consistency, making them suitable for practical applications requiring rapid and reliable Fibonacci computations.

The **fib6** algorithm stands out due to its logarithmic time complexity and minimal memory usage, making it highly effective for large input values without sacrificing performance. On the other hand, **fib53** offers similar performance characteristics while employing an iterative approach that further optimizes memory usage by avoiding recursion overhead.

Additionally, the experimental findings underline the practical limitations of closed-form solutions (**fib4** and **fib42**), which, despite their constant time complexity, are constrained by floating-point precision issues for large inputs. Meanwhile, iterative and memoized recursive algorithms (**fib2** and **fib3**) demonstrate stable and predictable performance, though with linear growth in execution time.

Overall, the analysis highlights that *logarithmic time complexity algorithms* such as **fib6** and **fib53** provide the best trade-off between computational efficiency and accuracy. Their reliable performance across various input sizes makes them ideal for applications in fields where Fibonacci number computations are essential.