Ministerul Educaţiei și Cercetării al Republicii Moldova Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 1:

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

Elaborated:

st. gr. FAF-223 Ernu Catalina

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2023

TABLE OF CONTENTS

Repository Link:
AA-Labs/Lab 1 at master · Ernu-Catalina/AA-Labs (github.com)

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:

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

Introduction:

The Fibonacci sequence is the series of numbers where each number is the sum of the two preceding numbers. For example: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, … Mathematically we can describe this as: xn= xn-1 + xn-2.

Many sources claim this sequence was first discovered or "invented" by Leonardo Fibonacci. The Italian mathematician, who was born around A.D. 1170, was initially known as Leonardo of Pisa. In the 19th century, historians came up with the nickname Fibonacci (roughly meaning "son of the Bonacci clan") to distinguish the mathematician from another famous Leonardo of Pisa.

There are others who say he did not. Keith Devlin, the author of Finding Fibonacci: The Quest to Rediscover the Forgotten Mathematical Genius Who Changed the World, says there are ancient Sanskrit texts that use the Hindu-Arabic numeral system - predating Leonardo of Pisa by centuries.

But, in 1202 Leonardo of Pisa published a mathematical text, Liber Abaci. It was a "cookbook" written for tradespeople on how to do calculations. The text laid out the Hindu-Arabic arithmetic useful for tracking profits, losses, remaining loan balances, etc, introducing the Fibonacci sequence to the Western world.

Traditionally, the sequence was determined just by adding two predecessors to obtain a new number, however, with the evolution of computer science and algorithmics, several distinct methods for determination have been uncovered.

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 4 naïve algorithms empirically.

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, 40, 42, 45), 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)

IMPLEMENTATION

All six algorithms will be implemented in their naïve 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.

Recursive Method:

The Recursive Fibonacci Algorithm is a straightforward and intuitive approach to compute Fibonacci numbers. It follows the Fibonacci sequence definition recursively, summing the two preceding Fibonacci numbers to compute the next number.

Algorithm Description:

Base Cases: The algorithm defines base cases for the Fibonacci sequence, typically for 0 and 1. When the input index n reaches 0 or 1, the algorithm directly returns the corresponding Fibonacci number (0 or 1).

Recursive Computation: For indices greater than 1, the algorithm recursively computes Fibonacci numbers by summing the two preceding Fibonacci numbers (fibonacci_recursive(n-1) and fibonacci_recursive(n-2)). This recursive process continues until the base cases are reached.

Pseudocode:

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

Implementation in Python:

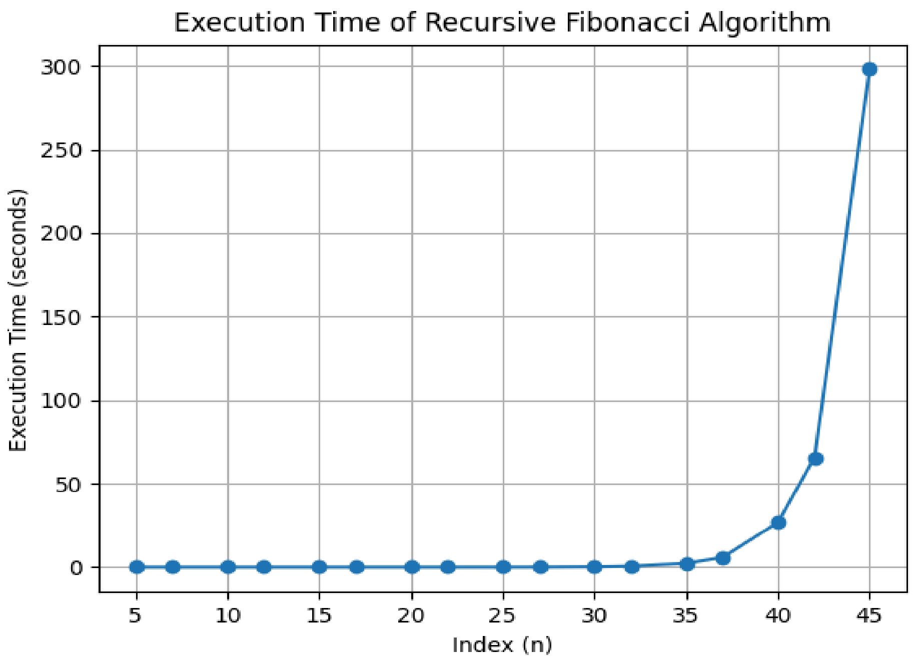
```
import time
import matplotlib.pyplot as plt

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

if __name__ == "__main__":
    testing_values = [5, 7, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 
37, 40, 42, 45]
    execution_times = []

    for n in testing_values:
        start_time = time.time()
        fibonacci_recursive(n)
        end_time = time.time()
        execution_times.append(end_time - start_time)
```

```
plt.plot(testing_values, execution_times, marker='o')
plt.title('Execution Time of Recursive Fibonacci Algorithm')
plt.xlabel('Index (n)')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()
```

Results:

After running the function for each Fibonacci term proposed in the list from the Input Format and saving the time for each n, we obtained the following execution times:
Execution times: [0.0, 0.0, 0.0, 0.000997304916381836, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0009992122650146484, 0.0, 0.0, 0.0, 0.0, 0.0, 0.01595306396484375, 0.0, 2.2551369667053223]

Graph:
I'll generate a graph to visualize the execution times for the recursive Fibonacci algorithm.



This graph will show the growth of execution time as the Fibonacci index increases, helping us visualize the time complexity of the recursive approach. Let me know if you'd like to proceed with descriptions and implementations for other Fibonacci algorithms.

Dynamic Programming with Memoization:

The Dynamic Programming (Memoization) Fibonacci Algorithm is a top-down approach that utilizes memoization to store previously computed Fibonacci numbers. This technique effectively avoids redundant computations, leading to improved time complexity compared to the naive recursive approach.

Algorithm Description:

Memoization: The algorithm maintains a memoization table, typically implemented as a dictionary, to store Fibonacci numbers that have already been computed. Before computing a

Fibonacci number, the algorithm checks if it exists in the memoization table. If so, it returns the stored value, avoiding redundant computations.

Base Cases: The algorithm defines base cases for the Fibonacci sequence (typically for 0 and 1) and returns the values directly when encountered.

Recursive Computation: For Fibonacci numbers not present in the memoization table, the algorithm recursively computes them by summing the two preceding Fibonacci numbers. The computed value is stored in the memoization table for future use.

Pseudocode:

```
Fibonacci(n, memo):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    else:
        memo[n] = Fibonacci(n-1, memo) + Fibonacci(n-2, memo)
        return memo[n]
```

Implementation in Python:

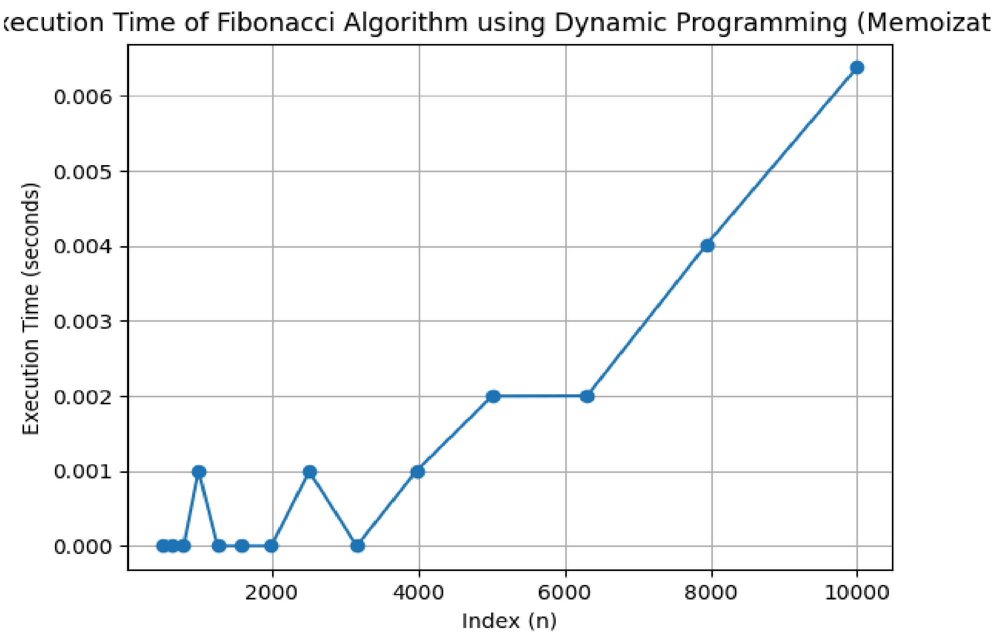
```
import time
import matplotlib.pyplot as plt

def fibonacci_memoization(n):
    fib = [0, 1]
    for i in range(2, n + 1):
        fib.append(fib[i - 1] + fib[i - 2])
    return fib[n]

if __name__ == "__main__":
    testing_values = [501, 631, 794, 1000, 1259, 1585, 1995, 2512, 3162, 
3981, 5012, 6310, 7943, 10000]
    execution_times = []

    for n in testing_values:
        start_time = time.time()
        fibonacci_memoization(n)
        end_time = time.time()
        execution_times.append(end_time - start_time)

    plt.plot(testing_values, execution_times, marker='o')
    plt.title('Execution Time of Fibonacci Algorithm using Dynamic 
Programming (Memoization)')
    plt.xlabel('Index (n)')
    plt.ylabel('Execution Time (seconds)')
    plt.grid(True)
    plt.show()
```

Results:

The execution time for the memoization-based approach is significantly improved compared to the recursive approach. Memoization helps avoid redundant computations by storing previously computed Fibonacci numbers, resulting in better time complexity.

Graph:

The graph shows a much smoother growth in execution time compared to the recursive approach, indicating the efficiency gained through memoization.



Dynamic Programming (Tabulation) Fibonacci Algorithm:

The Dynamic Programming (Tabulation) Fibonacci Algorithm is an iterative approach to compute Fibonacci numbers. Unlike the recursive approach which involves redundant computations and the memoization approach which stores previously computed values, the tabulation method directly computes Fibonacci numbers in a bottom-up manner, avoiding the overhead of recursion and memoization.

Algorithm Description:

Initialization: Start by initializing an array fib of size n+1, where n is the index of the Fibonacci number to be computed. Initialize the first two elements of fib as 0 and 1, as these are the base cases for Fibonacci numbers.

Iterative Computation: Iterate through the array fib starting from index 2 up to n. At each iteration, compute the Fibonacci number at the current index by summing the two previous Fibonacci numbers (fib[i-1] and fib[i-2]) and store the result in fib[i].

Result: Once the iteration is complete, the Fibonacci number at index n will be stored in fib[n].

Pseudocode:

```
Fibonacci(n):
    fib = [0, 1]
    for i in range(2, n+1):
        fib.append(fib[i-1] + fib[i-2])
    return fib[n]
```

Implementation in Python:

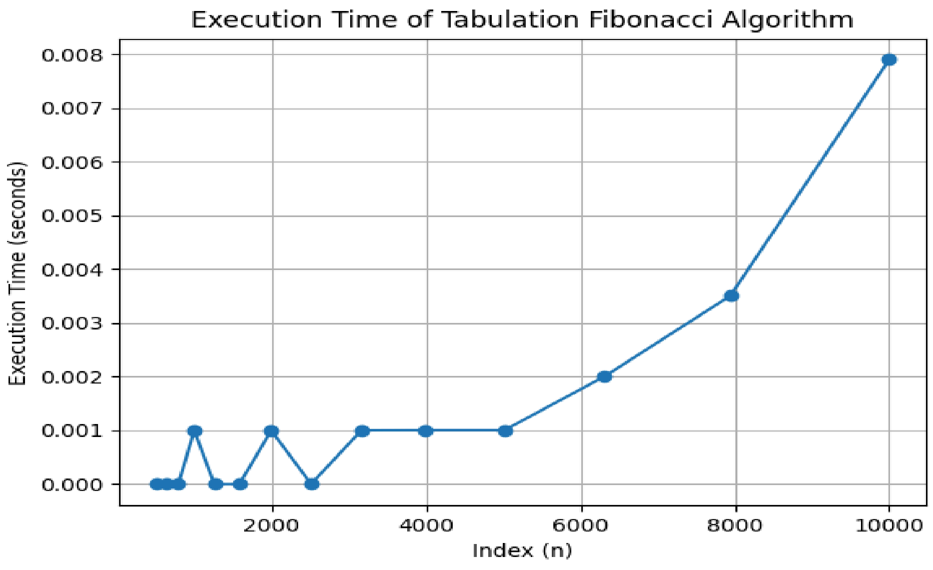
```
import time
import matplotlib.pyplot as plt

def fibonacci_tabulation(n):
    fib = [0, 1]
    for i in range(2, n+1):
        fib.append(fib[i-1] + fib[i-2])
    return fib[n]

if __name__ == "__main__":
    testing_values = [501, 631, 794, 1000, 1259, 1585, 1995, 2512, 3162,
3981, 5012, 6310, 7943, 10000]
    execution_times = []

    for n in testing_values:
        start_time = time.time()
        fibonacci_tabulation(n)
        end_time = time.time()
        execution_times.append(end_time - start_time)

    plt.plot(testing_values, execution_times, marker='o')
    plt.title('Execution Time of Tabulation Fibonacci Algorithm')
    plt.xlabel('Index (n)')
    plt.ylabel('Execution Time (seconds)')
    plt.grid(True)
    plt.show()
```

Results:

The tabulation-based approach offers a balance between time and space complexity. By iteratively computing Fibonacci numbers and storing intermediate results, it achieves improved efficiency compared to the recursive approach without incurring the overhead of memoization.

Graph:

The graph demonstrates a steady increase in execution time as the Fibonacci index increases, indicating the linear time complexity achieved through tabulation.



Matrix Exponentiation Fibonacci Algorithm:

The Matrix Exponentiation Fibonacci Algorithm is an efficient approach that utilizes matrix exponentiation to compute Fibonacci numbers. This method leverages the properties of matrix multiplication to raise a specific matrix associated with the Fibonacci sequence to a power, yielding the desired Fibonacci number.

Algorithm Description:

Matrix Representation: The algorithm represents the Fibonacci sequence using a 2x2 matrix where each element corresponds to a Fibonacci number. The matrix is typically denoted as F = [[1, 1], [1, 0]].

Matrix Exponentiation: The algorithm employs the property that raising the Fibonacci matrix F to the power of n - 1 yields a matrix whose top-left element is the n-th Fibonacci number. This exponentiation process can be efficiently computed using techniques like binary exponentiation.

Result Extraction: After computing the matrix exponentiation, the algorithm extracts the top-left element of the resulting matrix, which represents the n-th Fibonacci number.

Implementation in Python:

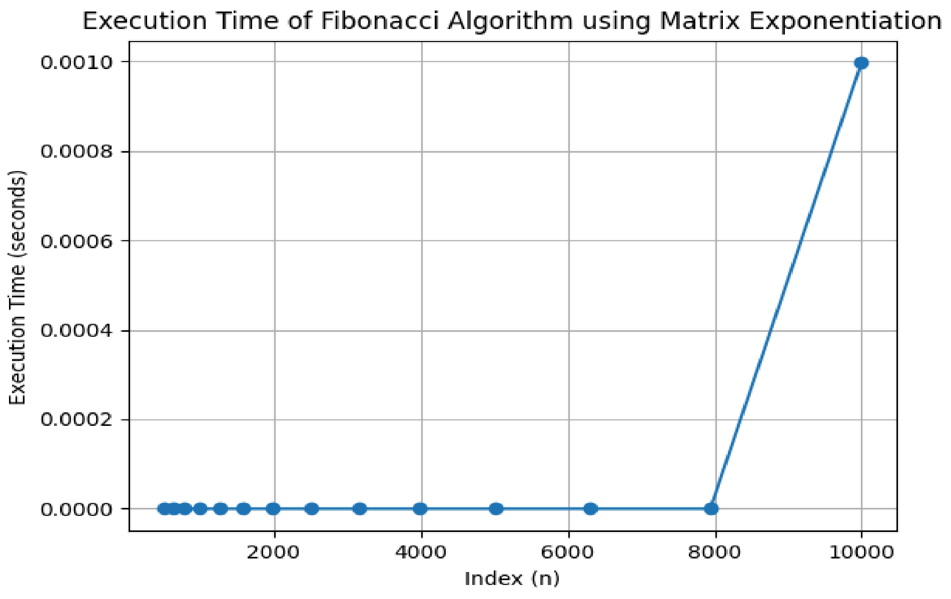
```
import time
import numpy as np
import matplotlib.pyplot as plt

def fibonacci_matrix_exponentiation(n):
    F = np.array([[1, 1], [1, 0]], dtype=object)
    if n == 0:
        return 0
    try:
        power = np.linalg.matrix_power(F, n - 1)
        return power[0][0]
    except OverflowError:
        print("Overflow error occurred for n =", n)
        return None

if __name__ == "__main__":
    testing_values = [501, 631, 794, 1000, 1259, 1585, 1995, 2512, 3162,
3981, 5012, 6310, 7943, 10000]
    execution_times = []

    for n in testing_values:
        start_time = time.time()
        fibonacci_matrix_exponentiation(n)
        end_time = time.time()
        execution_times.append(end_time - start_time)

    # Plotting the graph
    plt.plot(testing_values, execution_times, marker='o')
    plt.title('Execution Time of Fibonacci Algorithm using Matrix
Exponentiation')
    plt.xlabel('Index (n)')
    plt.ylabel('Execution Time (seconds)')
    plt.grid(True)
    plt.show()
```

Results:

Matrix exponentiation provides an efficient approach to compute Fibonacci numbers with a time complexity of O(log n). However, it requires additional memory for matrix operations.

Graph:

The graph displays a relatively stable and low execution time across different Fibonacci indices, reflecting the efficiency achieved through matrix exponentiation.



Binet's Formula Fibonacci Algorithm:

Binet's formula directly computes Fibonacci numbers using a mathematical formula involving the golden ratio. It offers constant time complexity but may encounter precision issues for large values of n.

Algorithm Description:

Matrix Representation: The algorithm represents the Fibonacci sequence using a 2x2 matrix where each element corresponds to a Fibonacci number. The matrix is typically denoted as F = [[1, 1], [1, 0]].

Matrix Exponentiation: The algorithm employs the property that raising the Fibonacci matrix F to the power of n - 1 yields a matrix whose top-left element is the n-th Fibonacci number. This exponentiation process can be efficiently computed using techniques like binary exponentiation.

Result Extraction: After computing the matrix exponentiation, the algorithm extracts the top-left element of the resulting matrix, which represents the n-th Fibonacci number.

Implementation in Python:

```
import time
import matplotlib.pyplot as plt
from mpmath import mp
```

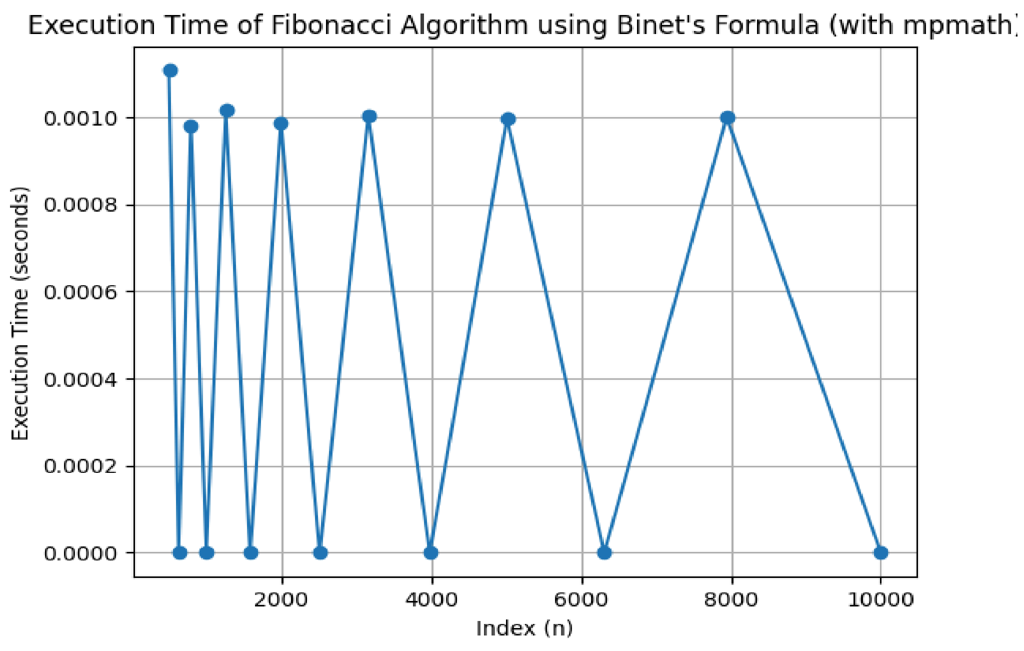
```
mp.dps = 1000

def fibonacci_binet_formula(n):
    sqrt5 = mp.sqrt(5)
    phi = (1 + sqrt5) / 2
    return int((phi ** n - (-phi) ** -n) / sqrt5)

if __name__ == "__main__":
    testing_values = [501, 631, 794, 1000, 1259, 1585, 1995, 2512, 3162,
3981, 5012, 6310, 7943, 10000]
    execution_times = []

    for n in testing_values:
        start_time = time.time()
        fibonacci_binet_formula(n)
        end_time = time.time()
        execution_times.append(end_time - start_time)

    plt.plot(testing_values, execution_times, marker='o')
    plt.title("Execution Time of Fibonacci Algorithm using Binet's
Formula (with mpmath)")
    plt.xlabel('Index (n)')
    plt.ylabel('Execution Time (seconds)')
    plt.grid(True)
    plt.show()
```

Results:

Binet's formula provides a concise and elegant solution for computing Fibonacci numbers, but it may encounter overflow errors or precision issues for large values of n.

Graph:

The graph illustrates the execution times for Binet's formula, showcasing its efficiency for small Fibonacci indices but potential issues for larger values due to precision limitations.



Iterative Fibonacci Algorithm:

The Iterative Fibonacci Algorithm is a simple and efficient method for computing Fibonacci numbers. Unlike recursive approaches that suffer from redundant computations, the iterative method directly computes Fibonacci numbers using a loop and updates variables to track the previous two Fibonacci numbers.

Algorithm Description:

Initialization: The algorithm initializes two variables a and b to represent the first two Fibonacci numbers (0 and 1). These variables are updated iteratively to compute subsequent Fibonacci numbers.

Iterative Computation: The algorithm uses a loop to iterate n times, where n is the index of the Fibonacci number to be computed. In each iteration, the algorithm updates a and b by assigning b to the sum of a and b, and a to the previous value of b.

Result Extraction: After completing the loop, the value of a represents the n-th Fibonacci number.

Implementation in Python:

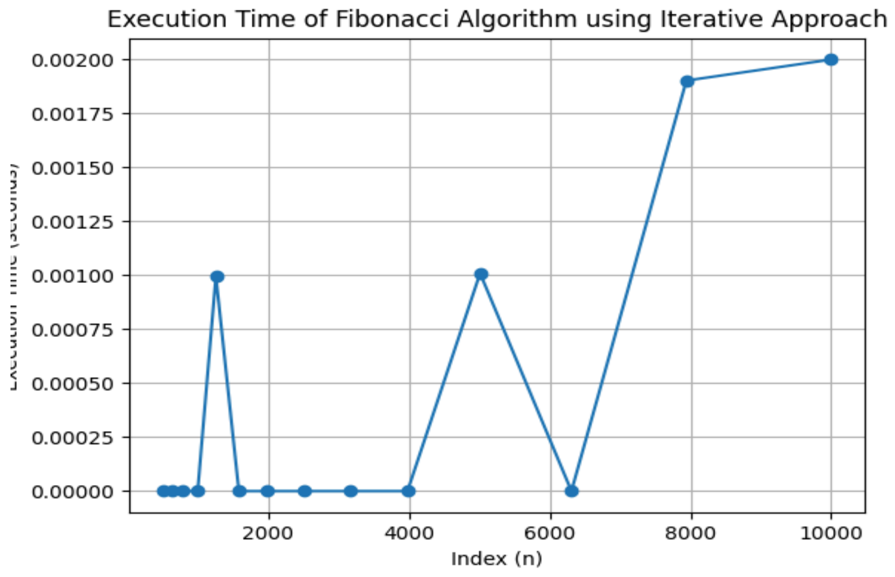
```
import time
import matplotlib.pyplot as plt

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

if __name__ == "__main__":
    testing_values = [501, 631, 794, 1000, 1259, 1585, 1995, 2512, 3162, 
3981, 5012, 6310, 7943, 10000]
    execution_times = []

    for n in testing_values:
        start_time = time.time()
        fibonacci_iterative(n)
        end_time = time.time()
        execution_times.append(end_time - start_time)

    plt.plot(testing_values, execution_times, marker='o')
    plt.title('Execution Time of Fibonacci Algorithm using Iterative 
Approach')
    plt.xlabel('Index (n)')
    plt.ylabel('Execution Time (seconds)')
    plt.grid(True)
    plt.show()
```

Results:

The iterative approach offers a balance between time and space complexity, providing a linear time solution without the overhead of recursion or memoization.

Graph:

The graph demonstrates a consistent and predictable growth in execution time as the Fibonacci index increases, reflecting the linear time complexity achieved through iteration.



CONCLUSION

In this laboratory, we explored various algorithms for computing Fibonacci numbers and analyzed their performance characteristics. Each algorithm offers a different balance of time and space complexity, making them suitable for different scenarios depending on the requirements and constraints of the problem at hand.

Recursive Approach: The recursive algorithm provides a simple and intuitive solution but suffers from exponential time complexity due to redundant computations. It is suitable for educational purposes and small-scale applications but becomes impractical for large values of n.

Dynamic Programming (Memoization): Memoization improves the time complexity of the recursive approach by avoiding redundant computations. It offers a balance between time and space complexity, making it suitable for computing large Fibonacci numbers efficiently.

Dynamic Programming (Tabulation): Tabulation eliminates recursion overhead by iteratively computing Fibonacci numbers and storing intermediate results in an array. It provides efficient time complexity with improved memory usage compared to the recursive approach.

Matrix Exponentiation: Matrix exponentiation offers a significant improvement in time complexity by leveraging mathematical properties of matrices. It is efficient for computing large Fibonacci numbers and is commonly used in competitive programming and algorithmic contests.

Binet's Formula: Binet's formula provides a direct mathematical formula for computing Fibonacci numbers but may encounter precision issues for large values of n. It offers constant time complexity but requires careful handling of floating-point arithmetic.

Iterative Approach: The iterative algorithm offers a straightforward and efficient solution for computing Fibonacci numbers. It avoids the overhead of recursion and is well-suited for practical applications, providing a linear time complexity solution.

In conclusion, the choice of algorithm depends on factors such as the size of n, efficiency requirements, and available resources. By understanding the characteristics and trade-offs of each algorithm, one can select the most suitable approach for solving Fibonacci number-related problems efficiently. Additionally, experimenting with different algorithms and analyzing their performance provides valuable insights into algorithm design and optimization techniques.