Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Laboratory Work 1:

Empirical Analysis of Five Algorithms for the Fibonacci N-th Term

Elaborated:
st. gr. FAF-243

Soimu Ionut

Verified:

asist. univ. Fistic Cristofor

Chișinău – 2026

Contents

1 Algorithm Analysis

1.1 Objective

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

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

An alternative to mathematical analysis of complexity is empirical analysis. This may be useful for: obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer. In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established.
2. Choose the efficiency metric to be used (number of executions of an operation (s) or time execution of all or part of the algorithm.
3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties).
4. The algorithm is implemented in a programming language.
5. Generating multiple sets of input data.
6. Run the program for each input data set.
7. The obtained data are analyzed.

The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behavior of the implementation of an algorithm then execution time is appropriate. After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

1.4 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 five distinct algorithms empirically.

1.5 Comparison Metric

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

1.6 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 capture low-scale behavior, while the second series will have a bigger scope to be able to compare the algorithms between themselves (501, 631, 794, 1000, 1259, 1585, 1995, 2512, 3162, 3981, 5012, 6310, 7943, 10000, 12589, 15849).

2 Implementation

All five algorithms will be implemented in their naive form in Python and analyzed empirically based on the time required for their completion. While the general trend of the results may besimilar to other experimental observations, the particular efficiency in rapport with input will vary depending on memory of the device used.

The measurement error margin is influenced by OS scheduling and interpreter overhead. For this reason, we focus on the overall trend rather than individual outliers, and we compare algorithms using the same input set.

2.1 Algorithm 1: Iterative Linear Method

Description: Computes Fibonacci numbers using a simple loop and constant space. Time complexity is $O(n)$ and space complexity is $O(1)$.

The algorithm keeps only the two most recent values. It represents the simplest efficient baseline and highlights the expected linear growth when n increases.

Pseudocode:

```
Fibonacci(n):
    a <- 0
    b <- 1
    repeat n times:
        a, b <- b, a + b
    return a
```

Python Preview:

```
def fib_iterative(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a
```

Results:

Table 1: Iterative Results

n	time (s)
5	0.000002
7	0.000002
10	0.000003
12	0.000002
15	0.000003
17	0.000003
20	0.000003
22	0.000003
25	0.000003
27	0.000003
30	0.000003
32	0.000003
35	0.000003
37	0.000002
40	0.000004
42	0.000004
45	0.000004
501	0.000041
631	0.000051
794	0.000068
1000	0.000089
1259	0.000116
1585	0.000161
1995	0.000210
2512	0.000286
3162	0.000328
3981	0.000452
5012	0.000649
6310	0.001062
7943	0.001517
10000	0.002203
12589	0.003007
15849	0.004528

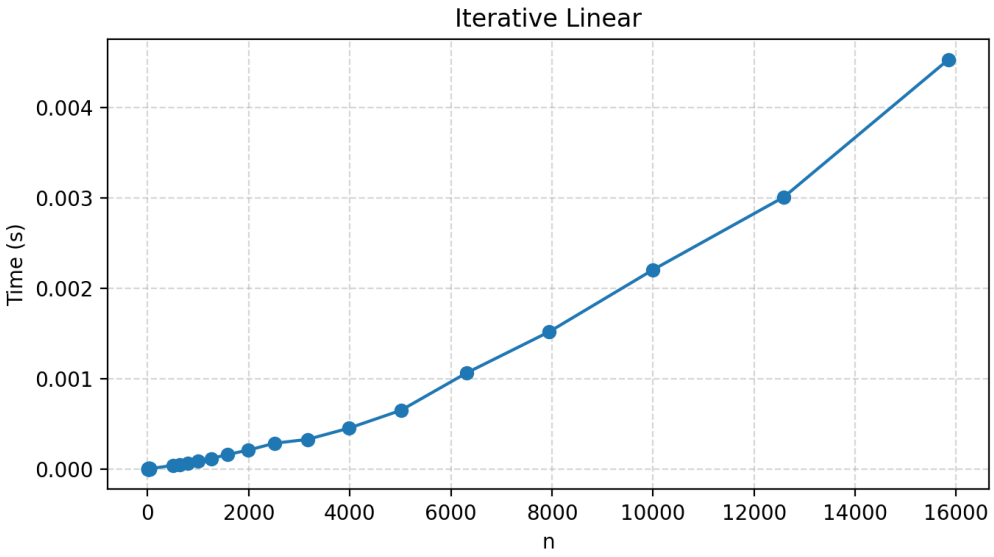

Figure 1: Iterative Linear Method

The timing curve shows steady linear growth with small fluctuations. This method is stable and avoids recursion overhead, making it a reliable reference for the other approaches.

2.2 Algorithm 2: Memoized Recursion

Description: Uses top-down recursion with memoization. Each term is computed once, yielding $O(n)$ time and $O(n)$ space.

The memoized approach trades extra memory for speed by caching intermediate results. In practice, it behaves similarly to the iterative method but with higher constant overhead because of dictionary lookups.

Pseudocode:

```
Fibonacci(n):
    memo <- empty map
    function solve(k):
        if k < 2: return k
        if k in memo: return memo[k]
        memo[k] <- solve(k-1) + solve(k-2)
        return memo[k]
    return solve(n)
```

Python Preview:

```
def fib_memoized(n):
    if n < 2:
        return n

    memo = {0: 0, 1: 1}
    stack = [n]
    while stack:
        k = stack.pop()
        if k in memo:
            continue
```

```
        k1, k2 = k - 1, k - 2
        if k1 in memo and k2 in memo:
            memo[k] = memo[k1] + memo[k2]
            continue
        stack.append(k)
        if k1 not in memo:
            stack.append(k1)
        if k2 not in memo:
            stack.append(k2)
    return memo[n]
```

Results:

Table 2: Memoized Results

n	time (s)
5	0.000005
7	0.000005
10	0.000006
12	0.000006
15	0.000008
17	0.000008
20	0.000010
22	0.000010
25	0.000011
27	0.000011
30	0.000012
32	0.000013
35	0.000013
37	0.000014
40	0.000015
42	0.000017
45	0.000018
501	0.000228
631	0.000325
794	0.000359
1000	0.000474
1259	0.000620
1585	0.000870
1995	0.000968
2512	0.001338
3162	0.001910
3981	0.002121
5012	0.002702
6310	0.003562
7943	0.005104
10000	0.006451
12589	0.009568
15849	0.012363

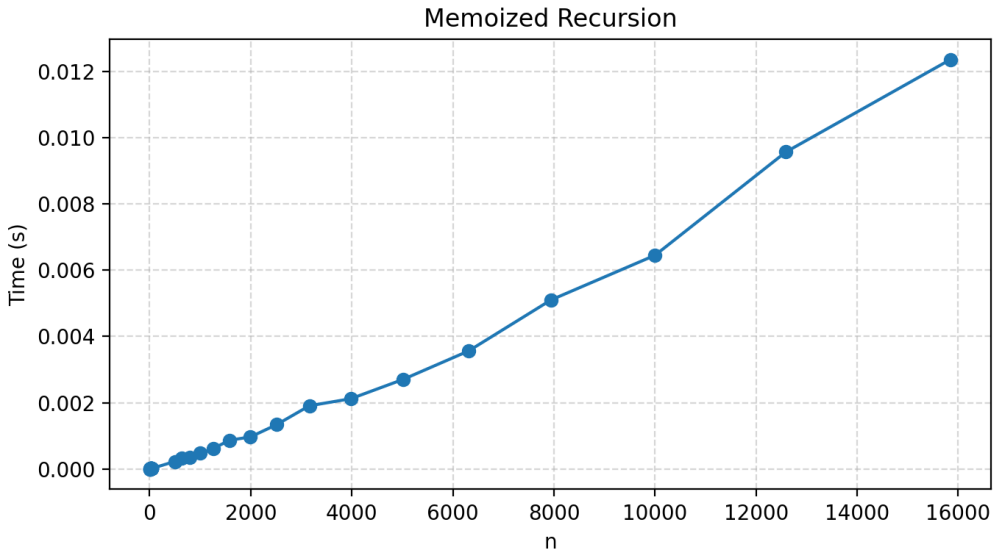

Figure 2: Memoized Recursion

The graph closely matches the iterative method, but the curve is slightly higher due to memo management. It remains linear and stable for the full input range.

2.3 Algorithm 3: Fast Doubling

Description: Uses doubling identities to compute $F(n)$ in $O(\log n)$ time and $O(\log n)$ space due to recursion depth.

This method reduces the problem size by half on each step and derives pairs $(F(k), F(k+1))$. It is efficient for large n because the number of multiplications grows logarithmically.

Pseudocode:

```
Fibonacci(n):
    function solve(k):
        if k == 0: return (0, 1)
        (a, b) <- solve(k // 2)
        c <- a * (2*b - a)
        d <- a*a + b*b
        if k is even: return (c, d)
        else: return (d, c + d)
    return solve(n).first
```

Python Preview:

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

```
11      return solve(n)[0]
```

Results:

Table 3: Fast Doubling Results

n	time (s)
5	0.000004
7	0.000004
10	0.000004
12	0.000004
15	0.000004
17	0.000004
20	0.000004
22	0.000004
25	0.000004
27	0.000004
30	0.000004
32	0.000004
35	0.000004
37	0.000004
40	0.000004
42	0.000004
45	0.000005
501	0.000007
631	0.000007
794	0.000008
1000	0.000008
1259	0.000010
1585	0.000009
1995	0.000010
2512	0.000011
3162	0.000014
3981	0.000017
5012	0.000021
6310	0.000030
7943	0.000037
10000	0.000053
12589	0.000081
15849	0.000104

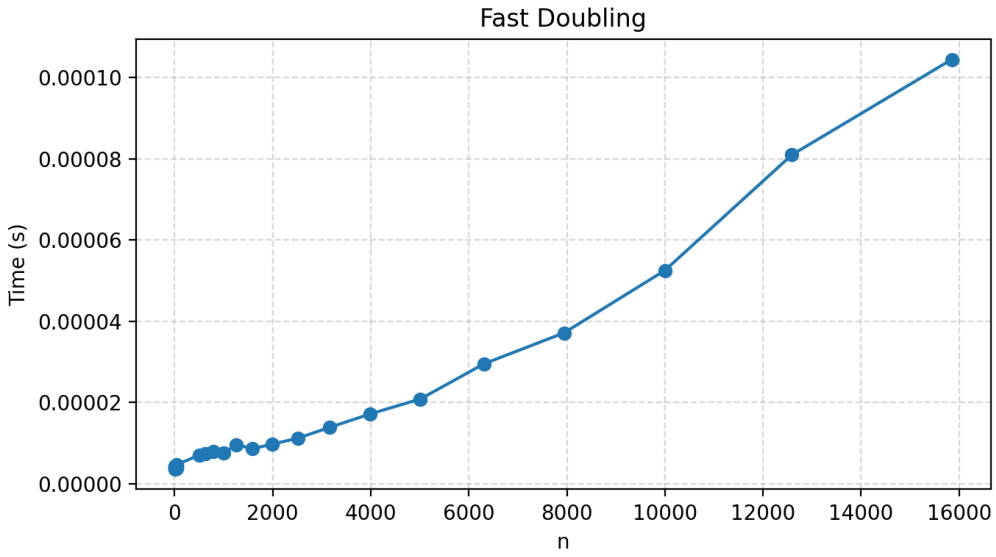

Figure 3: Fast Doubling Method

The timing curve shows only mild growth as n increases, reflecting the $O(\log n)$ complexity. Compared to linear methods, the advantage becomes more visible for the larger inputs.

2.4 Algorithm 4: Binomial Sum Formula

Description: Uses the combinatorial identity $F(n) = \sum_{k=0}^{\lfloor (n-1)/2 \rfloor} \binom{n-k-1}{k}$. The implementation updates terms iteratively in $O(n)$ time.

The terms are computed with integer arithmetic to avoid rounding errors. While the complexity is linear, each iteration involves several multiplications and divisions, leading to a larger constant factor.

Pseudocode:

```
Fibonacci(n):
    if n == 0: return 0
    max_k <- floor((n - 1) / 2)
    term <- 1
    sum <- 0
    for k from 0 to max_k:
        sum <- sum + term
        if k < max_k:
            term <- term * (n - 2*k - 1) * (n - 2*k - 2)
            term <- term / ((k + 1) * (n - k - 1))
    return sum
```

Python Preview:

```
def fib_binomial(n):
    if n == 0:
        return 0
    max_k = (n - 1) // 2
    term = 1
    total = 0
    for k in range(max_k + 1):
```

```
        total += term
        if k < max_k:
            numerator = (n - 2 * k - 1) * (n - 2 * k - 2)
            denominator = (k + 1) * (n - k - 1)
            term = term * numerator // denominator
    return total
```

Results:

Table 4: Binomial Results

n	time (s)
5	0.000004
7	0.000003
10	0.000003
12	0.000004
15	0.000006
17	0.000006
20	0.000005
22	0.000006
25	0.000006
27	0.000006
30	0.000007
32	0.000007
35	0.000008
37	0.000009
40	0.000009
42	0.000009
45	0.000010
501	0.000146
631	0.000190
794	0.000244
1000	0.000294
1259	0.000393
1585	0.000544
1995	0.000755
2512	0.001038
3162	0.001489
3981	0.002129
5012	0.003077
6310	0.004566
7943	0.006925
10000	0.008347
12589	0.010622
15849	0.012660

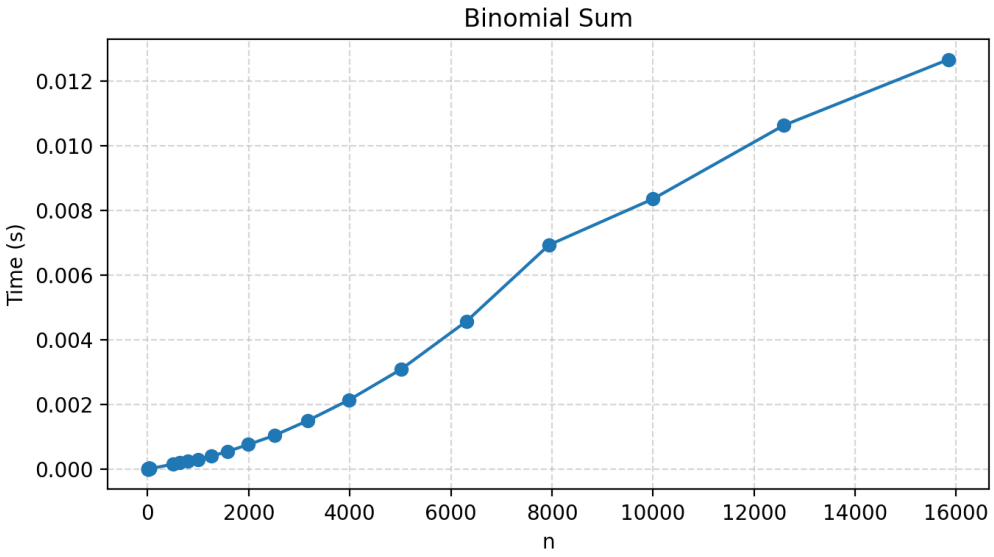

Figure 4: Binomial Sum Method

The resulting curve is linear but sits above the iterative and memoized curves. This matches the expectation that the extra arithmetic increases the runtime per step.

2.5 Algorithm 5: Fast Matrix Exponentiation

Description: Computes Q^n for $Q = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ using exponentiation by squaring. This method runs in $O(\log n)$ time.

Each squaring step multiplies two 2×2 matrices, which increases the constant cost per loop iteration. Nevertheless, the number of iterations is logarithmic in n.

Pseudocode:

```
Fibonacci(n):
    if n == 0: return 0
    result <- identity 2x2
    base <- [[1, 1], [1, 0]]
    while n > 0:
        if n is odd: result <- result * base
        base <- base * base
        n <- n // 2
    return result[0][1]
```

Python Preview:

```
def fib_matrix_fast(n):
    if n == 0:
        return 0

    def mat_mul(a, b):
        return [
            [a[0][0] * b[0][0] + a[0][1] * b[1][0],
             a[0][0] * b[0][1] + a[0][1] * b[1][1]],
            [a[1][0] * b[0][0] + a[1][1] * b[1][0],
```

```
            a[1][0] * b[0][1] + a[1][1] * b[1][1]],
        ]

    def mat_pow(p):
        result = [[1, 0], [0, 1]]
        base = [[1, 1], [1, 0]]
        while p > 0:
            if p & 1:
                result = mat_mul(result, base)
            base = mat_mul(base, base)
            p >>= 1
        return result

    return mat_pow(n)[0][1]
```

Results:

Table 5: Matrix Fast Results

n	time (s)
5	0.000004
7	0.000004
10	0.000005
12	0.000004
15	0.000005
17	0.000005
20	0.000005
22	0.000005
25	0.000005
27	0.000005
30	0.000005
32	0.000005
35	0.000006
37	0.000006
40	0.000005
42	0.000005
45	0.000006
501	0.000013
631	0.000012
794	0.000011
1000	0.000014
1259	0.000017
1585	0.000017
1995	0.000021
2512	0.000030
3162	0.000035
3981	0.000045
5012	0.000074
6310	0.000087
7943	0.000122
10000	0.000221
12589	0.000268
15849	0.000339

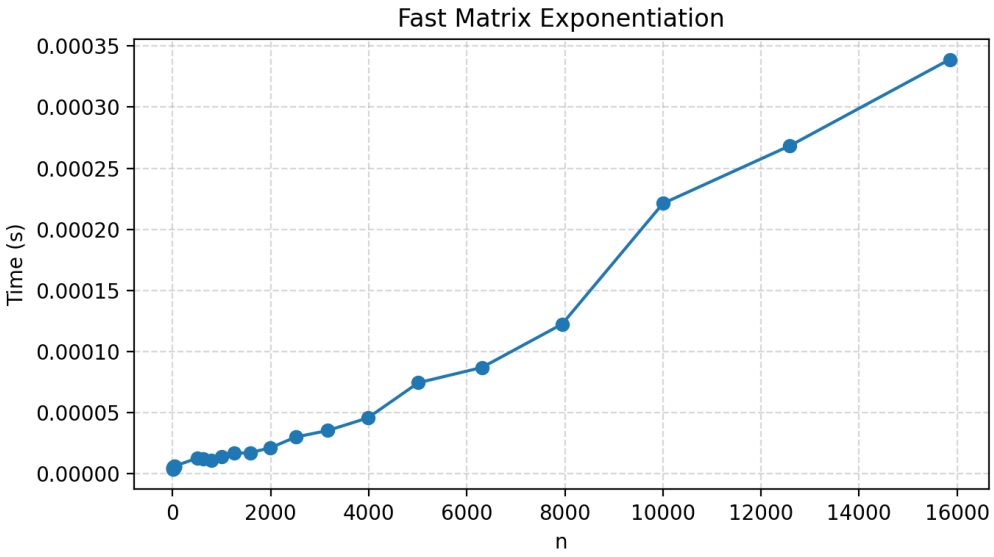

Figure 5: Fast Matrix Exponentiation

The curve is similar to fast doubling but slightly higher due to the extra matrix multiplication overhead. It remains efficient for large inputs and confirms the expected $O(\log n)$ behavior.

2.6 Combined Comparison

A combined plot provides a direct comparison between all algorithms. A logarithmic y-axis is used for readability.

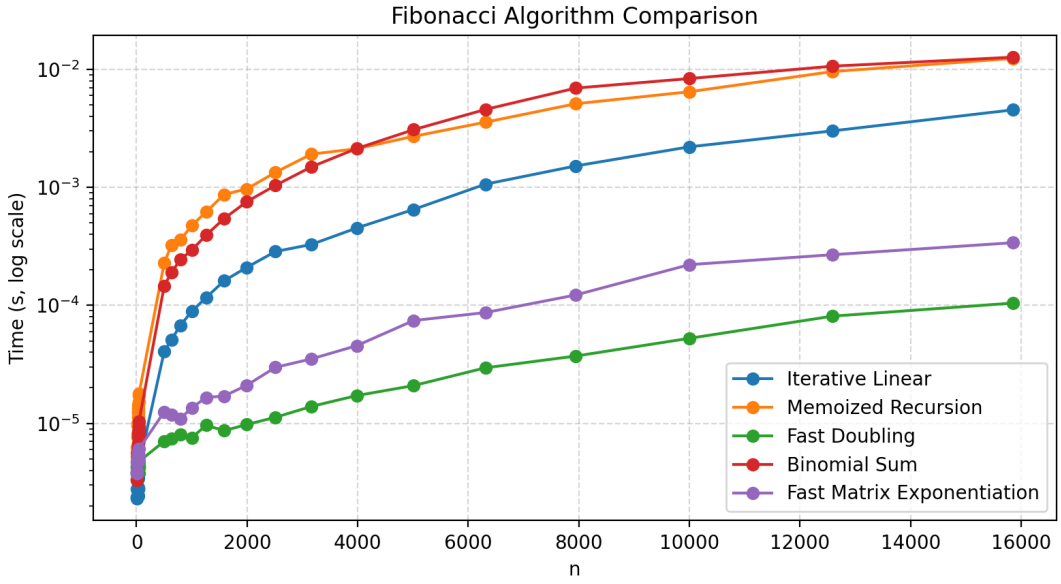

Figure 6: Comparison of All Methods (log scale)

3 Conclusion

For small inputs, the iterative linear method is the best practical choice due to its minimal overhead and constant memory usage. When a recursive style is desired while still avoiding exponential blowup, the memoized method is acceptable, but it uses more memory and is slightly slower than the iterative approach.

For medium to large inputs, the fast doubling method provides the best performance in this experiment because it achieves $O(\log n)$ growth with low constant factors. Fast matrix exponentiation is also efficient for large inputs, but its additional matrix multiplications make it slower than fast doubling in most cases. The binomial sum method is correct and linear, yet its heavier arithmetic makes it the least attractive among the linear-time methods, so it is best reserved for demonstrative or combinatorial contexts rather than performance-sensitive use.