

**Laboratory work 1:
Study and Empirical Analysis of Algorithms for
Determining
Fibonacci N-th Term**

Elaborated:
st. gr. FAF-233

Cebotari Alexandru

Verified:
asist. univ.

Fiștic Cristofor

TABLE OF CONTENTS

ALGORITHM ANALYSIS.....	3
Objective	3
Tasks.....	3
Theoretical Notes:	3
Introduction:	4
Comparison Metric:.....	4
Input Format:.....	4
IMPLEMENTATION	5
Recursive:.....	5
Memoization:	6
Bottom-Up Dynamic Programming:	8
Matrix Exponentiation:	9
Binet Formula:.....	11
Fast Doubling:	13
CONCLUSION	15

ALGORITHM ANALYSIS

Objective

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

Tasks:

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

Theoretical Notes:

Empirical analysis complements mathematical complexity analysis by evaluating algorithm efficiency in practice. It helps:

- Estimate the complexity class of an algorithm.
- Compare the performance of multiple algorithms or implementations.
- Assess the efficiency of algorithms on specific hardware.

The steps for empirical analysis are as follows:

1. Define the analysis objective: Determine the purpose, such as comparing algorithm efficiency or determining time complexity.
2. Choose an efficiency metric: Common metrics include the number of operations or execution time. For this lab, we focus on execution time.
3. Set input data properties: Specify the size or characteristics of the input data, such as the Fibonacci number to be computed.
4. Implement the algorithm: Implement algorithms like recursive, memoization, bottom-up, matrix exponentiation, binet formula, and fast doubling.
5. Generate test data: Create different input sizes to test algorithm performance.
6. Run the algorithms: Execute each algorithm and record the performance data.
7. Analyze results: Evaluate the data by calculating statistical measures or plotting graphs to show the relationship between problem size and efficiency.

The efficiency measure depends on the goal. If the aim is to study the theoretical performance, counting operations may be useful. To analyze real-world performance, execution time is more appropriate.

Introduction:

The Fibonacci sequence is a series of numbers in which each term is the sum of the two preceding ones. This sequence starts with 0 and 1, and the subsequent numbers are formed by adding these two: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, and so on, with the general recurrence relation $x_n = x_{n-1} + x_{n-2}$.

Although often attributed to Leonardo Fibonacci, an Italian mathematician born around A.D. 1170, the sequence actually appears in earlier mathematical works, notably ancient Sanskrit texts. Fibonacci introduced the sequence to the Western world in his 1202 book, *Liber Abaci*, where he demonstrated the application of the Hindu-Arabic numeral system for commercial calculations.

Traditionally, the Fibonacci sequence was computed by simple iteration, where each term is derived by summing the two preceding numbers. However, as computational techniques advanced, a variety of more efficient methods for calculating Fibonacci numbers emerged. These methods are generally classified into four categories: Recursive Methods, Dynamic Programming Methods, Matrix Power Methods, and Binet Formula Methods. Each category offers varying levels of efficiency, from naive approaches to more optimized solutions that significantly reduce computational complexity.

In this laboratory, we aim to empirically evaluate and compare the performance of five different Fibonacci calculation algorithms. By focusing on their execution times, we will analyze how each method scales with increasing input sizes and investigate the trade-offs between simplicity and efficiency in algorithm design.

Comparison Metric:

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

Input Format:

As input, each algorithm will receive two series of numbers that will contain the order of the Fibonacci terms being looked up. The first series will have a more limited scope, (5, 7, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 37), to accommodate the recursive method, while the second series will have a bigger scope to be able to compare the other algorithms between themselves (501, 631, 794, 1000, 1259, 1585, 1995, 2512, 3162, 3981, 5012, 6310, 7943, 10000, 12589, 15849, 20000, 25000, 30000, 35000, 40000, 45000, 50000, 60000, 70000, 80000, 90000, 100000).

IMPLEMENTATION

All six algorithms - Recursive, Memoization, Bottom-Up Dynamic Programming, Matrix Exponentiation, Binet Formula, and Fast Doubling—will be implemented in Python and analyzed empirically based on their execution time. While the overall trends observed may align with established theoretical expectations, the efficiency of each algorithm may vary depending on the specific input size and the hardware capabilities of the device used during the experiments.

To account for minor variations in execution time due to system performance, an error margin of 2.5 seconds will be considered, based on experimental observations and measurements.

Recursive:

The Recursive method for calculating Fibonacci numbers is the most straightforward approach. It directly follows the mathematical definition of the Fibonacci sequence, where each term is the sum of the two preceding ones. While conceptually simple and easy to implement, the recursive method suffers from inefficiency due to repeated calculations. For larger Fibonacci numbers, the method recalculates the same Fibonacci terms multiple times, leading to an exponential time complexity of $O(2^n)$. This makes the Recursive method unsuitable for large values of n , as the execution time increases rapidly.

Algorithm Description:

The pseudocode for the Recursive method is as follows:

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

Implementation:

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

Figure 1 Fibonacci Recursive Method in Python

Results:

After running the Recursive algorithm for each Fibonacci term in the first input series, the following results were obtained:

Results for Recursive:														
	5	7	10	12	15	17	20	22	25	27	30	32	35	37
Test 0	0.0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.002053	0.008506	0.019117	0.087207	0.219142	0.912786	2.412457
Test 1	0.0	0.0	0.0	0.0	0.0	0.001045	0.000998	0.001001	0.007506	0.020602	0.088492	0.219685	0.903488	2.372126
Test 2	0.0	0.0	0.0	0.0	0.0	0.000000	0.001000	0.002000	0.007559	0.019773	0.083092	0.220903	0.922473	2.491302

Figure 2 Results for Recursive Method

In Figure 2, we observe that the execution time for the Recursive method increases exponentially as n grows. For smaller Fibonacci numbers, the method runs relatively quickly, but as n approaches higher values, the execution time increases dramatically due to the repeated calculations. The graph in Figure 3 illustrates the exponential growth of execution time, confirming the $O(2^n)$ time complexity of the method.

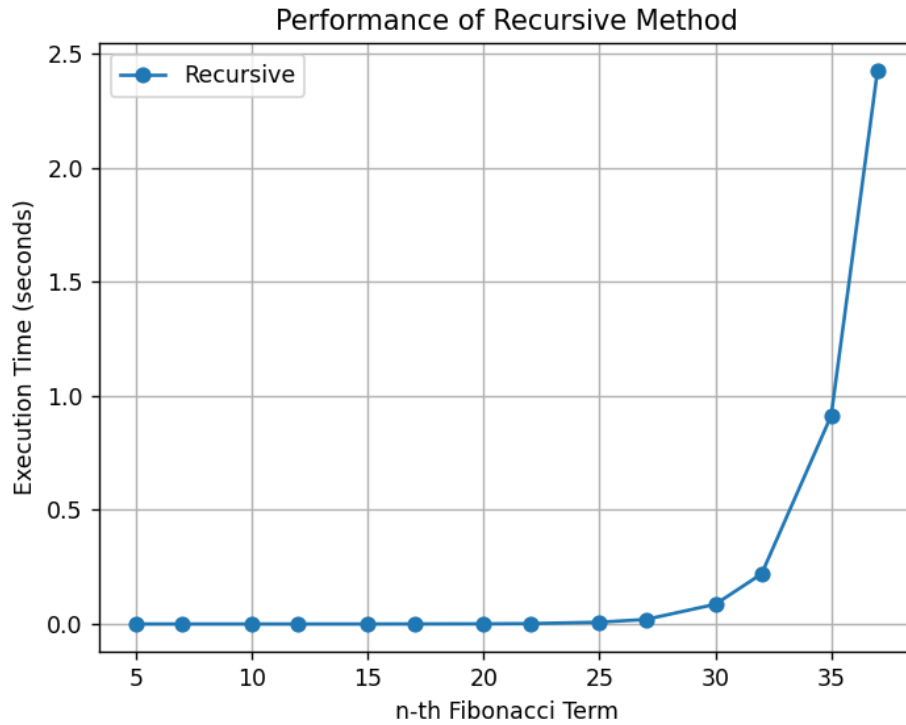


Figure 3 Fibonacci Recursive Method Graph

Memoization:

Memoization is an optimization technique applied to the traditional Recursive method that helps avoid redundant calculations by storing previously computed results in a cache (or table). This method ensures that each Fibonacci number is only computed once, which greatly reduces the number of recursive calls. As a result, the time complexity is reduced from the exponential $O(2^n)$ of the Recursive method to linear $O(n)$, making Memoization much more efficient for larger Fibonacci numbers. The trade-off, however, is the additional space required to store the computed values.

Algorithm Description:

The pseudocode for the Memoization method is as follows:

```
Fibonacci(n, memo):
    if n <= 1: return n
    if memo[n] is not None:
        return memo[n]
    memo[n] = Fibonacci(n-1, memo) + Fibonacci(n-2, memo)
    return memo[n]
```

Implementation:

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

Figure 4: Fibonacci Memoization Method in Python

Results:

After executing the Memoization algorithm for each Fibonacci term in the second input series, the following results were obtained:

Results for Memoization:												
	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310
Test 0	0.0	0.000000	0.0	0.0	0.0	0.001063	0.000	0.0	0.001	0.001001	0.000000	0.000998
Test 1	0.0	0.000000	0.0	0.0	0.0	0.000000	0.000	0.0	0.000	0.000000	0.001314	0.001070
Test 2	0.0	0.000999	0.0	0.0	0.0	0.000000	0.001	0.0	0.000	0.000999	0.000000	0.000931

20000	25000	30000	35000	40000	45000	50000	60000	70000	80000	90000	100000
0.009548	0.012641	0.019741	0.025014	0.031596	0.044744	0.062151	0.074213	0.104107	0.135650	0.169830	0.209465
0.008540	0.014073	0.020077	0.024523	0.036924	0.042036	0.054139	0.077636	0.105795	0.150403	0.169605	0.214622
0.008508	0.012549	0.018019	0.025020	0.035615	0.043081	0.051095	0.086206	0.099048	0.142176	0.171374	0.214556

Figure 5: Results for Memoization

In Figure 5, we observe that Memoization drastically reduces execution time compared to the Recursive method. The time complexity of the Memoization method approaches $O(n)$, making it significantly faster for larger Fibonacci numbers. The graph in Figure 6 clearly demonstrates the improvement, with the execution time growing linearly as n increases, which is characteristic of the $O(n)$ time complexity.

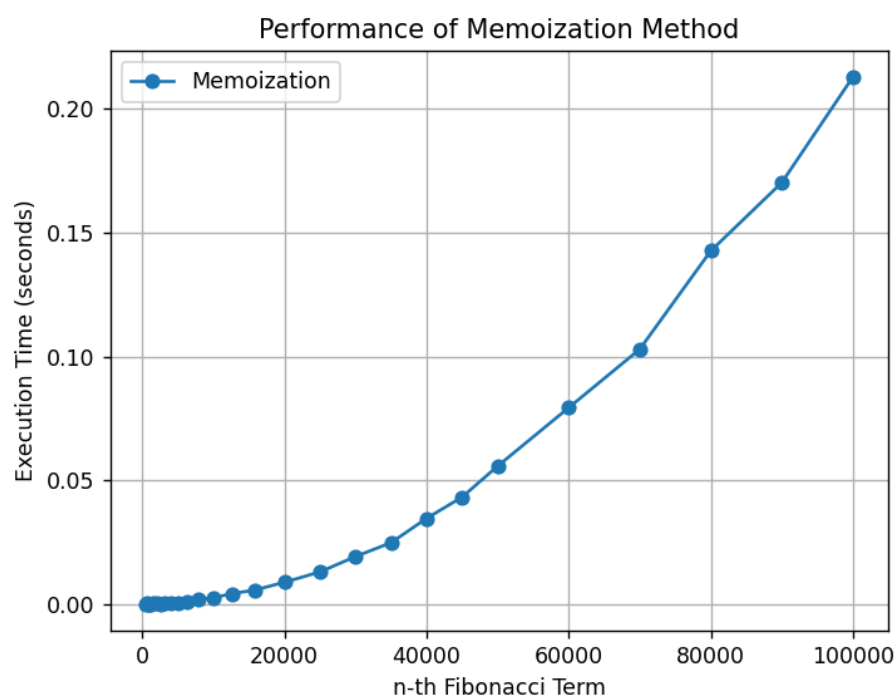


Figure 6 Fibonacci Memoization Method Graph

Bottom-Up Dynamic Programming:

The Bottom-Up Dynamic Programming method is an efficient approach for calculating Fibonacci numbers, eliminating the redundant recalculations inherent in the recursive method. It begins by solving the smallest subproblems and iteratively builds up the solution to the desired Fibonacci term. The method stores the results of previous calculations in a table, ensuring each Fibonacci number is computed only once. As a result, this method operates with a time complexity of $O(n)$, making it much more efficient than the naive recursive approach.

Algorithm Description:

The pseudocode for the Bottom-Up Dynamic Programming method is as follows:

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

Implementation:

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

Figure 7 Fibonacci Bottom-Up DP in Python

Results:

After running the Bottom-Up Dynamic Programming algorithm for each Fibonacci term in the third input series, the following results were recorded:

Results for Bottom-Up:													
	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	
Test 0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000	0.0	0.0	0.000000	0.000000	0.000000	
Test 1	0.0	0.0	0.0	0.0	0.0	0.0	0.000000	0.0	0.0	0.001133	0.000000	0.000998	
Test 2	0.0	0.0	0.0	0.0	0.0	0.0	0.001001	0.0	0.0	0.000000	0.001005	0.000000	

	20000	25000	30000	35000	40000	45000	50000	60000	70000	80000	90000	100000	
0.003517	0.004506	0.006041	0.008047	0.009568	0.012133	0.015277	0.020651	0.028201	0.037206	0.045988	0.061725		
0.003004	0.004001	0.006510	0.007578	0.009562	0.012562	0.015071	0.022135	0.027755	0.035764	0.049059	0.068319		
0.003041	0.004504	0.006542	0.008506	0.010563	0.012090	0.015642	0.021134	0.028269	0.035763	0.053942	0.056840		

Figure 8 Results for Bottom-Up DP

In Figure 8, we observe that the Bottom-Up DP method provides a substantial improvement in execution time compared to the recursive approach. The method's time complexity is linear, $O(n)$,

and scales efficiently with increasing values of n . The graph in Figure 9 demonstrates the consistent and predictable execution time, showing little variation as the input grows.

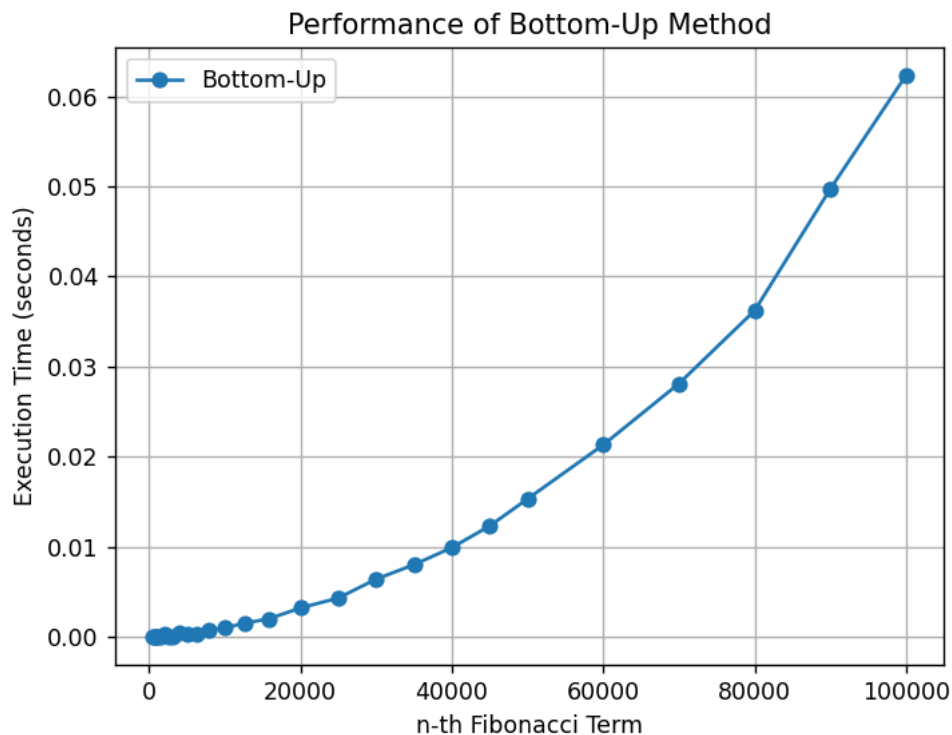


Figure 9 Fibonacci DP Graph

Matrix Exponentiation:

Matrix Exponentiation is an advanced method for calculating Fibonacci numbers efficiently by leveraging the properties of matrices. This method uses the fact that the Fibonacci sequence can be represented in matrix form, allowing for the computation of Fibonacci numbers through matrix multiplication and exponentiation. The time complexity of this method is $O(\log n)$, making it the most efficient for computing large Fibonacci numbers compared to other methods. By exponentiating a matrix to the power of $(n-1)$, it can compute Fibonacci numbers in logarithmic time, offering significant performance improvements for high-order Fibonacci numbers.

Algorithm Description:

The pseudocode for the Matrix Exponentiation method is as follows:

```
MatrixPower(M, p):
    result = [[1, 0], [0, 1]] # Identity matrix
    while p:
        if p % 2 == 1:
            result = MatrixMult(result, M)
        M = MatrixMult(M, M)
        p = p // 2
    return result
```

```

Fibonacci(n):
    if n <= 1:
        return n
    M = [[0, 1], [1, 1]]
    result = MatrixPower(M, n-1)
    return result[1][1]

```

Implementation:

```

def matrix_power(M, p):
    res = [[1, 0], [0, 1]]
    while p:
        if p % 2:
            res = matrix_mult(res, M)
        M = matrix_mult(M, M)
        p //= 2
    return res

def fibonacci_matrix_exponentiation(n):
    if n <= 1:
        return n
    M = [[0, 1], [1, 1]]
    result = matrix_power(M, n - 1)
    return result[1][1]

```

Figure 10 Fibonacci Matrix Exponentiation in Python

Results:

After running the Matrix Exponentiation algorithm for each Fibonacci term in the fourth input series, the following results were obtained:

Results for Matrix Exponentiation:													
	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943
Test 0	0.0	0.0	0.0	0.0	0.000	0.0	0.0	0.0	0.0	0.0	0.0	0.000000	0.0
Test 1	0.0	0.0	0.0	0.0	0.001	0.0	0.0	0.0	0.0	0.0	0.0	0.000000	0.0
Test 2	0.0	0.0	0.0	0.0	0.000	0.0	0.0	0.0	0.0	0.0	0.0	0.001042	0.0

15849	20000	25000	30000	35000	40000	45000	50000	60000	70000	80000	90000	100000
0.000	0.001505	0.001000	0.001047	0.002507	0.002998	0.003039	0.003003	0.004000	0.006069	0.007580	0.008565	0.010082
0.001	0.000000	0.000999	0.002002	0.001999	0.003142	0.002001	0.002999	0.004041	0.007566	0.008565	0.008506	0.008554
0.000	0.001002	0.001000	0.000998	0.002002	0.001999	0.004009	0.002506	0.003504	0.007513	0.008064	0.008571	0.009574

Figure 11 Results for Matrix Exponentiation

In Figure 11, we observe that the Matrix Exponentiation method performs exceptionally well, with its execution time remaining consistently low even for very large Fibonacci terms. The logarithmic time complexity, $O(\log n)$, is evident in the results, as the execution time grows slowly compared to the linear methods. The graph in Figure 12 showcases the efficiency of Matrix Exponentiation, especially when compared to methods with linear or exponential time complexity.

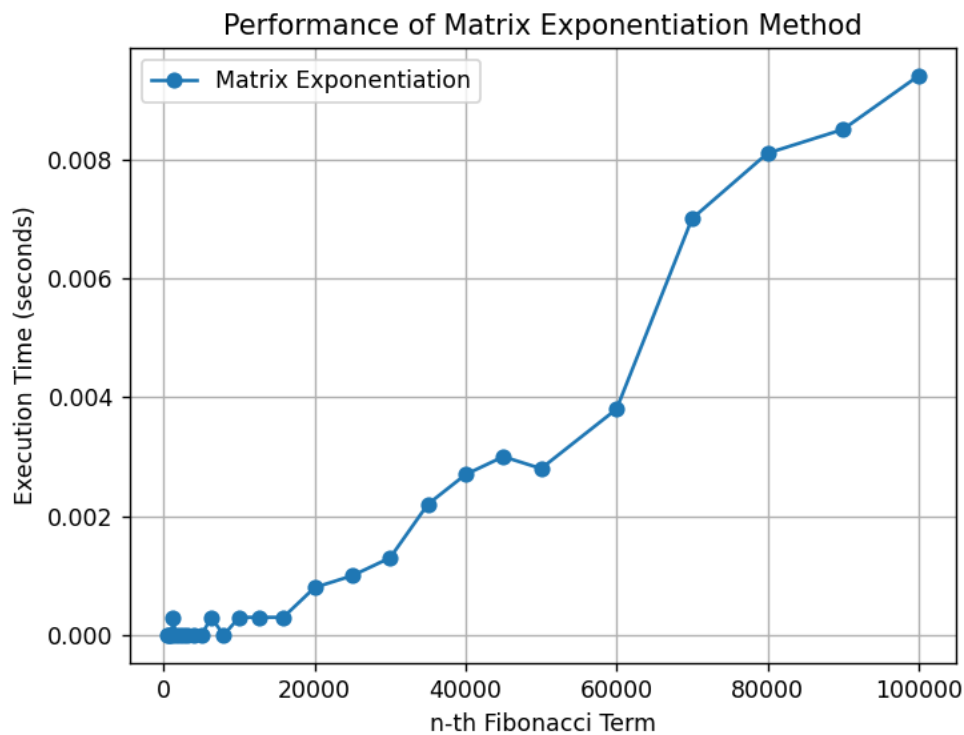


Figure 12 Fibonacci Matrix Exponentiation Graph

Binet Formula:

Binet's Formula provides a direct mathematical expression to calculate the n th Fibonacci number using the Golden Ratio (ϕ). It is derived from the closed-form solution of the Fibonacci recurrence relation. While this method allows for fast computation without iteration or recursion, its main limitation lies in the precision of floating-point arithmetic, particularly for larger Fibonacci numbers, where rounding errors can occur. Despite this, it offers constant time complexity, $O(1)$, and is the fastest method for small to medium Fibonacci numbers, especially when precision is not critical.

Algorithm Description:

The pseudocode for the Binet's Formula method is as follows:

```
Fibonacci(n) :
    sqrt_5 = sqrt(5)
    phi = (1 + sqrt_5) / 2
    return round((phi**n - (-phi)**(-n)) / sqrt_5)
```

Implementation:

```
def fibonacci_binet(n):
    try:
        sqrt_5 = Decimal(5).sqrt()
        phi = (Decimal(1) + sqrt_5) / Decimal(2)
        return int((phi**n / sqrt_5).quantize(Decimal(1)))
    except InvalidOperation:
        return None
```

Figure 13 Fibonacci Binet's Formula in Python

Results:

After running the Binet's Formula algorithm for each Fibonacci term in the fifth input series, the following results were recorded:

Results for Binet Formula:												
	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310
Test 0	0.00106	0.000000	0.000000	0.000	0.001061	0.001001	0.000000	0.001000	0.000999	0.001	0.001000	0.001004
Test 1	0.00100	0.001002	0.001294	0.001	0.000000	0.000000	0.001502	0.001001	0.001001	0.000	0.001001	0.000998
Test 2	0.00100	0.000999	0.000999	0.001	0.001000	0.000999	0.001003	0.000000	0.000999	0.001	0.000504	0.000000

20000	25000	30000	35000	40000	45000	50000	60000	70000	80000	90000	100000
0.001000	0.000999	0.001000	0.001001	0.000999	0.001	0.000000	0.001000	0.000000	0.001	0.001000	0.001
0.001001	0.001000	0.001000	0.001001	0.001000	0.001	0.001503	0.001000	0.000999	0.001	0.000504	0.001
0.000000	0.001000	0.000504	0.000000	0.001060	0.001	0.001004	0.001001	0.001071	0.001	0.001001	0.001

Figure 14 Results for Binet's Formula

In Figure 14, we observe that Binet's Formula provides rapid computation times due to its $O(1)$ time complexity. However, as the Fibonacci number increases, we notice small deviations in the results due to rounding errors in the calculation of the Golden Ratio. These errors become more significant for larger Fibonacci numbers, particularly when dealing with high values of n . The graph in Figure 15 demonstrates the consistent speed of the method, but also highlights potential inaccuracies for larger values.

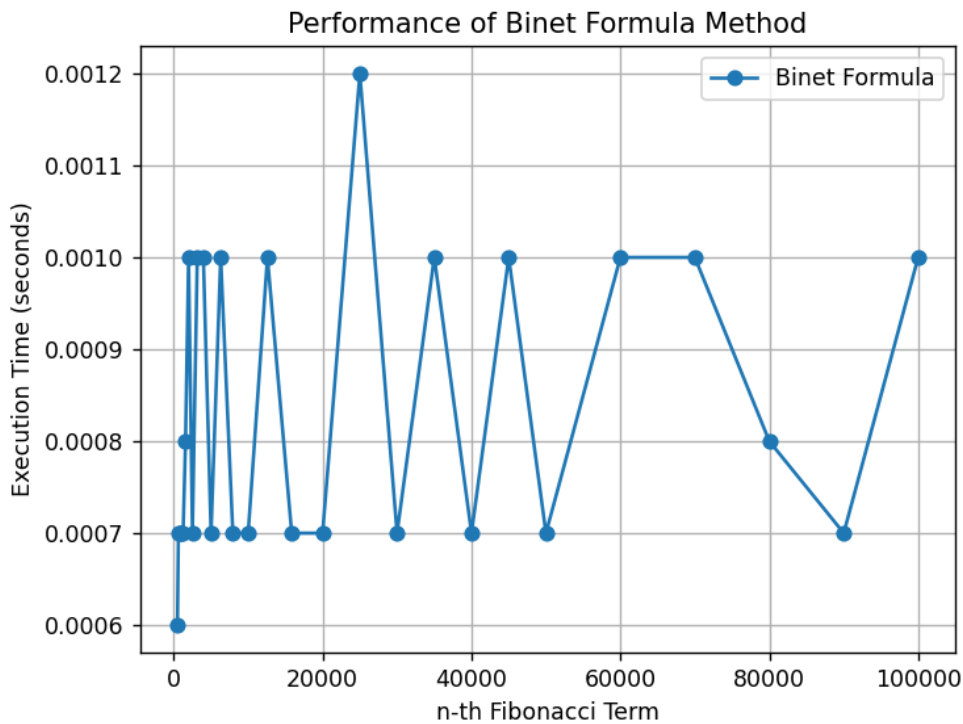


Figure 15 Fibonacci Binet's Formula Graph

Fast Doubling:

The Fast Doubling method is a highly efficient algorithm for computing Fibonacci numbers that leverages the properties of the Fibonacci sequence in a way that allows for faster calculations. By using the following identities:

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

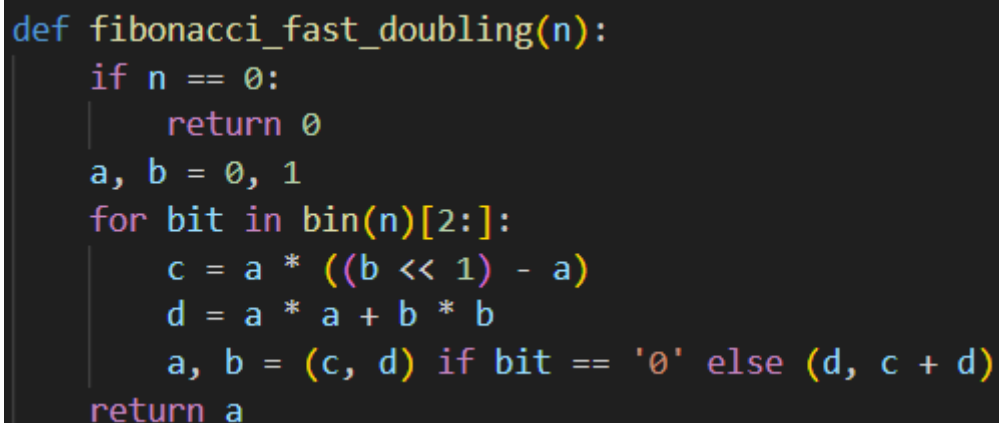
the method computes Fibonacci numbers in a logarithmic time complexity of $O(\log n)$, significantly reducing the number of operations compared to other methods like recursion or dynamic programming. Fast Doubling is especially powerful for very large Fibonacci numbers, where its logarithmic time complexity ensures that the algorithm runs efficiently.

Algorithm Description:

The pseudocode for the Fast Doubling method is as follows:

```
Fibonacci(n):  
    if n == 0:  
        return 0  
    a, b = 0, 1  
    for bit in bin(n)[2:]:  
        c = a * ((b << 1) - a)  
        d = a * a + b * b  
        a, b = (c, d) if bit == '0' else (d, c + d)  
    return a
```

Implementation:

A screenshot of a code editor with a dark background and light-colored text. The code is a Python function named 'fibonacci_fast_doubling(n)'. It starts with a base case 'if n == 0: return 0'. Then it initializes 'a, b = 0, 1'. A loop 'for bit in bin(n)[2:]' iterates over the binary representation of n (excluding the '0b' prefix). Inside the loop, it calculates 'c = a * ((b << 1) - a)' and 'd = a * a + b * b'. Then it updates 'a, b = (c, d) if bit == '0' else (d, c + d)'. Finally, it returns 'a'.

```
def fibonacci_fast_doubling(n):  
    if n == 0:  
        return 0  
    a, b = 0, 1  
    for bit in bin(n)[2:]:  
        c = a * ((b << 1) - a)  
        d = a * a + b * b  
        a, b = (c, d) if bit == '0' else (d, c + d)  
    return a
```

Figure 16 Fibonacci Fast Doubling in Python

Results:

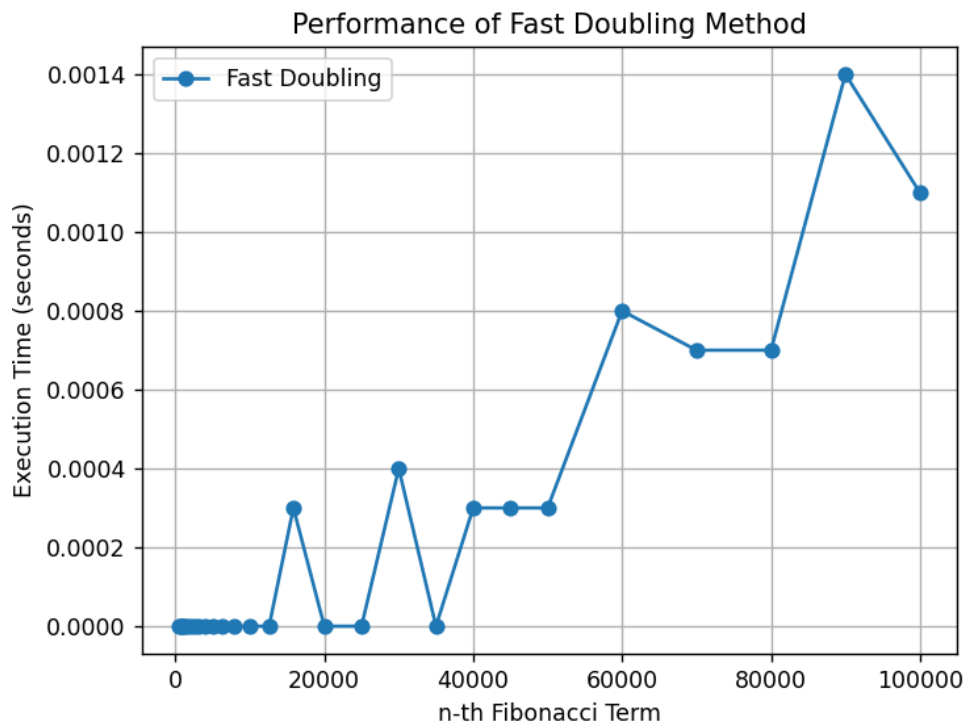
After executing the Fast Doubling algorithm for each Fibonacci term in the sixth input series, the following results were obtained:

Results for Fast Doubling:													
	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943
Test 0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Test 1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Test 2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

15849	20000	25000	30000	35000	40000	45000	50000	60000	70000	80000	90000	100000
0.001002	0.0	0.0	0.000000	0.0	0.001004	0.000996	0.000998	0.001503	0.000000	0.001002	0.001057	0.001000
0.000000	0.0	0.0	0.001066	0.0	0.000000	0.000000	0.000000	0.000000	0.000999	0.001001	0.002000	0.001270
0.000000	0.0	0.0	0.000000	0.0	0.000000	0.000000	0.000000	0.001004	0.000999	0.000000	0.001000	0.001001

Figure 17 Results for Fast Doubling

In Figure 17, the Fast Doubling method exhibits an excellent performance with significantly reduced computation times compared to the recursive and memoization methods. The execution time remains very low even for large Fibonacci terms, as the method's logarithmic time complexity ensures rapid computation. The graph in Figure 18 demonstrates how efficiently the method handles increasing values of n , showing minimal growth in execution time despite the large Fibonacci numbers being calculated.



CONCLUSION

Through empirical analysis, this paper evaluates the performance and efficiency of six different methods for calculating Fibonacci numbers. These methods range in both complexity and computational performance, providing insights into their potential applications depending on the size of the Fibonacci number and the resources available.

The Recursive method, although conceptually simple, is the least efficient due to its exponential time complexity of $O(2^n)$. It is best suited for calculating smaller Fibonacci numbers (up to around $n=30$) where computation times are manageable. However, its inefficiency makes it impractical for larger values of n , as the computation time increases rapidly with larger inputs.

The Binet Formula provides a quick and easy way to compute Fibonacci numbers using the Golden Ratio, with almost constant time complexity. It is an efficient choice for calculating Fibonacci numbers up to approximately $n=80$. However, due to the potential for rounding errors in floating-point arithmetic, the results may not be entirely accurate, particularly when working with very large numbers.

The Memoization method optimizes the naive Recursive method by storing previously calculated values in a cache, reducing the time complexity to $O(n)$. This results in significantly faster execution times, especially for larger Fibonacci numbers. Memoization is a highly effective approach for mid-range Fibonacci numbers and strikes a good balance between efficiency and simplicity. However, it does require additional space to store the intermediate results.

The Matrix Exponentiation method utilizes matrix properties to compute Fibonacci numbers in logarithmic time, $O(\log n)$. This makes it highly efficient for calculating large Fibonacci numbers, significantly outperforming both the Recursive and Memoization methods in terms of execution time for large inputs. Matrix Exponentiation is an excellent choice for very large Fibonacci numbers, but it is more complex to implement.

The Fast Doubling method, with its $O(\log n)$ time complexity, performs similarly to Matrix Exponentiation in terms of efficiency. This method is especially powerful when dealing with very large Fibonacci numbers, offering both speed and accuracy. Fast Doubling is a highly efficient choice and is often preferred for applications requiring rapid Fibonacci number generation.

In conclusion, each method has its own strengths and limitations. For small to medium Fibonacci numbers, the Recursive and Memoization methods may suffice, with Memoization offering better performance for larger n . For large Fibonacci numbers, both Matrix Exponentiation and Fast Doubling are excellent choices, with the latter being simpler to implement and providing faster results. Depending on the size of the Fibonacci number and the need for computational efficiency, these methods can be chosen to optimize both performance and accuracy.