MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Inginerie Software şi Automatică

BOTNARI MARIA-ELENA, FAF-232

REPORT

Laboratory Work nr. 1:
Study and Empirical Analysis of Algorithms for Determining Fibonacci N-th Term

Elaborated by: **Botnari Maria-Elena,** FAF-232
Checked by: **Fiştic Cristofor,** *university assistant*

Chișinău – 2025

Algorithm Analysis

Objective:

The goal of this analysis is to study and evaluate different algorithms used to determine the n-th Fibonacci number.

Tasks:

1. Implement at least 3 algorithms for calculating the Fibonacci n-th term.
2. Define the input properties for the algorithms to be analyzed.
3. Choose the metrics for comparing the algorithms.
4. Empirically analyze the algorithms.
5. Present the results from the analysis.
6. Draw conclusions based on the results.

Theoretical Notes:

In addition to the mathematical analysis of algorithm complexity, empirical analysis is another useful approach. This method helps with:

- Estimating the complexity class of an algorithm.
- Comparing the efficiency of multiple algorithms solving the same problem.
- Evaluating the efficiency of different implementations of the same algorithm.
- Understanding how well an algorithm performs on a specific computer.

Empirical analysis involves the following steps:

1. Define the analysis goal – What do you want to find out from this analysis?
2. Choose the efficiency metric – This could be the number of operations performed or the total execution time of the algorithm.
3. Set the input data properties – Decide on the size of the input data or any other specific characteristics relevant to the analysis.
4. Implement the algorithm in a programming language.

5. Generate multiple sets of input data to test the algorithm.
6. Run the algorithm on each set of input data.
7. Analyze the results – Look at the data to draw conclusions.

Choosing the Efficiency Metric:

The choice of the efficiency metric depends on the purpose of the analysis. For example:

- If the goal is to estimate the complexity class or check the accuracy of a theoretical prediction, then the number of operations performed is a good measure.
- If the goal is to understand the actual performance of an algorithm on a specific system, then execution time is the better choice.

After running the program with the test data, the results are recorded. To analyze them, you can calculate synthetic values like mean or standard deviation or create graphs showing the relationship between problem size and efficiency.

Introduction:

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones. For example: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, and so on.

The sequence is often associated with Leonardo Fibonacci, an Italian mathematician born around 1170 A.D. He was initially known as Leonardo of Pisa, and later historians called him Fibonacci (meaning "son of the Bonacci family") to distinguish him from another famous Leonardo of Pisa. However, some suggest that Fibonacci didn't "discover" the sequence. Keith Devlin, the author of *Finding Fibonacci*, mentions that ancient Sanskrit texts using the Hindu-Arabic numeral system refer to Fibonacci-like numbers, and these were written long before Fibonacci's time.

In 1202, Leonardo of Pisa published *Liber Abaci*, a book that introduced the Fibonacci sequence to the Western world. The book served as a guide for tradespeople, showing how to use the Hindu-Arabic number system to calculate profits, losses, and other financial matters.

Traditionally, the Fibonacci sequence was found by simply adding the two preceding numbers. But with the development of computer science and algorithms, many methods have been created to compute Fibonacci numbers more efficiently. These methods can be categorized into four groups: Recursive Methods, Dynamic Programming Methods, Matrix Power Methods, and the Binet Formula Methods. Each of these can be implemented in both naive and optimized forms to improve performance.

As mentioned earlier, the performance of an algorithm can be evaluated mathematically (using logical analysis) or empirically (based on experimental observations). In this laboratory, we will analyze four 6 algorithms empirically.

Comparison Metric:

The comparison metric for this analysis will be the **execution time** of each algorithm (T(n)).

Input Format:

Each algorithm will receive two sets of numbers representing the Fibonacci term positions we want to compute. The first set will have smaller numbers to test the recursive method (5, 7, 10, 12, 15, 17, 20, 22, 25, 27, 30). The second set will include larger numbers for comparing the performance of the other algorithms (10, 25, 50, 100, 250, 501, 631, 794, 1000, 1259, 1585, 1995, 2512, 3162, 3981, 5012, 6310, 7943, 10000, 12589, 15849).

Implementation

All six Fibonacci algorithms will be implemented in their basic forms using Python and analyzed empirically based on the time taken for each to complete. While the overall trends of the results may be similar to other studies, the specific performance relative to the input size may differ depending on the device's memory and processing capabilities. The error margin for this analysis will be set at 2.5 seconds, based on experimental measurements.

1. Naive recursion

The recursive method is one of the simplest ways to compute Fibonacci numbers but is also the least efficient. It calculates the n-th term by first computing its two predecessors and then adding them together. However, it does this by calling itself multiple times, often repeating calculations for the same term. This redundancy increases memory usage and execution time exponentially.

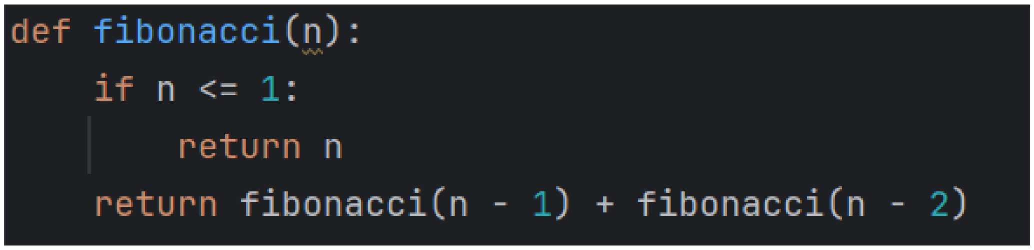
Algorithm description:

This method of finding the n-th Fibonacci term can be described in the following pseudocode:

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

Implementation:

The Fibonacci sequence can be defined recursively as follows:



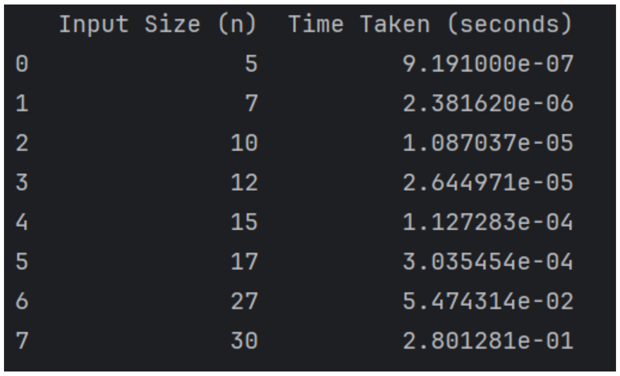
```python
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Fig. 1: Function that computes the n-th Fibonacci number recursively

Each time the function is called for F(n), it also calls itself to compute F(n-1) and F(n-2). These calls, in turn, generate their own recursive calls. This process creates a branching tree structure where each node (representing a function call) has two child nodes.

Since the number of calls doubles at each step, the total number of operations grows exponentially, leading to a time complexity of O(2^n). This means that for large values of n, the function becomes extremely slow.

Results:



```
   Input Size (n)  Time Taken (seconds)
0               5          9.191000e-07
1               7          2.381620e-06
2              10          1.087037e-05
3              12          2.644971e-05
4              15          1.127283e-04
5              17          3.035454e-04
6              27          5.474314e-02
7              30          2.801281e-01
```

Fig. 2: Table with execution times for different inputs for the recursive approach

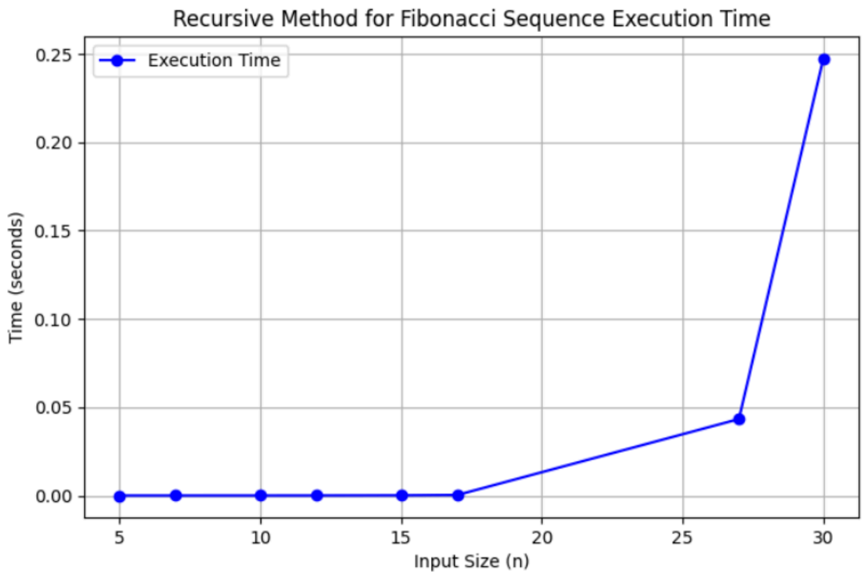


Fig. 3: Graph with execution times for different inputs for the recursive approach

Because this method repeats many calculations, it becomes impractical for large values of *n*. The time needed to compute Fibonacci numbers increases quickly, making this approach much less efficient than other methods. Even for small inputs, we can see that the execution time grows rapidly. For example, when the input increased from 25 to 30, a relatively small jump, the execution time still rose significantly. This shows how quickly the inefficiency becomes a problem as *n* gets larger.

2. Memoization (top-down dynamic programming)

Memoization improves the recursive approach by storing previously computed Fibonacci numbers in a lookup table (memo). Before computing F(n), the function first checks if n is already in memo. If it is, the stored result is returned, avoiding unnecessary calculations. If n is not in memo, F(n) is computed recursively and then stored for future use.

Unlike naive recursion, which repeatedly recalculates the same values, memoization significantly reduces the number of recursive calls. This optimization brings the time complexity down from O(2^n) to O(n) since each Fibonacci number is computed only once. The function is called n - 1 times, and each call does O(1) work, resulting in O(n) total operations.

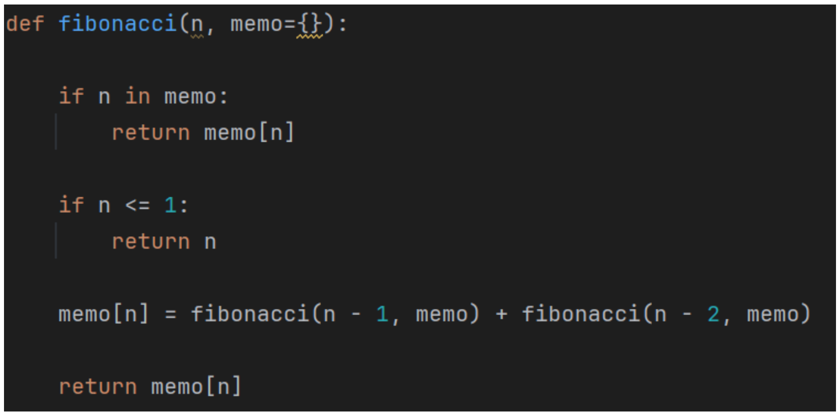
Algorithm Description:

This method of finding the n-th Fibonacci term can be described in the following pseudocode:

```
fibonacci(n, memo = {}):
 if n is in memo:
    return memo[n]
if $n \leq 1$:
    return n memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo)
return memo[n]
```

Implementation:

This method is implemented in the following way:

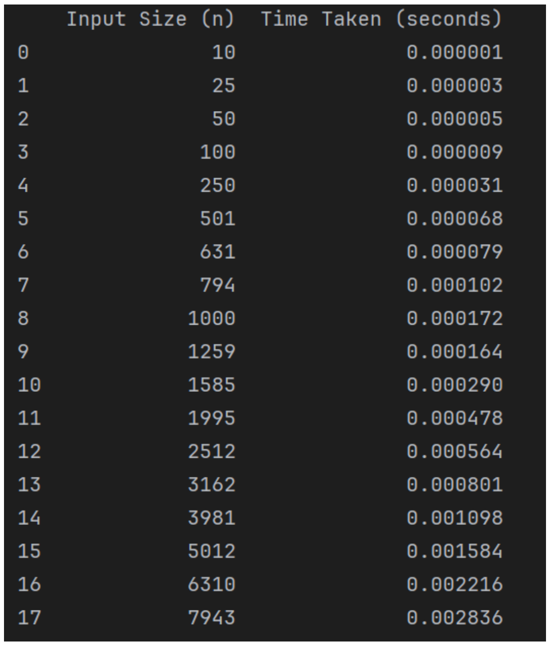


```python
def fibonacci(n, memo={}):

    if n in memo:
        return memo[n]

    if n <= 1:
        return n

    memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo)

    return memo[n]
```

Fig. 4: Function that computes the n-th Fibonacci number using the memoization method

This approach is similar to naive recursion but avoids redundant calculations by storing computed Fibonacci numbers in memo. Instead of recalculating F(n) multiple times, the function checks if the result is already stored and reuses it.

Results:



```
    Input Size (n)  Time Taken (seconds)
0               10              0.000001
1               25              0.000003
2               50              0.000005
3              100              0.000009
4              250              0.000031
5              501              0.000068
6              631              0.000079
7              794              0.000102
8             1000              0.000172
9             1259              0.000164
10            1585              0.000290
11            1995              0.000478
12            2512              0.000564
13            3162              0.000801
14            3981              0.001098
15            5012              0.001584
16            6310              0.002216
17            7943              0.002836
```

Fig. 5: Table with execution times for different inputs for the memoization method

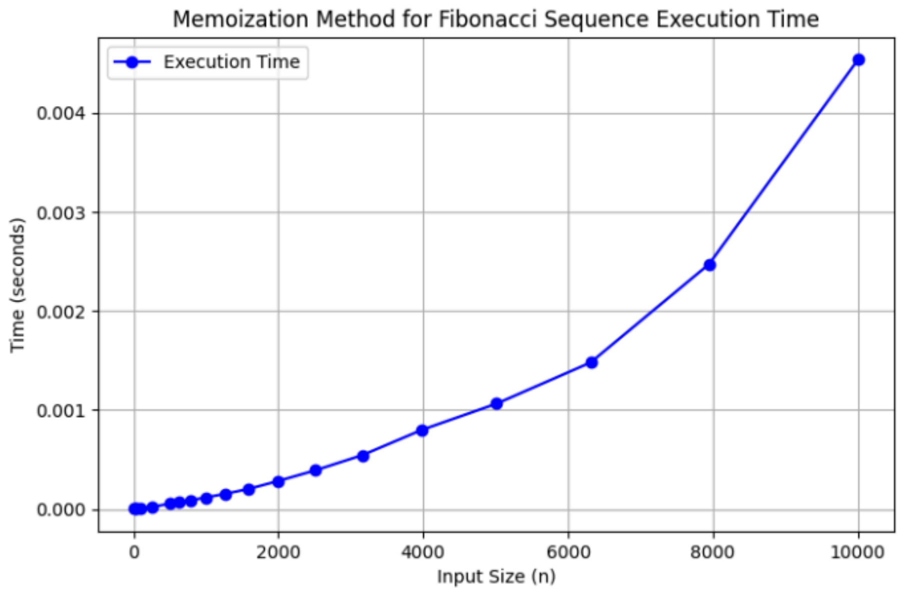


Fig. 6: Graph with execution times for different inputs for the memoization method

This method is significantly faster than naive recursion because it greatly reduces the number of recursive calls. In naive recursion, the execution time for n=30 is 62 times longer than the memoization method when n=10,000. This shows that memoization is not only much more efficient for large inputs but also for smaller ones, making it a far better choice overall.
However, it still has drawbacks:

- It requires extra memory to store previously computed values.
- It still relies on recursion, which can lead to stack overflow for very large n.

Overall, while memoization is a significant improvement over naive recursion, more efficient methods like iteration or matrix exponentiation provide even better performance.

3. Tabulation (bottom-up dynamic programming)

The tabulation method builds the Fibonacci sequence from the smallest values up to F(n). It starts with the known base cases: F(0) = 0 and F(1) = 1, then iteratively computes the next Fibonacci numbers until reaching F(n).

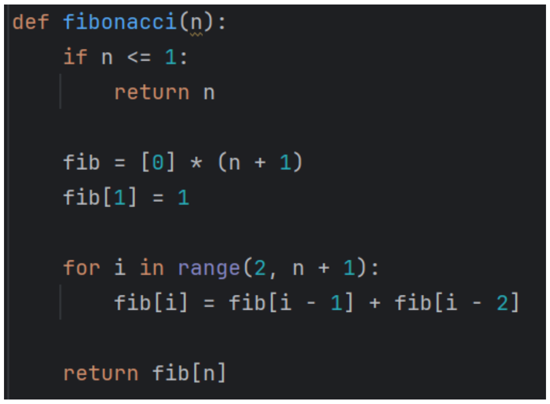
Unlike memoization, tabulation does not use recursion, meaning it avoids the overhead of multiple function calls. Instead, it stores all Fibonacci numbers in an array and calculates them one by one. This makes it more efficient in terms of execution time compared to the recursive methods.

Algorithms Description:

This method of finding the n-th Fibonacci term can be described in the following pseudocode:

```
fibonacci_series(n):
fib[n]
for i = 0 to n:
  fib[i] = 0
   i++

fib[1] = 1

for i = 2 to n:
   fib[i] = fib[i - 1] + fib[i - 1]
return fib[n]
```

Implementation:



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

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

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

    return fib[n]
```

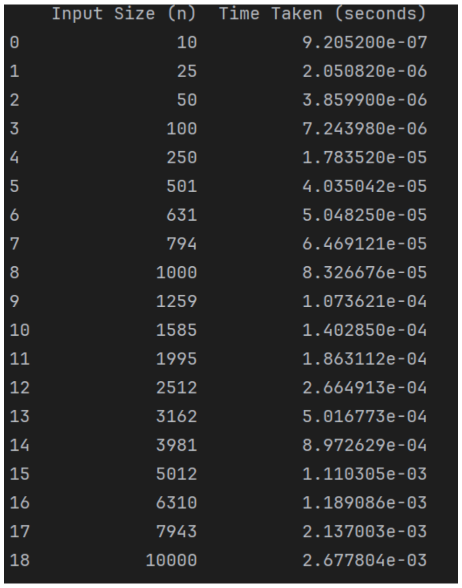
Fig. 7: Function that computes the n-th Fibonacci number using the tabulation method

The loop runs n - 1 times, and since each iteration performs one addition and one assignment, the number of operations is approximately 2(n - 1).

Unlike memoization, this method does not use recursion, making it faster because it avoids the overhead of function calls.

However, it stores all Fibonacci numbers in an array, leading to higher memory usage compared to the iterative approach.

Results:



```
    Input Size (n)  Time Taken (seconds)
0               10          9.205200e-07
1               25          2.050820e-06
2               50          3.859900e-06
3              100          7.243980e-06
4              250          1.783520e-05
5              501          4.035042e-05
6              631          5.048250e-05
7              794          6.469121e-05
8             1000          8.326676e-05
9             1259          1.073621e-04
10            1585          1.402850e-04
11            1995          1.863112e-04
12            2512          2.664913e-04
13            3162          5.016773e-04
14            3981          8.972629e-04
15            5012          1.110305e-03
16            6310          1.189086e-03
17            7943          2.137003e-03
18           10000          2.677804e-03
```

Fig. 8: Table with execution times for different inputs for the tabulation method

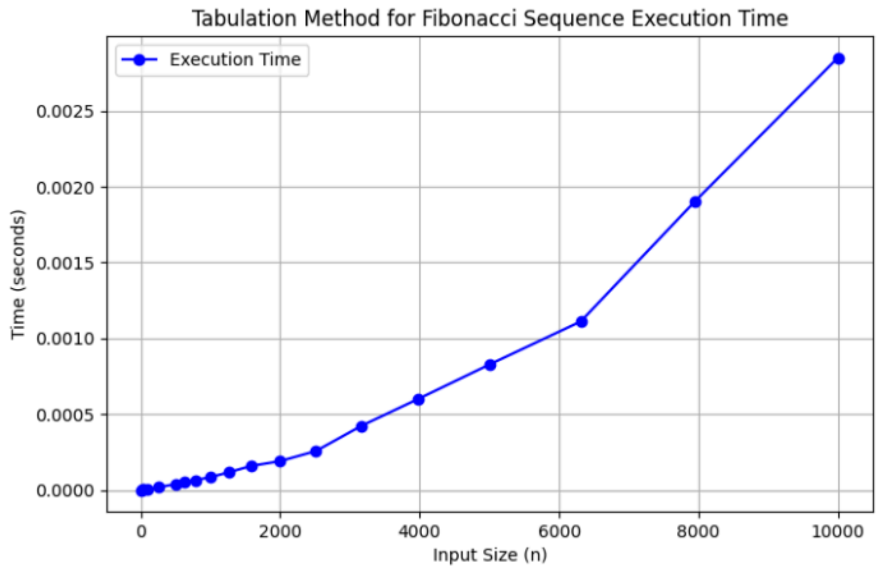


Fig. 9: Graph with execution times for different inputs for the tabulation method

Tabulation uses more memory than memoization, but it is generally faster because it avoids the overhead of recursion. For both small and large inputs, the difference in execution time between tabulation and memoization is minimal. When the input is less than 200, their runtimes are roughly around x^{-6}, and even for larger inputs, such as 10,000, they remain close to x^{-3}.

However, tabulation is still slightly slower than the iterative approach, which only keeps track of the last two Fibonacci numbers instead of storing the entire sequence. This makes the iterative method more memory-efficient while maintaining high speed.

Overall, tabulation strikes a good balance between simplicity and performance. But for extremely large values of *n*, the iterative method is the better choice, as it significantly reduces memory usage without sacrificing speed.

4. Iterative approach

Unlike the tabulation method, this approach only stores the last two Fibonacci numbers, reducing memory usage. It computes all Fibonacci numbers up to the n-th term but keeps track of only the last necessary values. This method doesn't use recursion and avoids storing the entire sequence, making it faster than the classic recursive approach.

Algorithm Description:

This method of finding the n-th Fibonacci term can be described in the following pseudocode:

```
fibonacci(n):
two_steps_back = 0
 one_step_back = 1
for i = 2 to n:
    current = two_steps_back + one_step_back
    two_steps_back = one_step_back
    one_step_back = current
return current
```

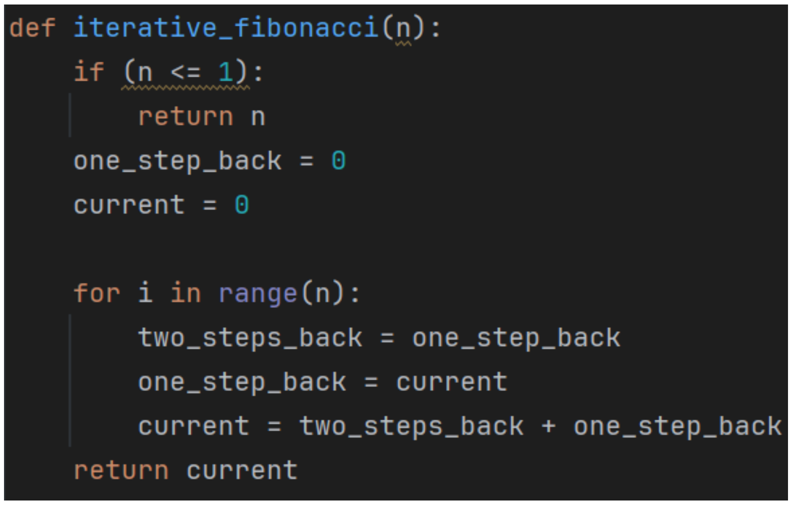
Implementation:

In each iteration, the following operations are performed:

- Assigning two_steps_back = one_step_back (1 operation)
- Assigning one_step_back = current (1 operation)
- Computing current = two_steps_back + one_step_back (1 operation)

Since the loop runs n - 1 times, the total number of operations is 3(n - 1).

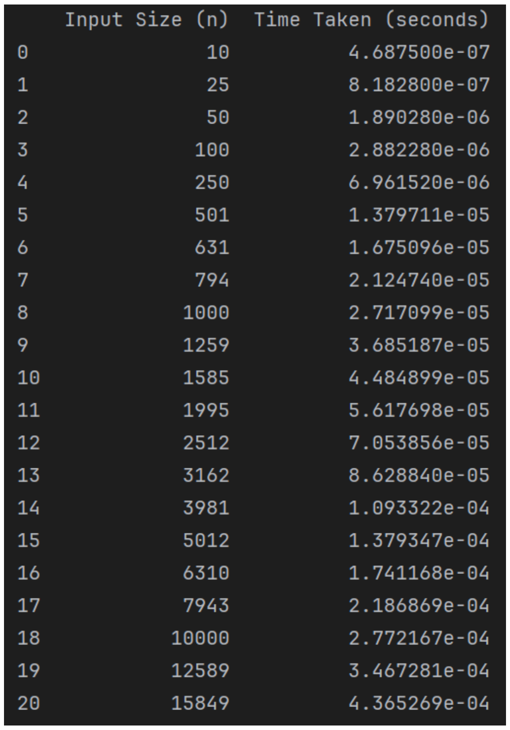
This method avoids recursion and only stores the last two Fibonacci numbers, using significantly less memory compared to the tabulation approach.



```
def iterative_fibonacci(n):
    if (n <= 1):
        return n
    one_step_back = 0
    current = 0

    for i in range(n):
        two_steps_back = one_step_back
        one_step_back = current
        current = two_steps_back + one_step_back
    return current
```

Fig. 10: Function that computes the n-th Fibonacci number using the iterative method

Results:



```
    Input Size (n)  Time Taken (seconds)
0               10          4.687500e-07
1               25          8.182800e-07
2               50          1.890280e-06
3              100          2.882280e-06
4              250          6.961520e-06
5              501          1.379711e-05
6              631          1.675096e-05
7              794          2.124740e-05
8             1000          2.717099e-05
9             1259          3.685187e-05
10            1585          4.484899e-05
11            1995          5.617698e-05
12            2512          7.053856e-05
13            3162          8.628840e-05
14            3981          1.093322e-04
15            5012          1.379347e-04
16            6310          1.741168e-04
17            7943          2.186869e-04
18           10000          2.772167e-04
19           12589          3.467281e-04
20           15849          4.365269e-04
```

Fig. 11: Table with execution times for different inputs for the iterative method

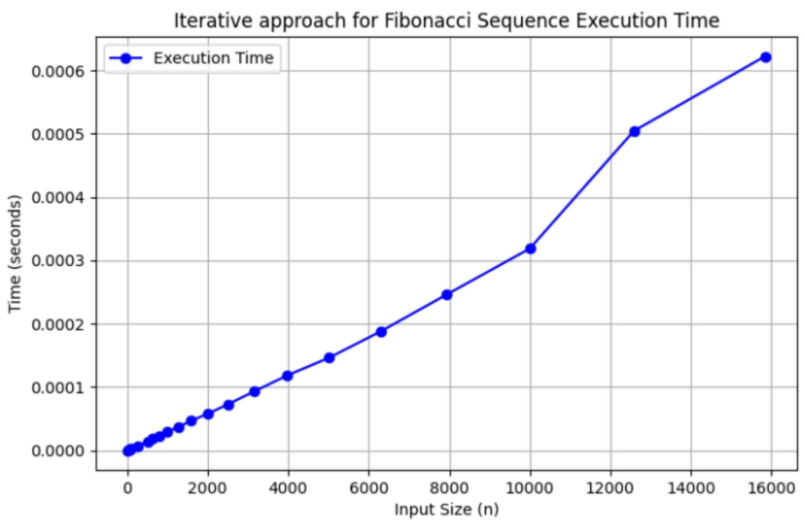


Fig. 12: Graph with execution times for different inputs for the iterative method

This method is faster than recursive approaches because it avoids unnecessary calculations and does not rely on recursion. For small inputs, both the iterative and tabulation methods perform very quickly, making the difference between them negligible. However, as the input size grows, the gap becomes more noticeable. For instance, when n=10.000, the tabulation method is about eight times slower than the iterative approach. With even larger values of n, the tabulation method continues to lag behind the iterative approach.

That said, it is still slower than more advanced techniques like the Binet formula and matrix exponentiation. Even so, it remains a reliable and efficient way to compute Fibonacci numbers while keeping memory usage low.

5. Matrix Exponentiation

The Fibonacci sequence can be defined with matrix exponentiation:

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

Matrix exponentiation provides an efficient way to compute Fibonacci numbers. The Fibonacci sequence can be represented as a matrix, and by raising the matrix to the power of n, we can find the n-th Fibonacci number. Using exponentiation by squaring allows us to compute the matrix power in O(log n) time, making this method very fast for large inputs.

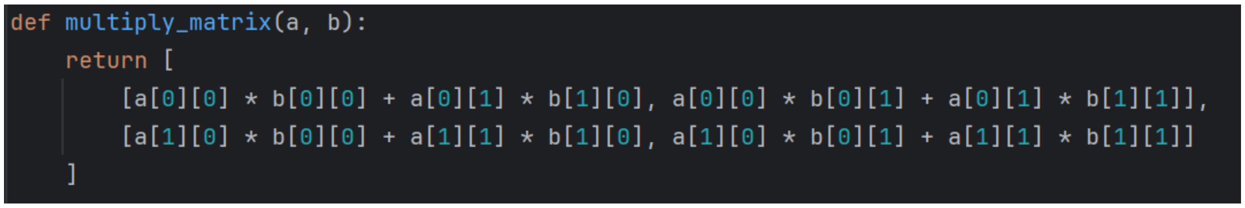
Algorithm Description:

This method of finding the n-th Fibonacci term can be described in the following pseudocode:

```
fibonacci_series(n):
base_matrix = [[1, 1], [1, 0]]
result = matrix_power(base_matrix, n-1)
return result[0][0]
```

Implementation:

The first step is to multiply two 2x2 matrices. Here's how it can be done:



```python
def multiply_matrix(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]]
    ]
```

Fig. 13: Function that multiplies two matrices

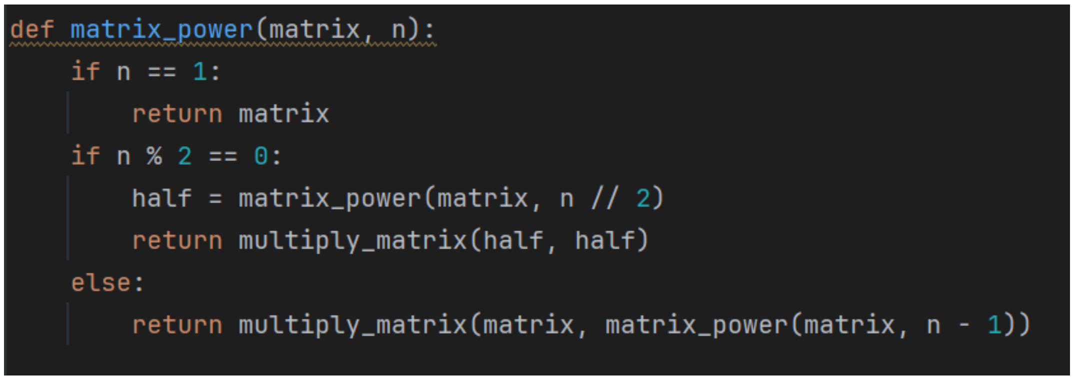
Next, we raise the matrix to the power of n using Exponentiation by Squaring:

- If n = 1, return the matrix itself.
- If n is even, compute the matrix raised to the power n/2 and multiply it with itself.

- If n is odd, compute the matrix raised to the power n-1 and then multiply it with the base matrix.

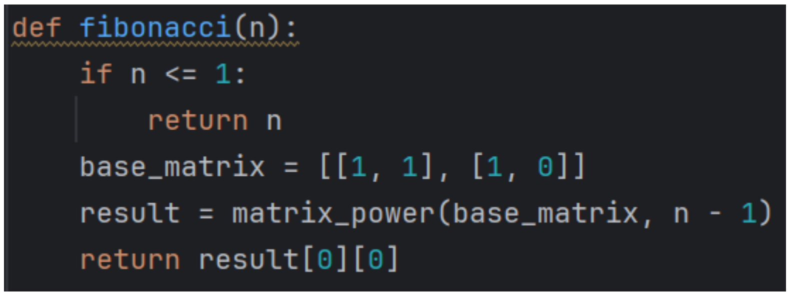
The function that raises the matrix to the power n computes the Fibonacci number using the matrix representation. This is done by recursively multiplying the matrix until it reaches the correct power.

By using this method, the number of matrix multiplications is reduced from O(n) to O(log n).



```python
def matrix_power(matrix, n):
    if n == 1:
        return matrix
    if n % 2 == 0:
        half = matrix_power(matrix, n // 2)
        return multiply_matrix(half, half)
    else:
        return multiply_matrix(matrix, matrix_power(matrix, n - 1))
```

Fig. 14: Function that raises the matrix to a power n



```python
def fibonacci(n):
    if n <= 1:
        return n
    base_matrix = [[1, 1], [1, 0]]
    result = matrix_power(base_matrix, n - 1)
    return result[0][0]
```

Fig. 15: Function that computes the n-th Fibonacci number using the matrix exponentiation method

Why result[0][0] gives the n-th Fibonacci Number?

We can rewrite the formula above as:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \times \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}$$

The final matrix multiplication gives us:

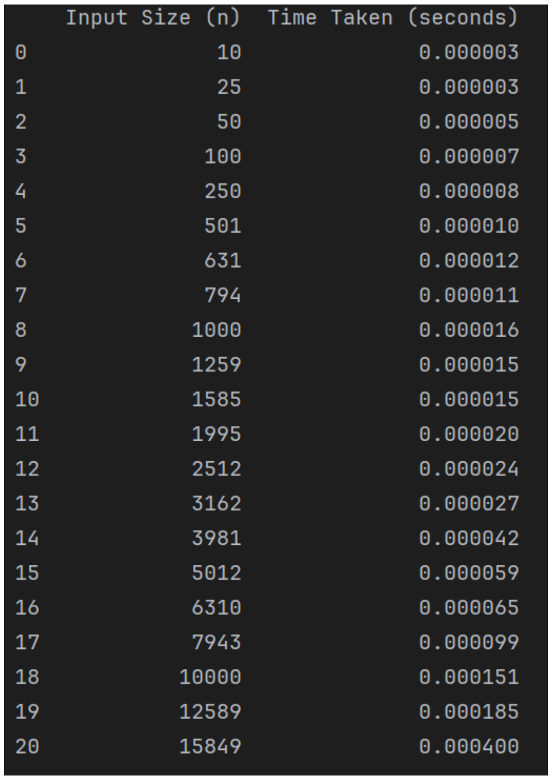
$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

which results in:

$$F_n = a \times 1 + b \times 0 = a$$

Thus, result[0][0] gives the n-th Fibonacci number.

Results:



```
    Input Size (n)  Time Taken (seconds)
0               10              0.000003
1               25              0.000003
2               50              0.000005
3              100              0.000007
4              250              0.000008
5              501              0.000010
6              631              0.000012
7              794              0.000011
8             1000              0.000016
9             1259              0.000015
10            1585              0.000015
11            1995              0.000020
12            2512              0.000024
13            3162              0.000027
14            3981              0.000042
15            5012              0.000059
16            6310              0.000065
17            7943              0.000099
18           10000              0.000151
19           12589              0.000185
20           15849              0.000400
```

Fig. 16: Table with execution times for different inputs for the matrix exponentiation method

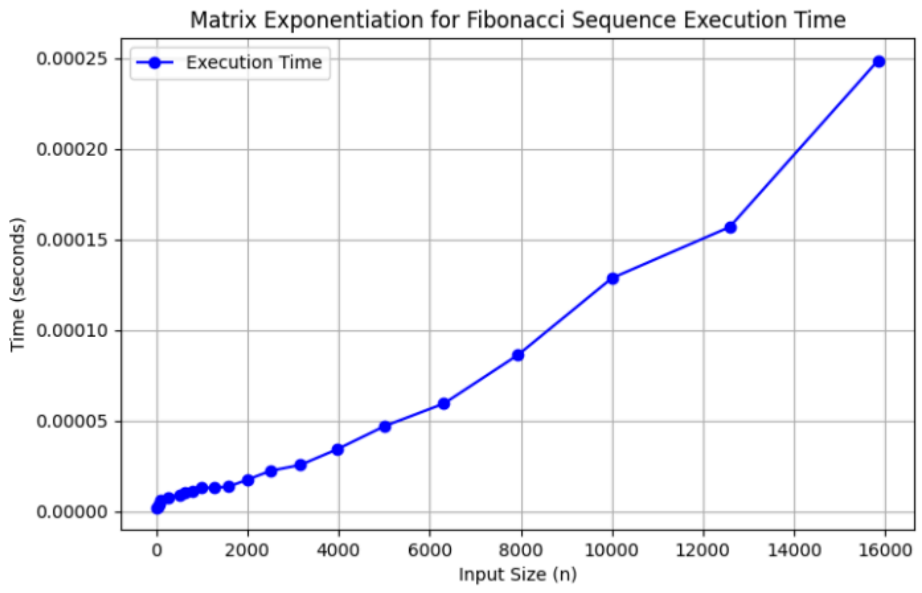


Fig. 17: Graph with execution times for different inputs for the matrix exponentiation method

Matrix exponentiation is an incredibly efficient method, even for large inputs. However, it's not faster than the iterative method, since this method does something as time-consuming as multiplying matrices. In fact, the iterative method, with small values, it outperforms the matrix exponentiation approach a little. For values of n up to 2000, the execution time remains in the range of $x^{-6} - x^{-5}$, whereas the iterative method takes roughly $x^{-7} - x^{-4}$, making it a little slower. Even from values bigger than 2000 up until 15000, the iterative method is still mostly ten times faster than the matrix exponentiation one.

However, this method is much faster than the recursive methods discussed above. One of the key reasons for its efficiency is that it computes Fibonacci numbers in O(log n) time, which is extremely fast. In fact, its speed is comparable to the approach using Binet's formula. This makes matrix exponentiation a powerful and practical method for quickly calculating Fibonacci numbers, even for very large values of n.

6. Binet’s Formula

The Binet Formula provides a direct way to compute the n-th Fibonacci number without needing to compute previous terms. It expresses the n-th Fibonacci number using n and the golden ratio (denoted as phi). This formula shows that the ratio of two consecutive Fibonacci numbers approaches the golden ratio as n increases.

The formula for the n-th Fibonacci term is:

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

Where phi (the golden ratio) is:

$$\phi = \frac{1+\sqrt{5}}{2}$$

The time complexity of the Binet Formula is O(1) because it calculates the Fibonacci number in a constant number of operations, regardless of the size of n. Unlike other methods that involve loops or recursion, the Binet Formula computes the result directly.
While this method is extremely fast, its accuracy can be problematic for large n. This is because sqrt(5) is an irrational number, and when using floating-point arithmetic, rounding errors can accumulate. Over time, these errors can lead to incorrect results, especially for very large values of n

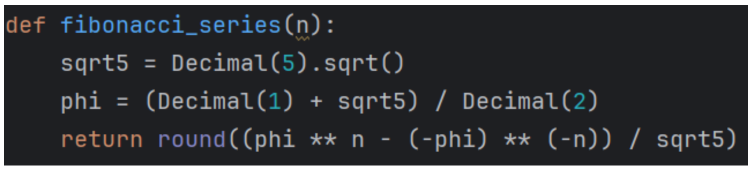
Algorithm Description:

This method of finding the n-th Fibonacci term can be described in the following pseudocode:

```
fibonacci_series(n):
sqrt5 = sqrt(5)
phi = (1 + sqrt5)/2
return (phi^n - (-phi)^(-n)) / sqrt5
```

Implementation:

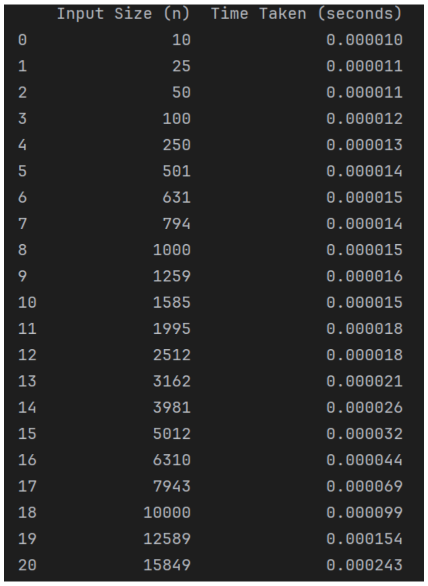
In Python, for better precision, we can use the Decimal data type from the decimal module. This allows for high-precision arithmetic and avoids the limitations of floating-point numbers, which can be inaccurate for large values of n. For very large n, the phi^n term grows exponentially, and using float might lead to rounding errors or overflow.



```python
def fibonacci_series(n):
    sqrt5 = Decimal(5).sqrt()
    phi = (Decimal(1) + sqrt5) / Decimal(2)
    return round((phi ** n - (-phi) ** (-n)) / sqrt5)
```

Fig. 18: Function that computes the n-th Fibonacci number using Binet's Formula

Results:



```
    Input Size (n)  Time Taken (seconds)
0               10              0.000010
1               25              0.000011
2               50              0.000011
3              100              0.000012
4              250              0.000013
5              501              0.000014
6              631              0.000015
7              794              0.000014
8             1000              0.000015
9             1259              0.000016
10            1585              0.000015
11            1995              0.000018
12            2512              0.000018
13            3162              0.000021
14            3981              0.000026
15            5012              0.000032
16            6310              0.000044
17            7943              0.000069
18           10000              0.000099
19           12589              0.000154
20           15849              0.000243
```

Fig. 19: Table with execution times for different inputs for the method that used Binet's Formula

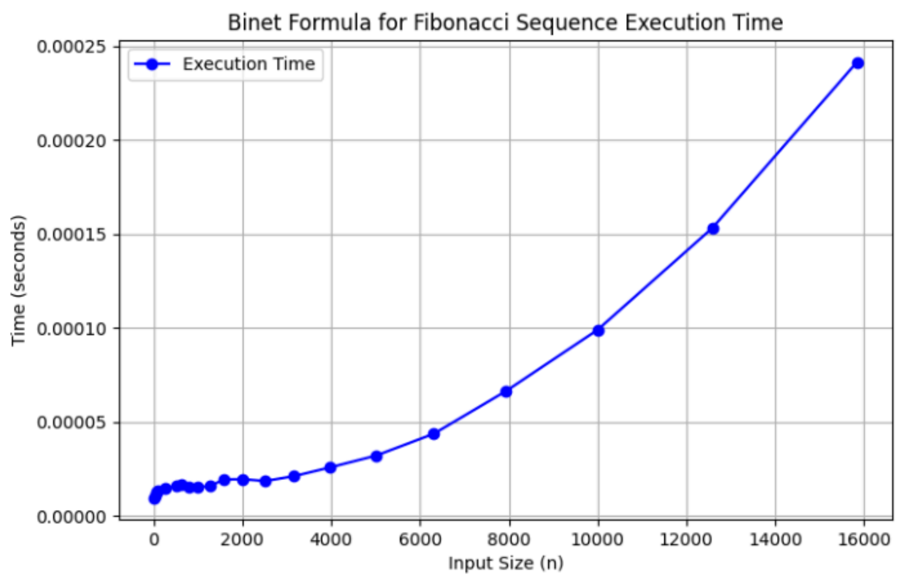


Fig. 19: Graph with execution times for different inputs for the method that used Binet's Formula

The Binet Formula and the Matrix Exponentiation method have similar execution times, meaning both are equally efficient in terms of speed. Since the Binet Formula has an O(1) time complexity, it remains fast even for large values of *n*. However, as *n* increases, the formula becomes less accurate due to floating-point precision errors. For extremely large Fibonacci numbers, these rounding errors can lead to significant inaccuracies. While the Binet Formula is a quick way to compute Fibonacci numbers, it is not the best choice for very high values of *n*, as the results may become unreliable.

Conclusion

Through empirical analysis, this paper tested six different methods for computing Fibonacci numbers, comparing their efficiency in terms of both accuracy and time complexity. The goal was to identify the most appropriate method for different scenarios and determine possible areas for improvement.

1. **Naive Recursion**: While simple to implement, this method is highly inefficient due to its exponential time complexity ($O(2^n)$). It is suitable for calculating Fibonacci numbers for small values of n (up to around 30), but becomes impractical for larger values due to excessive computation time and memory usage.
2. **Memoization**: By storing previously computed results, this method significantly improves performance compared to naive recursion. Its time complexity is reduced to O(n), making it more suitable for larger values of n. However, it still relies on recursion, which can lead to stack overflow for very large inputs.
3. **Tabulation**: This method, which builds the Fibonacci sequence iteratively from the bottom-up, avoids recursion and uses O(n) time complexity. It is more efficient than both naive recursion and memoization, but it requires more memory since it stores all Fibonacci numbers up to the n-th term. Nonetheless, it remains a strong choice for computing Fibonacci numbers efficiently.
4. **Iterative Approach**: This method further optimizes the tabulation approach by storing only the last two Fibonacci numbers, reducing memory usage while maintaining a time complexity of O(n). It provides an efficient solution for computing Fibonacci numbers with minimal memory usage and is faster than naive recursion, memoization and matrix exponentiation.
5. **Matrix Exponentiation**: By using matrix exponentiation, Fibonacci numbers can be computed in O(log n) time. This method is highly efficient for large values of n, outperforming the dynamic programming methods in terms of speed.
6. **Binet's Formula**: The Binet Formula provides a direct calculation of the n-th Fibonacci number with O(1) time complexity. It is extremely fast, but its accuracy diminishes for very large n due to rounding errors in floating-point arithmetic. For most practical purposes, it is fast but may not be reliable for very large values of n.

In conclusion, each method has its strengths and weaknesses. For small values of n, naive recursion can be used, though memoization and tabulation offer improvements. For larger values, matrix exponentiation and the iterative approach are more efficient. The Binet formula is best for quick calculations of smaller Fibonacci numbers but is not recommended for larger ones due to precision issues. Optimizations can further improve the performance of dynamic programming and matrix methods, making them suitable for even larger inputs.