



**MINISTRY OF EDUCATION AND RESEARCH
OF THE REPUBLIC OF MOLDOVA**

Technical University of Moldova

Faculty of Computers, Informatics, and Microelectronics

Department of Software and Automation Engineering

Report

Laboratory Work No.1

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

Checked by:

Fiștic Cristofor, university assistant

Made by:

Alexandra Mihalevschi, FAF-232

Chișinău – 2024

Table of contents

INTRODUCTION	3
The purpose of the laboratory work	3
The condition of the tasks.....	3
Theoretical notes.....	3
Fibonacci sequence: Theoretical explanation.....	5
Definition of the Fibonacci sequence	5
Mathematical properties of Fibonacci numbers	6
ALGORITHM ANALYSIS	7
Comparison Metrics	7
Approach	7
Input format	7
Key observations	7
IMPLEMENTATION OF ALGORITHMS IN CODE.....	9
Naive recursive	9
Memoized recursive approach (top-down dynamic programming)	10
Bottom-up dynamic programming (tabulation).....	12
Matrix exponentiation/matrix power method	15
Iterative approach (Loop based)	18
Binet formula.....	20
PARALEL COMPARISION OF THE ALGORITHMS	22
CONCLUSION	24

INTRODUCTION

The purpose of the laboratory work

The purpose of this laboratory work is to study and analyze different algorithms for determining Fibonacci n-th term.

The condition of the tasks

To do:

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 is a method of investigation or research that relies on observed and measured data rather than theory or pure logic. The aim is to gather real-world evidence to form conclusions or test hypotheses. This approach is used in a variety of fields, including science, economics, social sciences, and engineering, to understand phenomena based on actual data or experiences rather than on theoretical models or assumptions.

Key features of empirical analysis:

1. Data collection: Empirical analysis is rooted in data. This data could come from:
 - Experiments: Controlled studies where variables are manipulated.
 - Observations: Passive collection of data from real-world settings without interference.
 - Existing Datasets: Using previously collected data from secondary sources.
2. Empirical evidence: The data collected serves as empirical evidence, which means it is based on actual observations or experiments. For instance, in a scientific study, empirical evidence would be the direct measurements or observations made during an experiment, such as temperature readings, survey responses, or sensor outputs.

3. Testing hypotheses: Empirical analysis is often used to test hypotheses or theories. A hypothesis is a proposed explanation for a phenomenon, and empirical analysis provides the real-world data to either support or reject this hypothesis.
 - Support: If the collected data matches the predicted outcomes, the hypothesis is supported.
 - Rejection: If the data contradicts the hypothesis, it may be rejected or revised.
4. Quantitative vs. qualitative:
 - Quantitative empirical analysis: This deals with numerical data and uses statistical methods to analyze the relationships, trends, and patterns. For example, regression analysis, hypothesis testing, or machine learning algorithms.
 - Qualitative empirical analysis: This focuses on non-numerical data like interviews, observations, and case studies. It involves analyzing patterns, themes, or narratives in the data. Methods like content analysis or grounded theory are often employed in qualitative research.
5. Real-world context: One of the most significant aspects of empirical analysis is its focus on real-world context. The conclusions drawn from empirical analysis are not theoretical but based on what is actually observed in practice. This means the findings are often highly applicable in real-world settings.
6. Objectivity and replicability: A strong empirical analysis emphasizes objectivity—ensuring the results are not biased by personal opinions or external factors. Moreover, a good empirical study should be replicable, meaning that others can replicate the experiment or observation and obtain similar results.

Steps in empirical analysis:

1. Define the problem:
 - Empirical analysis often begins with a well-defined research question or hypothesis. This could involve identifying a gap in existing knowledge or understanding a real-world problem.
2. Data collection:
 - Decide how the data will be gathered, whether through experiments, surveys, observations, or the use of pre-existing data.

3. Data analysis:

- The data is analyzed to test hypotheses or to look for patterns, trends, or correlations. Statistical techniques like regression analysis, t-tests, or ANOVA may be used for quantitative data.

4. Interpretation of results:

- Based on the analysis, researchers interpret the results, determining whether the data supports or contradicts the hypothesis. This interpretation may also involve making recommendations or suggesting further research.

5. Conclusion and reporting:

- The findings are summarized and presented in a way that allows others to evaluate the methods and results. In academic settings, this is typically presented as a research paper, thesis, or report.

Fibonacci sequence: Theoretical explanation

Fibonacci sequence, the sequence of numbers 1, 1, 2, 3, 5, 8, 13, 21, ..., each of which, after the second, is the sum of the two previous numbers; that is, the n th Fibonacci number

$$F_n = F_{n-1} + F_{n-2}$$

The sequence was noted by the medieval Italian mathematician Fibonacci (Leonardo Pisano) in his *Liber abaci* (1202; “Book of the Abacus”), which also popularized Hindu-Arabic numerals and the decimal number system in Europe. Fibonacci introduced the sequence in the context of the problem of how many pairs of rabbits there would be in an enclosed area if every month a pair produced a new pair and rabbit pairs could produce another pair beginning in their second month. The numbers of the sequence occur throughout nature, such as in the spirals of sunflower heads and snail shells. The ratios between successive terms of the sequence tend to the golden ratio $\varphi = (1 + \sqrt{5})/2$ or 1.6180....

Definition of the Fibonacci sequence

The Fibonacci series is a sequence of numbers where each number is the sum of the two preceding ones, starting from 0 and 1.

The series starts as follows:

$$F(0)=0, \quad F(1)=1, \quad F(2)=1,$$

$$\begin{array}{lll} F(3)=2, & F(4)=3, & F(5)=5, \\ F(6)=8, & F(7)=13, & \dots \end{array}$$

Mathematical properties of Fibonacci numbers

1. Golden ratio:

- The Fibonacci sequence is closely related to the golden ratio, often denoted by ϕ . The golden ratio is an irrational number approximately equal to 1.6180339887.
- As you go further into the Fibonacci sequence, the ratio of consecutive Fibonacci numbers approaches the golden ratio:

$$\frac{F_{n+1}}{F_n} \approx \phi$$

This relationship between Fibonacci numbers and the golden ratio is fundamental in various areas such as geometry, art, and architecture. The golden ratio is said to have aesthetically pleasing properties and has been used in the design of historical buildings and works of art.

2. Binet's formula:

- The Fibonacci numbers can be calculated explicitly using a formula known as Binet's formula. It involves the golden ratio and its conjugate. The formula for the n -th Fibonacci number is:

$$F_n = \frac{\phi^n - (1 - \phi)^n}{\sqrt{5}}$$

Binet's formula provides a direct, closed-form expression to calculate any Fibonacci number without relying on recursion or iteration.

ALGORITHM ANALYSIS

In this laboratory report, I analyzed and compared the performance of different Fibonacci algorithms using **empirical analysis**. The goal was to understand how the execution time of each algorithm changes as the input size, **n**, increases. I tested 6 different algorithms to compute Fibonacci numbers and measure the time they take to calculate the Fibonacci number for various values of **n**. The algorithms include:

1. **Naive recursive** (Exponential Complexity)
2. **Memoized recursive** (Top-down dynamic programming)
3. **Bottom-up dynamic programming** (Tabulation)
4. **Matrix exponentiation**
5. **Binet's Formula** (Approximation)
6. **Iterative approach**

Comparison Metrics

For this analysis, I focused on a single metric: **execution time, $T(n)$** . The execution time represents how long each algorithm takes to compute the Fibonacci number for a given input size. I recorded the time taken for each algorithm at different values of **n**, from small values like 5 up to larger values (e.g., 1000). The goal was to compare how quickly each algorithm runs as **n** increases.

Approach

I implemented each algorithm in Python and then measured the time it took to compute the Fibonacci number using the `time.time()` function. For each algorithm, I ran the computation for several values of **n** and recorded the time taken. The measured times were then plotted to visualize how each algorithm behaves as **n** increases.

GitHub link: https://github.com/AlexandraMihalevschi/AA_Lab1

Input format

In this laboratory work, the input format consists of a single integer **n**, representing the position in the Fibonacci sequence for which the corresponding Fibonacci number needs to be computed. The input values are predefined as lists of integers, with smaller values used for recursive methods and larger values for more efficient approaches.

Key observations

1. **Naive recursive**: This method has an exponential time complexity, which means its execution time increases very quickly as **n** gets larger. As expected, this algorithm performs poorly for larger values of **n**, taking a significant amount of time.

2. **Memoized recursive:** By caching previous results, this method reduces the number of redundant calculations, bringing the time complexity down to linear. It performs much better than the naive approach, especially for larger **n**.
3. **Bottom-up dynamic programming:** This algorithm also has a linear time complexity and is very efficient. It uses a table to store intermediate results and calculates the Fibonacci
4. number in a bottom-up manner. It performs similarly to the memoized recursive approach but is more efficient in terms of space and time.
5. **Matrix exponentiation:** This approach uses matrix multiplication to compute Fibonacci numbers and has a time complexity of $O(\log n)$. It performs extremely well, even for large values of **n**, and provides the fastest results overall.
6. **Binet's Formula:** This is an approximation formula and computes the Fibonacci number using a closed-form expression. It has constant time complexity and performs very well on smaller values of **n**. However, its precision can decrease for larger values of **n** due to the limitations of floating-point calculations.
7. **Iterative approach:** This method uses a simple loop to calculate the Fibonacci number. It has linear time complexity and is very straightforward. It performs similarly to the bottom-up dynamic programming approach and is easy to implement.

IMPLEMENTATION OF ALGORITHMS IN CODE

Naïve recursive

The naïve recursive approach follows the direct mathematical definition of the Fibonacci sequence. It calculates each Fibonacci number by recursively calling the function for the two preceding numbers, leading to a large number of redundant calculations. This method is often used for educational purposes to illustrate the concept of recursion, but it is highly inefficient for large inputs due to the excessive number of recursive calls. Here is the formula used by the algorithm:

$$F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 1$$

Where $F(0)=0$ and $F(1)=1$

The function calls itself twice for each value of **n**, leading to an exponential number of recursive calls.

Time complexity: $O(2^n)$

Pseudocode:

```
function Fibonacci(n):  
    if n == 0:  
        return 0  
  
    if n == 1:  
        return 1  
  
    return Fibonacci(n-1) + Fibonacci(n-2)
```

Code implementation:

```
def fib_recursive(n): 4 usages new *  
    if n <= 1:  
        return n  
    return fib_recursive(n-1) + fib_recursive(n-2)
```

Figure 1. Code implementation for the recursive function

Results:

```
Testing Recursive method:  
Time for fib_recursive(5): 0.000000 seconds  
Time for fib_recursive(8): 0.000000 seconds  
Time for fib_recursive(13): 0.000000 seconds  
Time for fib_recursive(21): 0.002000 seconds  
Time for fib_recursive(34): 1.012178 seconds
```

Figure 2. Time required for computing

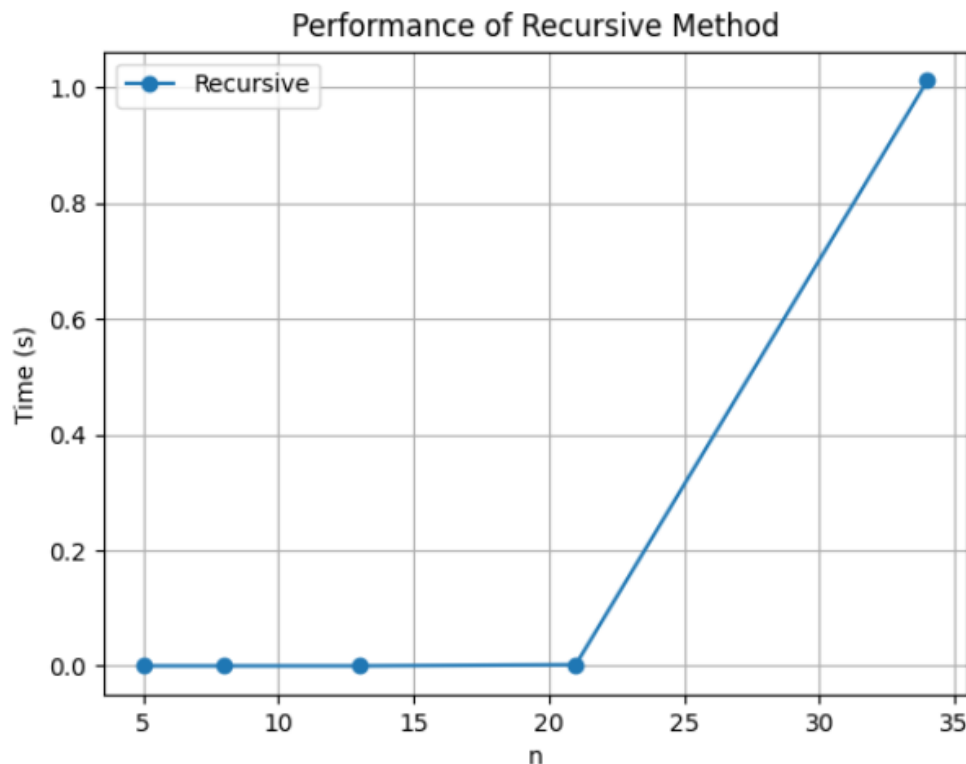


Figure 3. The performance of recursive method is slowing down right after the 21 value and it's getting worse

Conclusions:

Many Fibonacci numbers are recalculated multiple. This unnecessary recomputation causes an exponential time complexity, making it **impractical** for large values of n .

Memoized recursive approach (top-down dynamic programming)

Memoization is an optimization technique that stores the results of expensive function calls and returns the cached result when the same inputs occur again. By applying memoization to the

recursive Fibonacci function, we prevent redundant computations by storing previously computed values in a lookup table or cache.

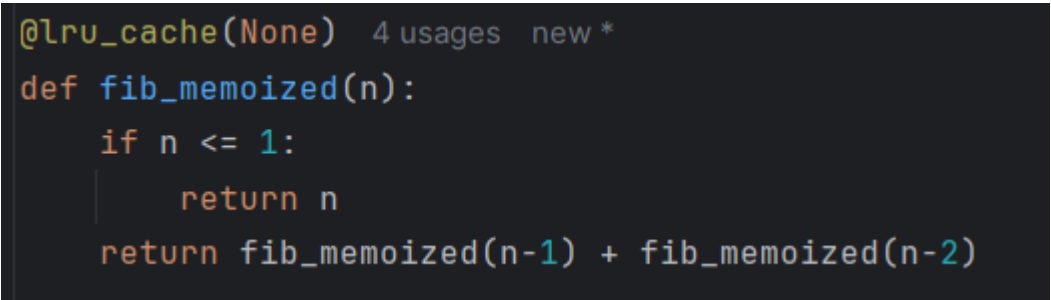
Using Python's `@lru_cache(None)`, each Fibonacci number is computed only once and then stored, significantly reducing the number of recursive calls. Unlike the naïve recursive approach, which recalculates the same Fibonacci numbers multiple times, memoization ensures that each Fibonacci number is computed only once and retrieved in constant time $O(1)$ when needed.

Time complexity: $O(n)$

Pseudocode:

```
function Fibonacci(n, memo={}):  
    if n in memo:  
        return memo[n]  
  
    if n == 0:  
        return 0  
  
    if n == 1:  
        return 1  
  
    memo[n] = Fibonacci(n-1, memo) + Fibonacci(n-2, memo)  
    return memo[n]
```

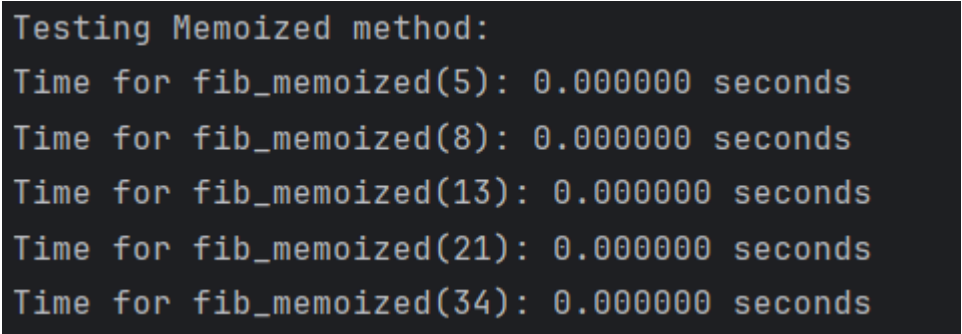
Code implementation:



```
@lru_cache(None) 4 usages new *  
def fib_memoized(n):  
    if n <= 1:  
        return n  
    return fib_memoized(n-1) + fib_memoized(n-2)
```

Figure 4. Code implementation for the recursive function

Results:



```
Testing Memoized method:  
Time for fib_memoized(5): 0.000000 seconds  
Time for fib_memoized(8): 0.000000 seconds  
Time for fib_memoized(13): 0.000000 seconds  
Time for fib_memoized(21): 0.000000 seconds  
Time for fib_memoized(34): 0.000000 seconds
```

Figure 5. Time required for computing

Here is the performance graph:

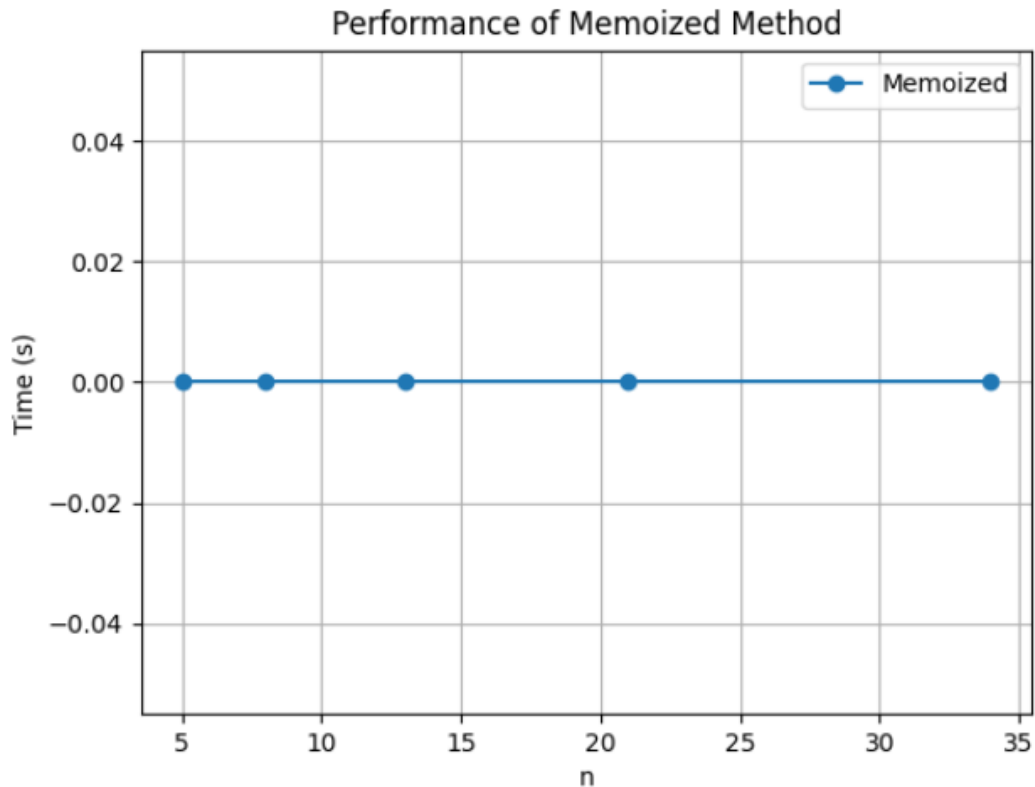


Figure 6. The performance of this method is better than recursive

Conclusions:

This method is much better, it reduces the time complexity to $O(n)$ since each number is computed once. However, it still requires $O(n)$ space to store the computed values in memory. Unfortunately, for my machine I had to use a smaller set of numbers for this method, otherwise it crashed. But it is very good in time of speed, even compared on the same set with previous method, which makes it a whole lot better choice for computing n -th term of the Fibonacci sequence.

Bottom-up dynamic programming (tabulation)

Dynamic programming is a technique that solves problems by breaking them down into smaller subproblems and solving them iteratively. The bottom-up approach, also known as **tabulation**, starts with base cases and iteratively builds up solutions for larger values of n using a table (array) to store previously computed Fibonacci numbers.

Instead of recursion, a simple loop is used to compute $F(n)$. The formula remains the same but instead of making recursive calls, the function iterates from 2 to n , updating an array (dp)

where each value is stored and retrieved when needed. This avoids the overhead of recursive function calls and reduces the memory footprint.

Time complexity: $O(n)$

Pseudocode:

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

Code implementation:

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

Figure 7. Code implementation for the recursive function

Results:

```
Testing Bottom-Up method:  
Time for fib_bottom_up(5): 0.000000 seconds  
Time for fib_bottom_up(21): 0.000000 seconds  
Time for fib_bottom_up(89): 0.001029 seconds  
Time for fib_bottom_up(233): 0.000000 seconds  
Time for fib_bottom_up(987): 0.000000 seconds  
Time for fib_bottom_up(1597): 0.000000 seconds  
Time for fib_bottom_up(2584): 0.000508 seconds  
Time for fib_bottom_up(4181): 0.000000 seconds  
Time for fib_bottom_up(10946): 0.007183 seconds  
Time for fib_bottom_up(17711): 0.014535 seconds  
Time for fib_bottom_up(28657): 0.042898 seconds
```

Figure 8. Time required for computing

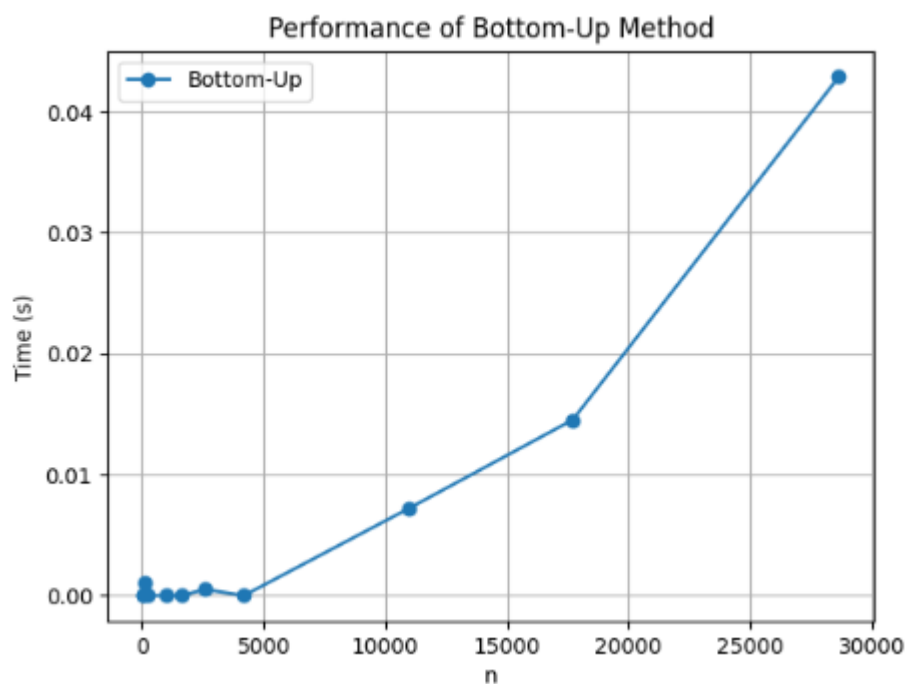


Figure 9. The performance of this method is better than recursive

Conclusions:

This algorithm is also good, since it has a linear time complexity, $O(n)$, because we compute each Fibonacci number once, and the space complexity is also $O(n)$ due to the array used

for storing values. Since the used list of values this time was the bigger one, we can see the tendency of the linear complexity, as is expected by this algorithm.

Matrix exponentiation/matrix power method

A more mathematical approach to computing Fibonacci numbers efficiently is using **matrix exponentiation**. The Fibonacci sequence can be represented using matrix multiplication:

$$\begin{bmatrix} f_n \\ f_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f_{n-1} \\ f_{n-2} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{bmatrix}$$

By raising this matrix to the power of (n-1), the Fibonacci number F(n) can be obtained directly from the resulting matrix.

This method leverages divide-and-conquer exponentiation (also known as fast exponentiation or binary exponentiation) to compute large powers efficiently. Instead of multiplying the matrix nnn times, we reduce the number of multiplications using the property:

$$A^{2k} = (A^k) \times (A^k), \quad A^{2k+1} = A \times (A^{2k})$$

This allows us to compute Fibonacci numbers in $O(\log n)$ time, making it significantly faster than any of the previous methods for large values of nnn. The space complexity is $O(1)$ if implemented iteratively $O(\log n)$ if implemented recursively.

Time complexity: $O(\log n)$

Pseudocode:

```
function MatrixMultiply(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]]]

function MatrixPower(M, n):
    if n == 1:
        return M
    if n % 2 == 0:
        half = MatrixPower(M, n/2)
        return MatrixMultiply(half, half)
    else:
        return MatrixMultiply(M, MatrixPower(M, n-1))
```

```

function Fibonacci(n):
    if n == 0:
        return 0

    M = [[1, 1], [1, 0]]
    result = MatrixPower(M, n-1)
    return result[0][0]

```

Code implementation:

```

def matrix_mult(A, B): 2 usages new *
    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 matrix_power(F, n): 3 usages new *
    if n == 1:
        return F
    if n % 2 == 0:
        half = matrix_power(F, n // 2)
        return matrix_mult(half, half)
    else:
        return matrix_mult(F, matrix_power(F, n - 1))

def fib_matrix(n): 2 usages new *
    if n == 0:
        return 0
    F = [[1, 1], [1, 0]]
    result = matrix_power(F, n - 1)
    return result[0][0]

```

Figure 10. Code implementation for the recursive function

Results:

```
Testing Matrix Power method:  
Time for fib_matrix(5): 0.000000 seconds  
Time for fib_matrix(21): 0.000000 seconds  
Time for fib_matrix(89): 0.000000 seconds  
Time for fib_matrix(233): 0.000000 seconds  
Time for fib_matrix(987): 0.000000 seconds  
Time for fib_matrix(1597): 0.000000 seconds  
Time for fib_matrix(2584): 0.000000 seconds  
Time for fib_matrix(4181): 0.000000 seconds  
Time for fib_matrix(10946): 0.000000 seconds  
Time for fib_matrix(17711): 0.001008 seconds  
Time for fib_matrix(28657): 0.000000 seconds
```

Figure 11. Time required for computing

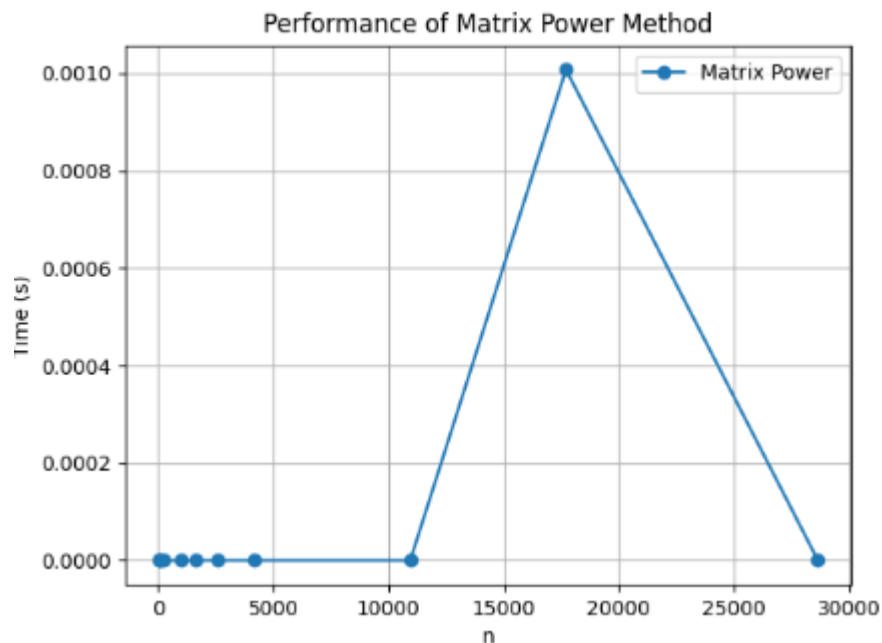


Figure 12. The performance of this method is better than recursive

Conclusions:

This algorithm is also good, since it has a logarithmic time complexity, $O(\log n)$, because it uses the divide and conquer approach. However, during multiple simulations, there was often a spike of required time to compute a value. This might be caused by the error of time computing,

as well by the nature of this method, which involves repeated function call and creation of intermediate matrices, whichh could lead to some overflows or precision issues.

Iterative approach (Loop based)

The iterative approach is one of the simplest and most efficient ways to compute Fibonacci numbers. Instead of using recursion or storing a full table of values, it maintains only the last two Fibonacci numbers and updates them iteratively.

The logic is straightforward: start with $F(0)=0$ and $F(1)=1$, then loop from 2 to **n**, computing each Fibonacci number by summing the previous two values. This method avoids function call overhead and unnecessary memory usage, making it an excellent choice for practical applications.

Since we only store two values at a time, the space complexity is $O(1)$, and the time complexity is $O(n)$, as we iterate through all numbers up to **n**. This makes it both time-efficient and space-efficient.

Time complexity: $O(n)$

Pseudocode:

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

Code implementation:

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

Figure 13. Code implementation for the recursive function

Results:

```
Testing Iterative method:  
Time for fib_iterative(5): 0.000000 seconds  
Time for fib_iterative(21): 0.000000 seconds  
Time for fib_iterative(89): 0.000000 seconds  
Time for fib_iterative(233): 0.000000 seconds  
Time for fib_iterative(987): 0.000000 seconds  
Time for fib_iterative(1597): 0.001000 seconds  
Time for fib_iterative(2584): 0.000000 seconds  
Time for fib_iterative(4181): 0.000000 seconds  
Time for fib_iterative(10946): 0.002002 seconds  
Time for fib_iterative(17711): 0.004283 seconds  
Time for fib_iterative(28657): 0.010627 seconds
```

Figure 14. Time required for computing

The performance graph looks like this:

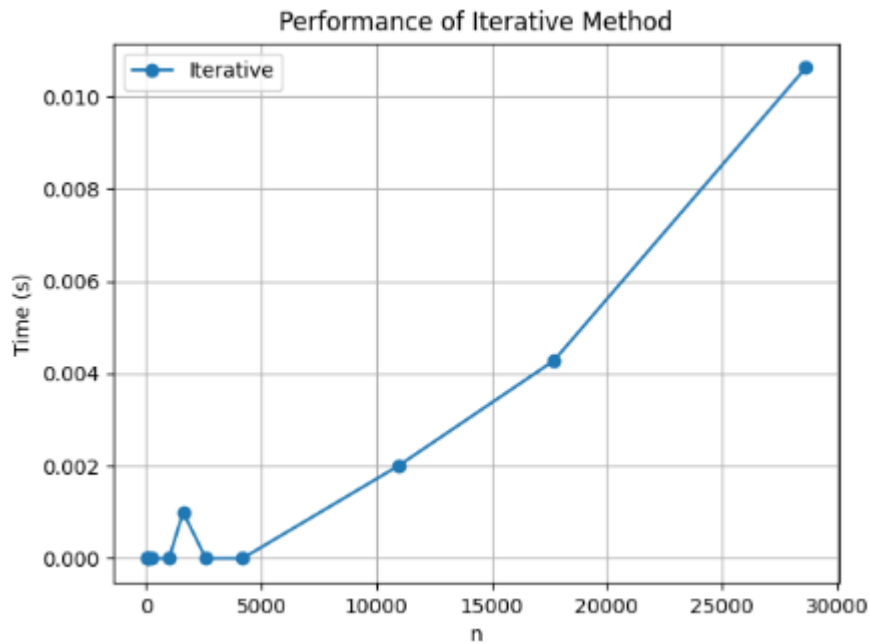


Figure 15. The performance of this method is better than recursive

Conclusions:

This algorithm is maybe not the best from all the algorithms from here, but it is still good enough, considering its performance. From the graph we observe that it is stable and behaves linearly. It's not the best from the time perspective, because with increasing n it will increase in needed time. However, a good pro is its simplicity and space complexity, does not require a lot of functions or extreme computations.

Binet formula

Binet's formula expresses the Fibonacci sequence in terms of the **golden ratio** (ϕ) as follows:

$$F_n = \frac{\phi^n + \phi^n}{\sqrt{5}}$$

$$F_n = \frac{\phi^n - (1 - \phi)^n}{\sqrt{5}}$$

where ϕ is the **golden ratio**. This formula theoretically allows us to compute Fibonacci numbers in constant time $O(1)$ by directly substituting n into the equation.

However, due to the use of floating-point arithmetic, it suffers from **precision errors** when n becomes large. The term $\phi^n + \phi^n$ approaches zero for large n , but the floating-point precision

of the exponentiation and division operations can lead to rounding errors. Additionally, raising a floating-point number to a high power can cause overflow issues.

Time complexity: $O(1)$

Pseudocode:

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

Code implementation:

```
def fib_binet(n):  
    if n > 70: # Limit n to avoid overflow  
        raise ValueError("Binet formula is not accurate for n > 70")  
    sqrt5 = np.sqrt(5)  
    phi = (1 + sqrt5) / 2  
    return round((phi**n - (-1/phi)**n) / sqrt5)
```

Figure 16. Code implementation for the recursive function

Results:

```
Time for fib_binet(5): 0.001045 seconds  
Time for fib_binet(8): 0.000000 seconds  
Time for fib_binet(13): 0.000000 seconds  
Time for fib_binet(21): 0.000000 seconds  
Time for fib_binet(34): 0.000000 seconds
```

Figure 17. Time required for computing

Here is the performance graph:

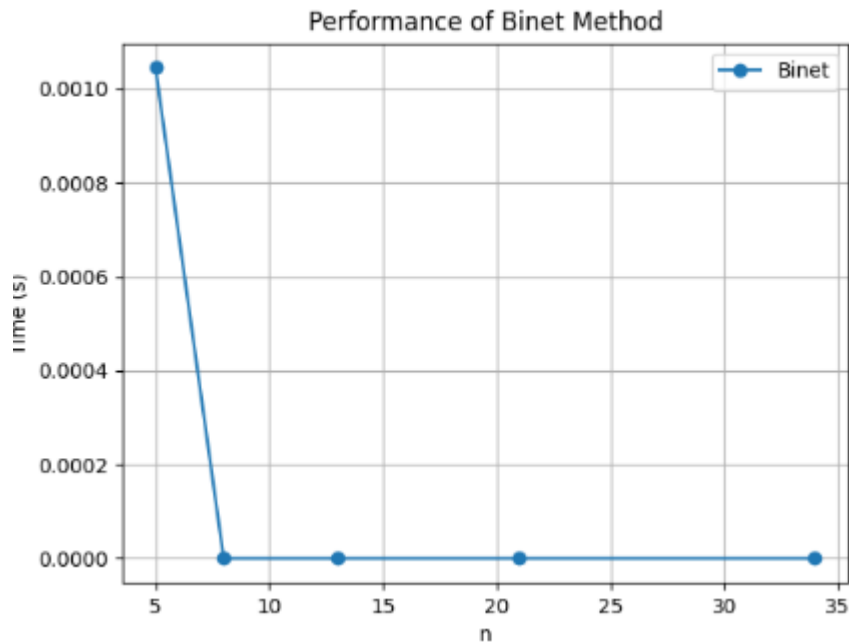


Figure 18. The performance of this method is better than recursive

Conclusions:

Though mathematically interesting, the method is not the best from all those analysed. It might be impractical for large n 's and it makes this method a bit unreliable. However, for small n and for the fastness, the method has potential to be used, in dependence of the situation. However, it's important to understand, that the error's are quite possible to appear, as we see in the graph (the spike at the very first value).

PARALEL COMPARISION OF THE ALGORITHMS

In this chapter, I will compare different ways to calculate Fibonacci numbers and see how fast they work. I tested previously the methods and by measuring the time each method takes, I can find out now which one is the fastest and most efficient.

Graph representation:

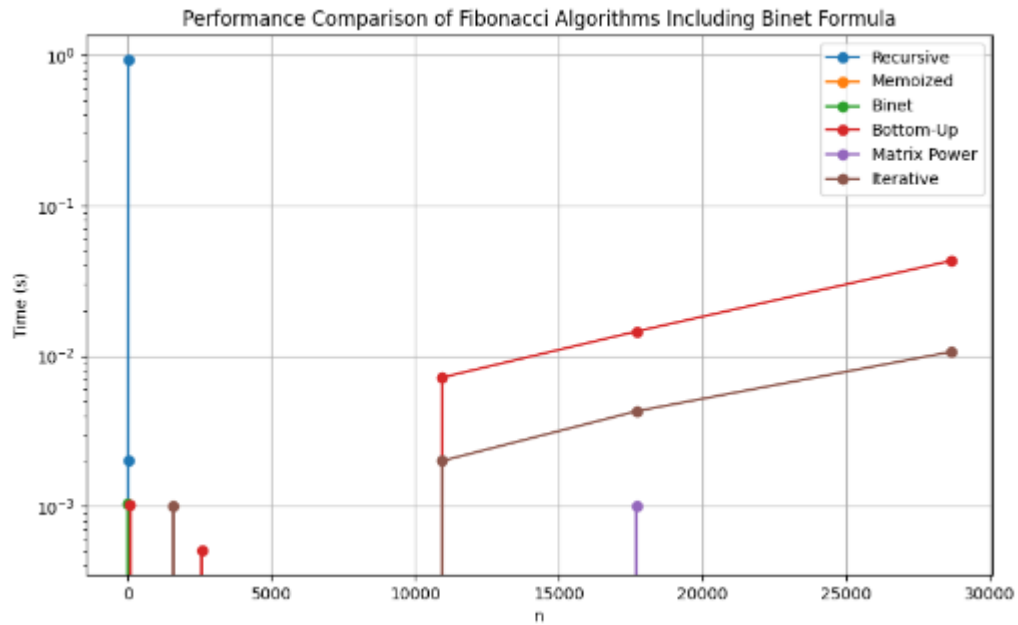


Figure 19. The overall performance of all methods

From this graph representation, it's clear that recursive method is the worst of them all. Binet and Memoized are pretty similar in performance, as are the Bottom-up, and Iterative. One of the most efficient from the graph is the metrix power, since at the great values of n the time values are quite small, which proves it's effectiveness and reliability at large values of n.

Method	Time complexity	Notes
Naïve Recursion	$O(2^n)$	Exponential and inefficient
Memoization (Top-Down DP)	$O(n)$	Stores results to avoid recomputation
Bottom-Up DP	$O(n)$	Iterative, efficient
Matrix Exponentiation	$O(\log n)$	Fastest method for large n
Iterative	$O(n)$	Simple and space-efficient
Binet's Formula	$O(1)$	Not reliable for large n

Table 1. The performance of all methods

CONCLUSION

Through empirical analysis, this paper has evaluated five different methods for computing Fibonacci numbers, analyzing their accuracy and efficiency in terms of execution time complexity. The purpose of this analysis was to determine the practical applicability of each approach, as well as to identify potential optimizations for improving their feasibility.

The **Naive Recursive** method, while conceptually simple and easy to implement, exhibits exponential time complexity $O(2^n)$, making it impractical for larger Fibonacci numbers beyond $n=30$ due to excessive computational overhead. The **Memoized Recursive** approach significantly improves efficiency by storing intermediate results, reducing the complexity to $O(n)$, making it feasible for moderately large values but still constrained by recursion overhead.

The **Bottom-Up Dynamic Programming** approach eliminates recursion and achieves $O(n)$ complexity with a straightforward tabulation method, making it efficient for computing Fibonacci numbers well beyond the limits of naive recursion. The **Matrix Exponentiation** method further optimizes performance, reducing complexity to $O(\log n)$ by leveraging matrix power calculations, making it one of the fastest approaches for large values of n . Lastly, the **Iterative Approach**, being the most memory-efficient with $O(n)$ complexity, offers an optimal balance between efficiency and simplicity.

The **Binet's Formula** method provides an almost constant-time solution with $O(1)$ complexity by using the Golden ratio and its mathematical properties. This makes it one of the fastest ways to compute Fibonacci numbers. However, due to floating-point arithmetic and rounding errors, Binet's method can lose precision when computing large Fibonacci numbers.

Based on these findings, the **Naive Recursive** method is only practical for small inputs, while the **Memoized Recursive** and **Bottom-Up Dynamic Programming** methods provide better performance for moderately large values. The **Matrix Exponentiation** method proves to be the most efficient for large Fibonacci numbers, and the **Iterative Approach** is a strong alternative for balanced execution time and minimal memory usage. The **Binet's Formula** method is highly efficient but should be used with caution due to precision limitations. Future optimizations could further enhance these methods, especially by refining the dynamic programming and matrix exponentiation approaches to further reduce computation time.