Ministry of Education of Republic of Moldova
Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics

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

Verified:
Fistic Cristofor asist. univ.

Belih Dmitrii

Moldova, February 2025

Contents

1 Algorithm Analysis

1.1 Objective

Study and analyze different algorithm for determining Fibonaci n-th term.

1.2 Tasks

- Implement at least 3 algorithms for determining Fibonacci n-th term;
- Decide properties of input format that will be used for algorithm analysis;
- Decide the comparison metric for the algorithms;
- Analyze empirically the algorithms;
- Present the results of the obtained data;
- 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 defined as:

$$F(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F(n-1) + F(n-2), & n \geq 2 \end{cases} \tag{1.1}$$

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 7 native algorithms

empirically.

Several approaches exist for computing Fibonacci numbers, each with different time and space complexities.

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

Each algorithm receives two series of integers, representing the indices of Fibonacci terms to be retrieved:

- **Limited Scope Series:** Designed to accommodate the recursive method, this series contains smaller indices:

$$5, 7, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 37, 40, 42, 45$$

- **Extended Scope Series:** Intended for comparison of non-recursive algorithms, this series contains larger indices:

$$501, 631, 794, 1000, 1259, 1585, 1995, 2512, 3162, 3981, 5012, 6310, 7943, 10000, 12589, 15849$$

This format ensures an effective performance evaluation across different Fibonacci calculation methods.

2
Implementation

All seven algorithms will be implemented in their native 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.

2.1 Recursive Approach

In this section, we define a recursive function to calculate the n-th Fibonacci number. The function checks for invalid input and returns the Fibonacci number based on the base cases (0 and 1) or by recursively calling itself with reduced values of n. The driver program prints the 10th Fibonacci number.

2.1.1 Python Implementation

```
# Function for nth Fibonacci number
def Fibonacci(n):
    if n <= 0:
        print("Incorrect input")
    # First Fibonacci number is 0
    elif n == 1:
        return 0
    # Second Fibonacci number is 1
    elif n == 2:
        return 1
    else:
        return Fibonacci(n-1) + Fibonacci(n-2)
print(Fibonacci(10))
```

Listing 2.1: *Recursive Approach*

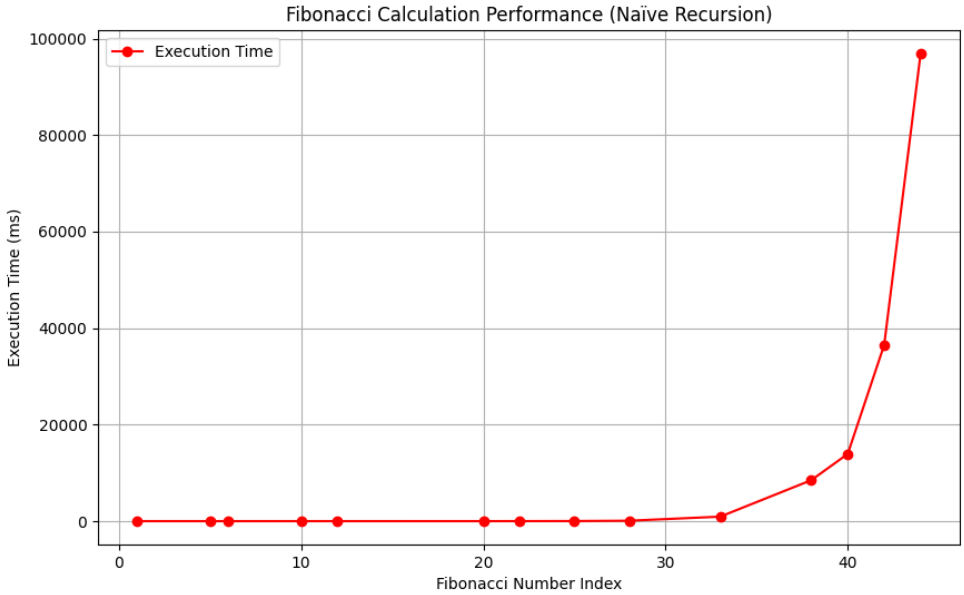

Figure 2.1: *Recursion graph*

Time Complexity: $O(2^N)$
Auxiliary Space: $O(N)$

2.2 Dynamic Programming Approach

The code defines a function fibonacci(n) that calculates the nth Fibonacci number using dynamic programming. It initializes a list FibArray with the first two Fibonacci numbers (0 and 1). The function checks if the Fibonacci number for n is already present in FibArray and returns it. Otherwise, it calculates the Fibonacci number recursively, stores it in FibArray for future use, and returns the calculated value.

2.2.1 Python Implementation

```
FibArray = [0, 1]
def fibonacci(n):
    if n<0:
        print("Incorrect input")
    elif n<= len(FibArray):
        return FibArray[n-1]
    else:
        temp_fib = fibonacci(n-1)+fibonacci(n-2)
        FibArray.append(temp_fib)
        return temp_fib
print(fibonacci(9))
```

Listing 2.2: *Dynamic Programming Approach*

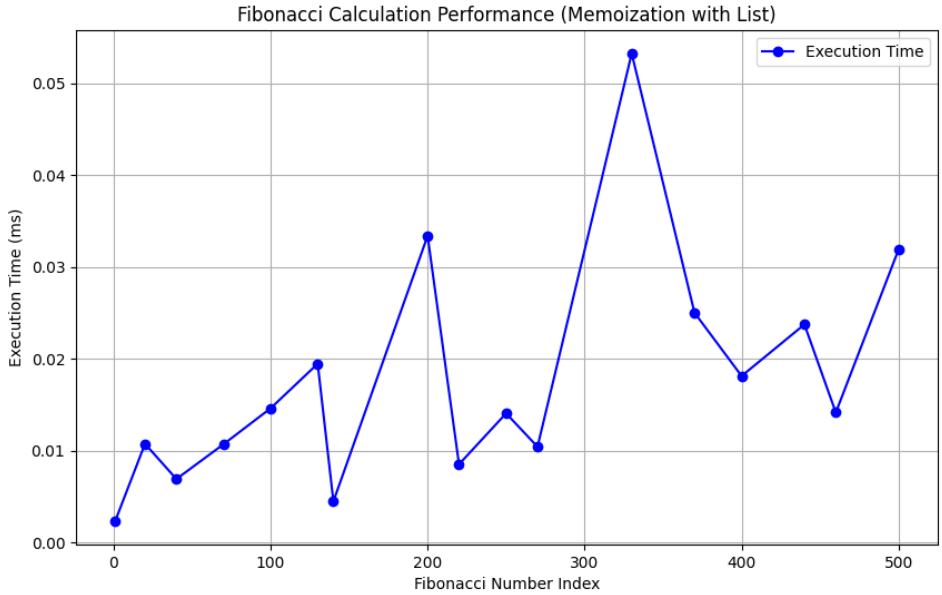

Figure 2.2: *Dynamic Programming graph*

Time Complexity: $O(N)$
Auxiliary Space: $O(N)$

2.3 Matrix Exponentiation

Fibonacci numbers can be computed in $O(\log n)$ time using matrix exponentiation:

$$\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} F(1) \\ F(0) \end{bmatrix} \tag{2.1}$$

This section describes an $O(n)$ approach to calculate the n-th Fibonacci number using matrix exponentiation. The method relies on the fact that if we multiply the matrix $M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ to itself n times (i.e., calculate M^n), the $(n+1)$-th Fibonacci number is obtained as the element at row 0 and column 0 in the resultant matrix.

2.3.1 Python Implementation

```python
def fib(n):
    F = [[1, 1],
         [1, 0]]
    if (n == 0):
        return 0
    power(F, n - 1)
    return F[0][0]

def multiply(F, M):
    x = (F[0][0] * M[0][0] + F[0][1] * M[1][0])
    y = (F[0][0] * M[0][1] + F[0][1] * M[1][1])
    z = (F[1][0] * M[0][0] + F[1][1] * M[1][0])
    w = (F[1][0] * M[0][1] + F[1][1] * M[1][1])
    F[0][0] = x
    F[0][1] = y
    F[1][0] = z
    F[1][1] = w

def power(F, n):
    M = [[1, 1], [1, 0]]
    for i in range(2, n + 1):
        multiply(F, M)

if __name__ == "__main__":
    n = 9
    print(fib(n))
```

Listing 2.3: *Matrix Exponentiation*

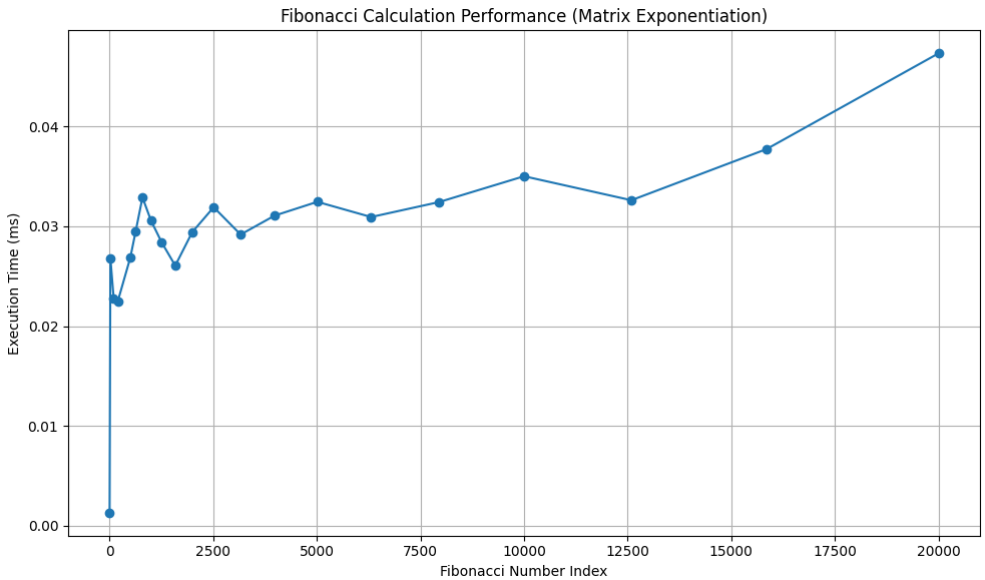

Figure 2.3: *Matrix Exponentiation graph*

2.4 Memoization Approach

In the previous approach, there is a lot of redundant calculation, as Fibonacci numbers are computed repeatedly. To avoid this, we can store the results of previously computed Fibonacci numbers in a memo table. This ensures that each Fibonacci number is computed only once, reducing the exponential time complexity of the naive approach $(O(2^n))$ to a more efficient $O(n)$ time complexity.

2.4.1 Python Implementation

```python
def nth_fibonacci_util(n, memo):
    if n <= 1:
        return n
    if memo[n] != -1:
        return memo[n]
    memo[n] = nth_fibonacci_util(n - 1, memo) + nth_fibonacci_util(n - 2, memo)
    return memo[n]

def nth_fibonacci(n):
    memo = [-1] * (n + 1)
    return nth_fibonacci_util(n, memo)

if __name__ == "__main__":
    n = 5
    result = nth_fibonacci(n)
    print(result)
```

Listing 2.4: *Memoization Approach*

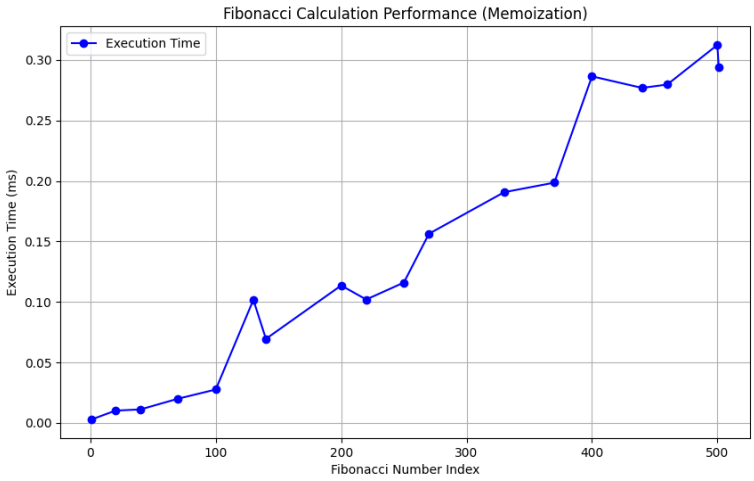

Figure 2.4: *Memoization graph*

Time Complexity: $O(n)$
Auxiliary Space: $O(n)$

2.5 Bottom-Up Approach

This approach uses dynamic programming to solve the Fibonacci problem by storing previously calculated Fibonacci numbers, avoiding the repeated calculations of the recursive approach. Instead of breaking down the problem recursively, it iteratively builds up the solution by calculating Fibonacci numbers from the bottom up.

2.5.1 Python Implementation

```python
def nth_fibonacci(n):
    if n <= 1:
        return n

    dp = [0] * (n + 1)
    dp[0] = 0
    dp[1] = 1

    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]

n = 5
result = nth_fibonacci(n)
print(result)
```

Listing 2.5: *Bottom-Up Approach*

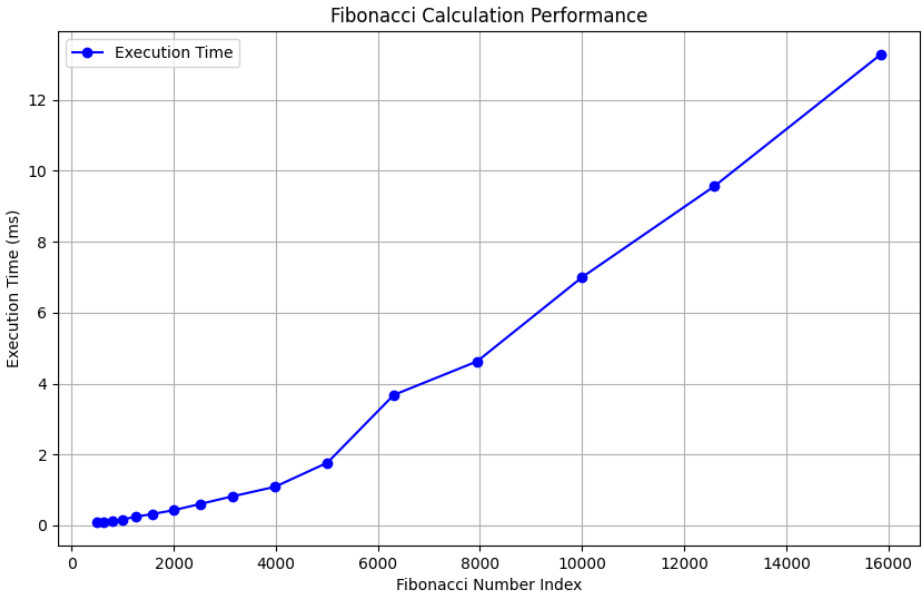

Figure 2.5: *Bottom-Up Approach graph*

Time Complexity: $O(n)$
Auxiliary Space: $O(n)$

2.6 Space Optimized Approach

This approach is just an optimization of the above iterative approach. Instead of using the extra array for storing the Fibonacci numbers, we can store the values in variables. We keep the previous two numbers only because that is all we need to get the next Fibonacci number in series.

2.6.1 Python Implementation

```python
def nth_fibonacci(n):
    if n <= 1:
        return n
    curr = 0
    prev1 = 1
    prev2 = 0
    for i in range(2, n + 1):
        curr = prev1 + prev2
        prev2 = prev1
        prev1 = curr
    return curr
n = 5
result = nth_fibonacci(n)
print(result)
```

Listing 2.6: *Space Optimized Approach*

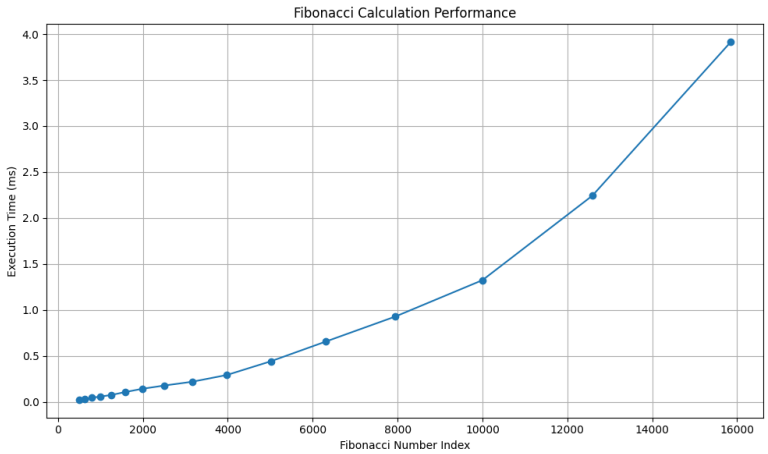

Figure 2.6: *Space Optimized graph*

Time Complexity: $O(n)$
Auxiliary Space: $O(1)$

2.7 Binet Formula Method

Binet's formula provides a mathematical expression to compute Fibonacci numbers directly:

$$f_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right] \tag{2.2}$$

where,

$$\alpha = \frac{1+\sqrt{5}}{2}$$
$$\beta = \frac{1-\sqrt{5}}{2}$$

2.7.1 Python Implementation

```
import numpy as np

a = np.arange(1, 11)
lengthA = len(a)

sqrtFive = np.sqrt(5)
alpha = (1 + sqrtFive) / 2
beta = (1 - sqrtFive) / 2

Fn = np.rint(((alpha ** a) - (beta ** a)) / (sqrtFive))
print("The first {} numbers of Fibonacci series are {} .".format(lengthA, Fn))
```

Listing 2.7: *Binet Formula Method*

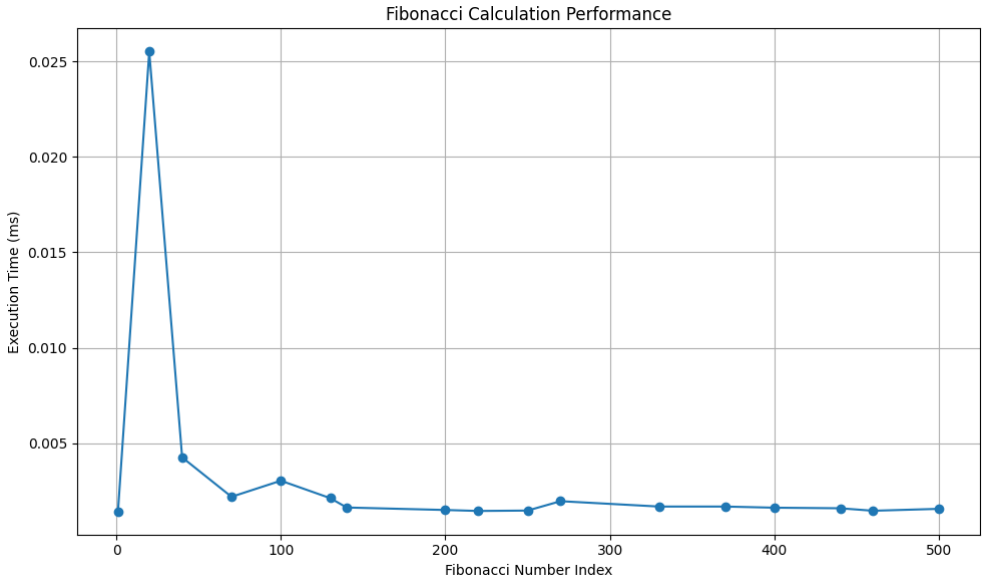

Figure 2.7: *Binet Formula Method graph*

3 Conclusion

Through empirical analysis, this paper has tested seven classes of methods for their efficiency in providing accurate results and their time complexity. The goal was to delineate the scopes within which each method could be effectively used and to identify potential improvements to enhance their feasibility.

Recursive Method

The recursive method is the simplest to implement but suffers from exponential time complexity $O(2^N)$. It is suitable for small values of n (up to 30) where computational resources and execution time are not significant concerns.

Dynamic Programming Method

The dynamic programming approach improves upon the recursive method by storing intermediate results, reducing the time complexity to $O(N)$. This method is efficient for larger values of n and ensures exact results without redundant calculations.

Matrix Exponentiation Method

Matrix exponentiation offers a more advanced approach with a time complexity of $O(\log N)$ when optimized. This method is highly efficient for very large values of n and provides exact results. It is particularly useful in scenarios where performance is critical.

Memoization Method

Memoization enhances the recursive approach by storing previously computed Fibonacci numbers, reducing the time complexity to $O(N)$. This method is efficient and avoids the pitfalls of redundant calculations, making it suitable for moderate values of n.

Bottom-Up Approach

The bottom-up approach iteratively builds the solution from the base cases, ensuring a time complexity of $O(N)$. It is space-efficient and avoids the overhead of recursive calls, making it a practical choice for larger values of n.

Space Optimized Approach

This method further optimizes the bottom-up approach by using only a constant amount of space, reducing the auxiliary space complexity to $O(1)$. It is ideal for scenarios where memory usage is a concern, while maintaining a time complexity of $O(N)$.

Binet Formula Method

Binet's formula provides a direct mathematical expression to compute Fibonacci numbers with an almost constant time complexity. However, it is susceptible to rounding errors, especially for large n, and is recommended for values up to 80. Verification of results is advised when using this method.

Summary

Each method has its strengths and weaknesses, making them suitable for different scenarios:

- **Recursive Method**: Best for small n due to its simplicity.
- **Dynamic Programming and Memoization**: Efficient for moderate n with exact results.
- **Matrix Exponentiation**: Optimal for very large n with logarithmic complexity.
- **Bottom-Up and Space Optimized Approaches**: Practical for larger n with linear complexity and reduced space usage.
- **Binet Formula**: Suitable for moderate n with constant time complexity, but requires result verification.

Future work could focus on further optimizing these methods, particularly in reducing space complexity and improving the accuracy of the Binet formula for larger values of n. Additionally, exploring hybrid approaches that combine the strengths of multiple methods could yield even more efficient solutions.