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-221 **Mîndrescu Dragomir**
Verified:
asist. univ. **Fiștic Cristofor**

Chișinău - 2024

TABLE OF CONTENTS

ALGORITHM ANALYSIS

Objective

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

Tasks :

1. Develop at least three different algorithms to compute the n-th term in the Fibonacci sequence.
2. Define the input format criteria for analyzing these algorithms.
3. Choose a metric for comparing the performance of these algorithms.
4. Conduct an empirical analysis of the algorithms.
5. Summarize the findings from the data collected.
6. Draw conclusions from the laboratory exercise.

Theoretical Notes:

Instead of solely relying on mathematical complexity analysis, empirical analysis offers an alternative for assessing an algorithm's performance. This approach is valuable for gaining initial insights into an algorithm's complexity class, comparing the efficiency of multiple algorithms or implementations for the same task, and understanding the performance of an algorithm on specific hardware.

In empirical analysis, the process typically involves establishing the analysis objective, selecting an efficiency metric (such as the number of operations or execution time), defining the relevant properties of the input data, implementing the algorithm in a programming language, generating various input data sets, executing the program with these data sets, and analyzing the results. The choice of efficiency measure depends on the analysis goal, whether it's understanding complexity class, verifying theoretical estimates, or assessing implementation behavior.

Introduction:

The Fibonacci sequence is a series where each number is the sum of the two preceding ones, starting from 0 and 1. It's mathematically expressed as $x_n = x_{n-1} + x_{n-2}$ Although commonly attributed to Leonardo Fibonacci, who introduced it to the Western world through his 1202 publication "Liber Abaci", earlier references exist in ancient Sanskrit texts. Over time, various methods for calculating Fibonacci numbers have emerged, categorized into Recursive, Dynamic Programming, Matrix Power, and Binet Formula Methods. This project will focus on the empirical analysis of four naive algorithms across these categories.

Comparison Metric:

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

Input Format:

The algorithms will be tested with two sets of numbers indicating the Fibonacci sequence's order. The first set (5, 7, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 37, 40, 42, 45) caters to the recursive method's limitations, while the second set (501, 631, 794, 1000, 1259, 1585, 1995, 2512, 3162, 3981, 5012, 6310, 7943, 10000, 12589, 15849) allows for a broader comparison of the other algorithms.

IMPLEMENTATION

Recursive Method:

The recursive method, often seen as the least efficient approach, computes the n-th term of the Fibonacci sequence by first calculating its preceding terms and then summing them up. This method involves the function calling itself multiple times, thereby performing the same computation for the same term more than once. This repetition not only uses extra memory but also, in theory, doubles its execution time due to the redundant operations.

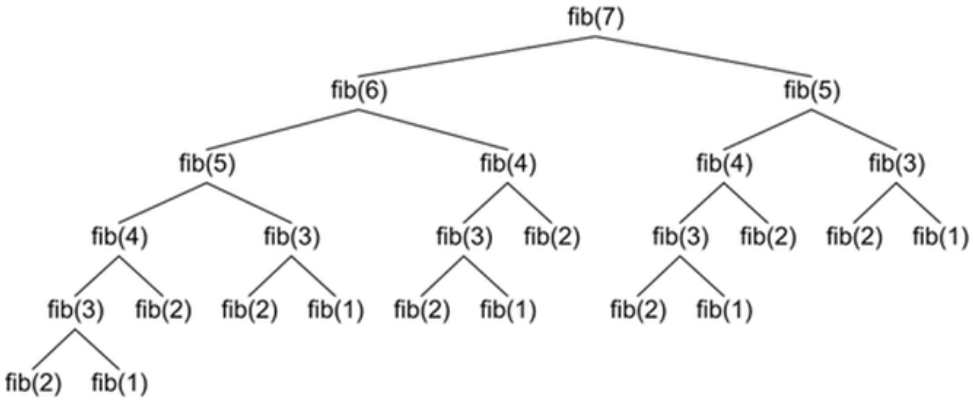

Figure 1 Fibonacci Recursion

Algorithm Description:

The naive recursive Fibonacci method follows the algorithm as shown in the next pseudocode:

```
Function Fibonacci(n)
    If n <= 1 Then
        Return n
    Else
        Return Fibonacci(n-1) + Fibonacci(n-2)
End Function
```

Implementation:

```swift
import Foundation

func fibonacciRecursive(n: Int) -> Int {
    if n == 1 {
        return 0
    } else if n == 2 {
        return 1
    }
    return fibonacciRecursive(n: n - 1) + fibonacciRecursive(n: n - 2)
}
```

Figure 2 Fibonacci Recursion in Swift

Results:

After executing the function for each Fibonacci term suggested in the list provided in the initial input format and recording the time taken for each term, we collected the following outcomes.

```
Term 2 | Term 5 | Term 7 | Term 12 | Term 23 | Term 38 | Term 45 | Term 46
------------------------------------------------------------------------
0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00013 s | 0.15911 s | 4.22081 s | 6.84296 s
0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00011 s | 0.14491 s | 4.21131 s | 6.81214 s
0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00012 s | 0.14494 s | 4.21028 s | 6.86871 s
Program ended with exit code: 0
```

Figure 3 Results for first set of inputs

Figure 3 illustrates the table displaying the results for the initial set of inputs. The top row signifies the Fibonacci nth term for which the functions were executed. Beginning from the second row, the values represent the elapsed time in seconds from when the function commenced running to when it concluded. It's worth noting that among the functions tested, the Recursive Method Fibonacci function exhibited an increasing time trend for these initial nth terms.

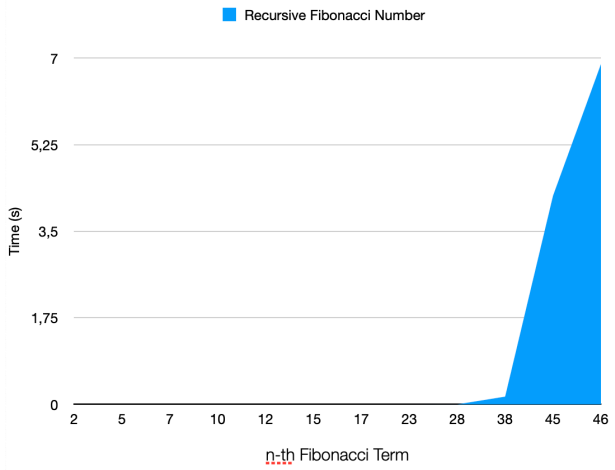

Figure 4 Graph of Recursive Fibonacci Number

Not only that, but also in the graph in Figure 4 that shows the growth of the time needed for the
operations, we may easily see the spike in time complexity that happens after the 42nd term, leading us to
deduce that the Time Complexity is exponential. T(2^n).

Memoization Method:

Memoization is a programming technique used to optimize recursive algorithms by storing previously computed results for future reference. In the context of finding the nth term of the Fibonacci sequence, memoization involves storing Fibonacci numbers as they are calculated, reducing redundant computations and significantly improving performance. By caching results, the time complexity is reduced from exponential to linear, making the algorithm more efficient, particularly for larger values of n. This approach enhances the overall scalability and speed of Fibonacci number calculations.

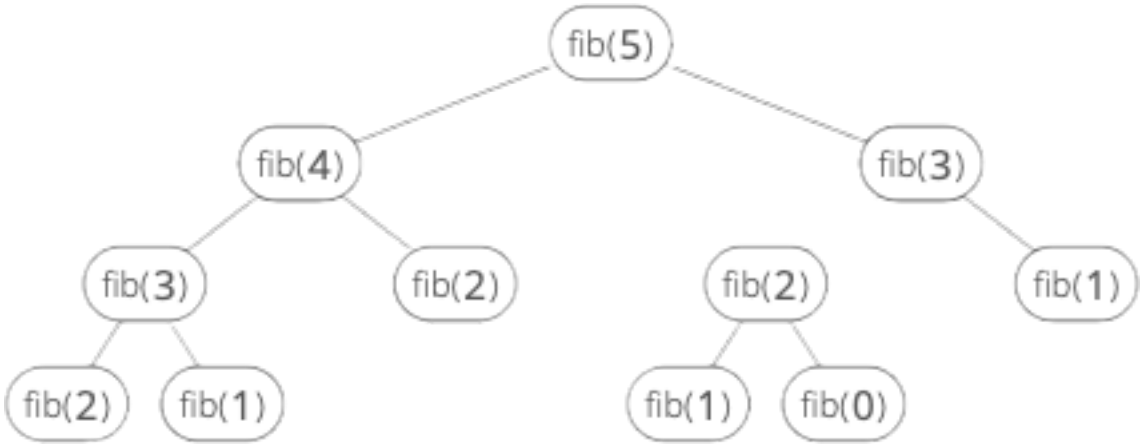

Figure 5 Fibonacci Memoization

Algorithm Description:

The Memoization Fibonacci method follows the algorithm as shown in the next pseudocode:

```
function fib_memo(n, memo):
    if n <= 1:
        return n
    if memo[n] is not defined:
        memo[n] = fib_memo(n-1, memo) + fib_memo(n-2, memo)
    return memo[n]

function find_nth_fibonacci(n):
    memo = array of size n+1 initialized with undefined values
    return fib_memo(n, memo)
```

Implementation:

```
8  import Foundation
9
10 var memo = [Int: Double]()
11 func fibonacciMemoization(n: Int) -> Double {
12     if n <= 1 {
13         return Double(n)
14     }
15     if let result = memo[n] {
16         return result
17     }
18     memo[n] = fibonacciMemoization(n: n - 1) + fibonacciMemoization(n: n - 2)
19     return memo[n]!
20 }
```

Figure 6 Fibonacci memoization in Swift

Figure 6 illustrates the table displaying the results for the initial set of inputs. The top row signifies the Fibonacci nth term for which the functions were executed. Beginning from the second row, the values represent the elapsed time in seconds from when the function commenced running to when it concluded. It's worth noting that among the functions tested, the Recursive Method Fibonacci function exhibited an increasing time trend for these initial nth terms.

Results:

```
Term 23 | Term 28 | Term 38 | Term 45 | Term 100 | Term 200 | Term 300 | Term 400 | Term 500 | Term 600 | Term 20000 | Term 30000
-------------------------------------------------------------------------------------------------------------------------------
0.00005 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00001 s | 0.00003 s | 0.00002 s | 0.00003 s | 0.00002 s | 0.00002 s | 0.00499 s | 0.00257 s
0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s
0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s
Program ended with exit code: 0
```

Figure 7 Results for the first set of inputs

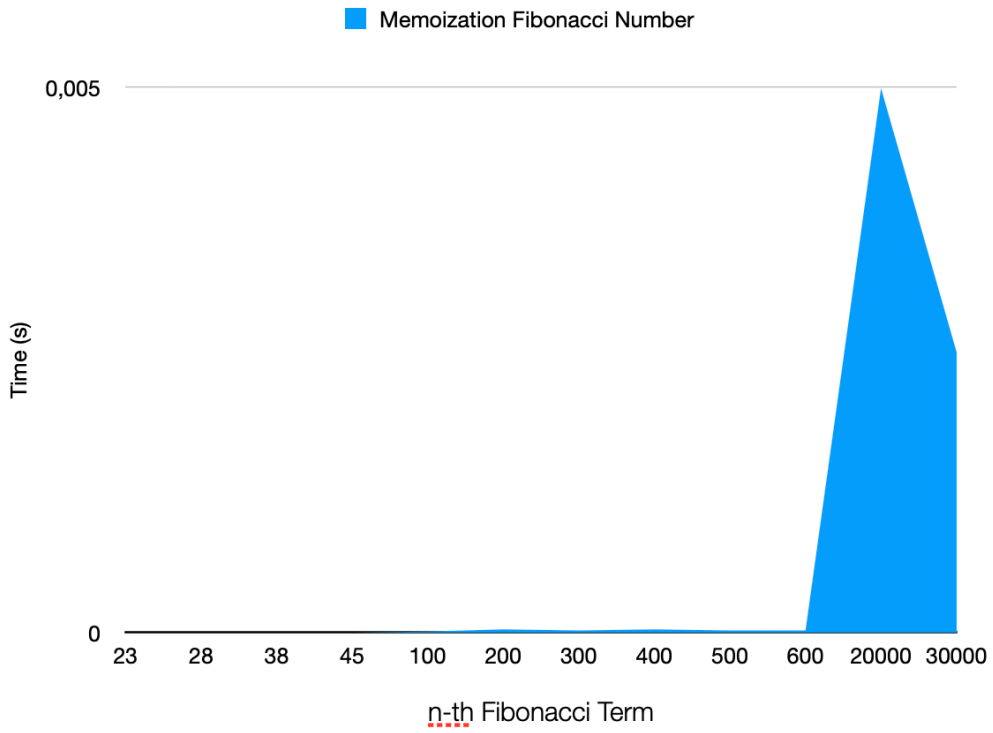

Figure 8 Graph of Memoization Fibonacci Number

Tabulation Method:

Tabulation is another dynamic programming technique used to optimize algorithms by storing precomputed results in a table, typically in an iterative manner. In the context of finding the nth Fibonacci number, tabulation involves filling up a table with Fibonacci numbers starting from the base cases and iteratively calculating each subsequent Fibonacci number until reaching the desired nth number.

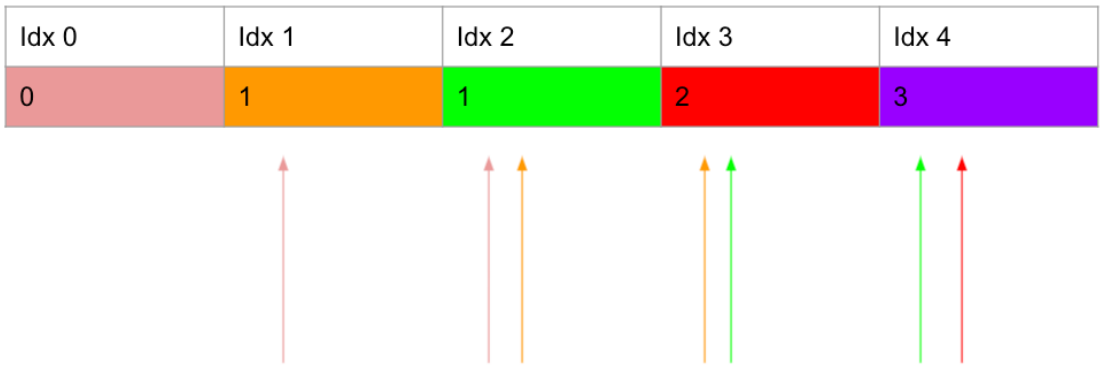

Figure 9 Fibonacci Tabulation

This method avoids recursive calls altogether and directly computes the Fibonacci numbers in a bottom-up fashion, typically resulting in better space efficiency compared to memoization. Tabulation is particularly useful when the recursive structure of a problem can be expressed as a straightforward iterative process, making it easier to implement and understand. It offers an alternative approach to dynamic programming that complements memoization and can be more suitable for certain types of problems or programming environments.

Algorithm Description:

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

Implementation:

```swift
import Foundation

func fibonacciTabulation(n: Int) -> Int {
    if n <= 1 {
        return n
    }
    var fibNumbers = [0, 1] + Array(repeating: 0, count: n - 1)
    for i in 2...n {
        fibNumbers[i] = fibNumbers[i - 1] + fibNumbers[i - 2]
    }
    return fibNumbers[n]
}

```

Figure 10 Fibonacci Tabulation in Swift

Results:

```
Term 23 | Term 28 | Term 38 | Term 45 | Term 100 | Term 200 | Term 300 | Term 400 | Term 500 | Term 600 | Term 20000 | Term 30000
------------------------------------------------------------------------------------------------------------------------------
0.00121 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00002 s | 0.00077 s | 0.00008 s | 0.00010 s | 0.00013 s | 0.00015 s | 0.00388 s | 0.00570 s
0.00001 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00002 s | 0.00004 s | 0.00007 s | 0.00010 s | 0.00012 s | 0.00014 s | 0.00367 s | 0.00538 s
0.00001 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00002 s | 0.00004 s | 0.00007 s | 0.00009 s | 0.00012 s | 0.00014 s | 0.00356 s | 0.00523 s
Program ended with exit code: 0
```

Figure 11 Fibonacci Tabulation results

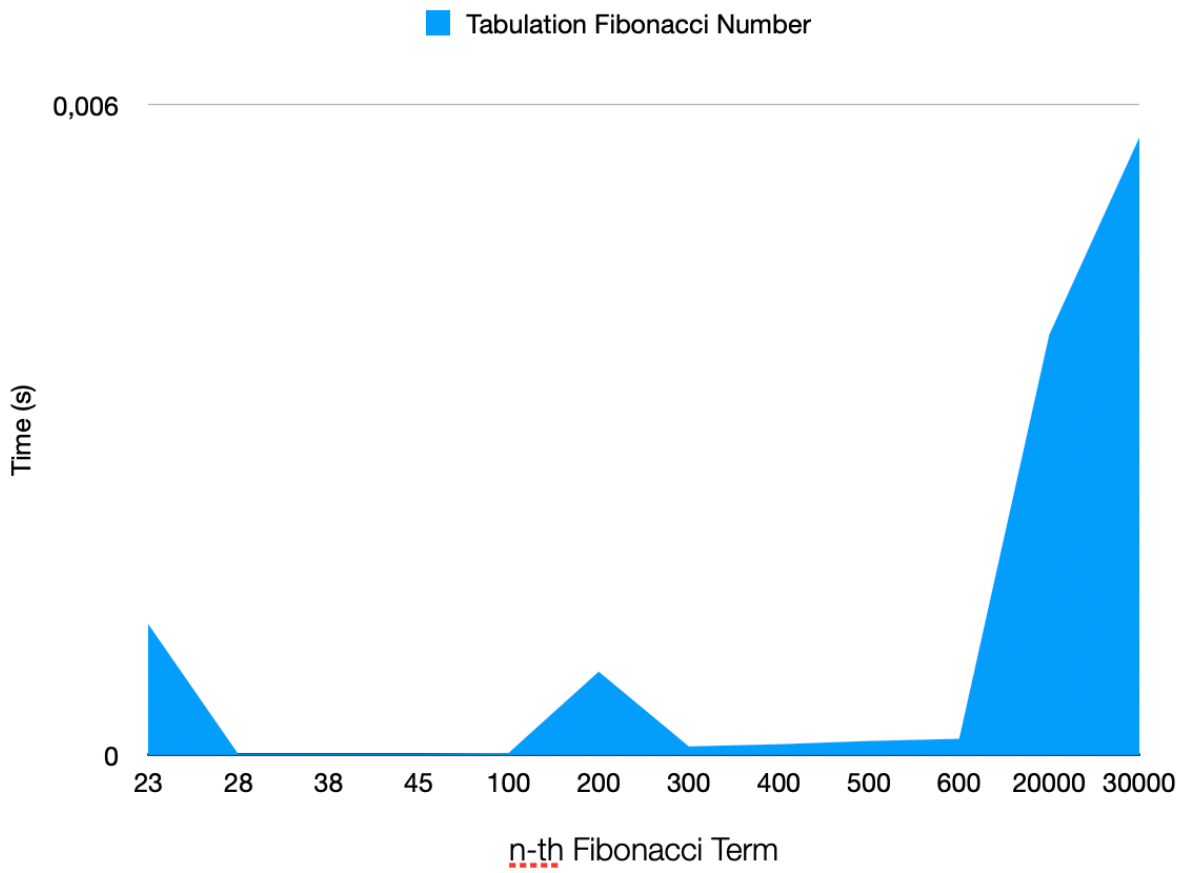

Figure 12 Graph of Fibonacci Tabulation Method

As you can see, this method is slower than Memoization method.

Space Optimized Method:

The space-optimized method for computing Fibonacci numbers aims to reduce the memory footprint required to store intermediate results while still achieving the same time complexity as the tabulation method. This approach only stores the last two Fibonacci numbers calculated rather than maintaining an entire table of results. By updating and reusing only the necessary variables, this method significantly reduces memory consumption. The key idea is to iteratively calculate Fibonacci numbers while only storing the two most recent values, discarding

the rest as they become unnecessary. This results in a space complexity of O(1), making it highly efficient in terms of memory usage.

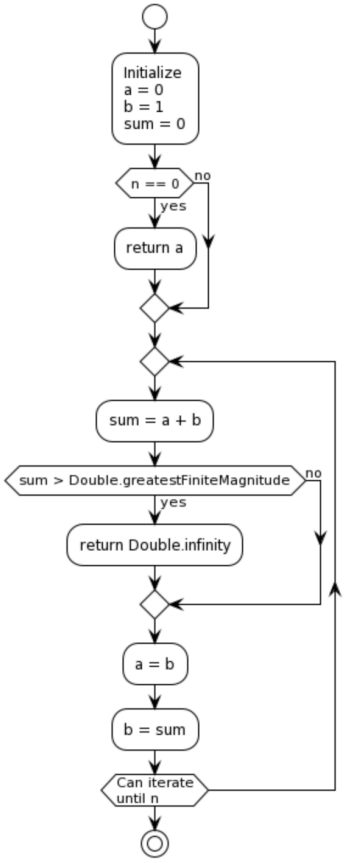

Figure 13 Fibonacci Space Optimization

Algorithm Description:

Here's a description of the space-optimized method:

```
function find_nth_fibonacci(n):
    if n <= 1:
        return n
    prev = 0
    curr = 1
    for i from 2 to n:
        next = prev + curr
        prev = curr
        curr = next
    return curr
```

In this pseudocode:

- Find_nth_fibonacci is the main function that iteratively computes Fibonacci numbers using only two variables (prev and curr) to store the last two Fibonacci numbers.
- The loop calculates the next Fibonacci number (next) by summing prev and curr, then updates prev and curr accordingly.
- Finally, the nth Fibonacci number (curr) is returned. This approach achieves a space complexity of O(1) while maintaining a time complexity of O(n).

Implementation:

```swift
import Foundation

func fibonacciSpaceOptimized(n: Int) -> Double {
    var a: Double = 0, b: Double = 1, sum: Double = 0
    if n == 0 { return a }
    for _ in 2...n {
        sum = a + b
        if sum > Double.greatestFiniteMagnitude {
            return Double.infinity
        }
        a = b
        b = sum
    }
    return b
}
```

Figure 14 Fibonacci Space Optimization in Swift

Results:

```
Term 23 | Term 28 | Term 38 | Term 45 | Term 100 | Term 200 | Term 300 | Term 400 | Term 500 | Term 600 | Term 20000 | Term 30000
-------------------------------------------------------------------------------------------------------------------------------
0.00001 s | 0.00000 s | 0.00001 s | 0.00001 s | 0.00002 s | 0.00003 s | 0.00005 s | 0.00007 s | 0.00008 s | 0.00010 s | 0.00025 s | 0.00025 s
0.00000 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00002 s | 0.00003 s | 0.00005 s | 0.00007 s | 0.00008 s | 0.00010 s | 0.00025 s | 0.00025 s
0.00000 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00002 s | 0.00003 s | 0.00005 s | 0.00007 s | 0.00008 s | 0.00011 s | 0.00025 s | 0.00025 s
Program ended with exit code: 0
```

Figure 15 Fibonacci Space Optimization Output

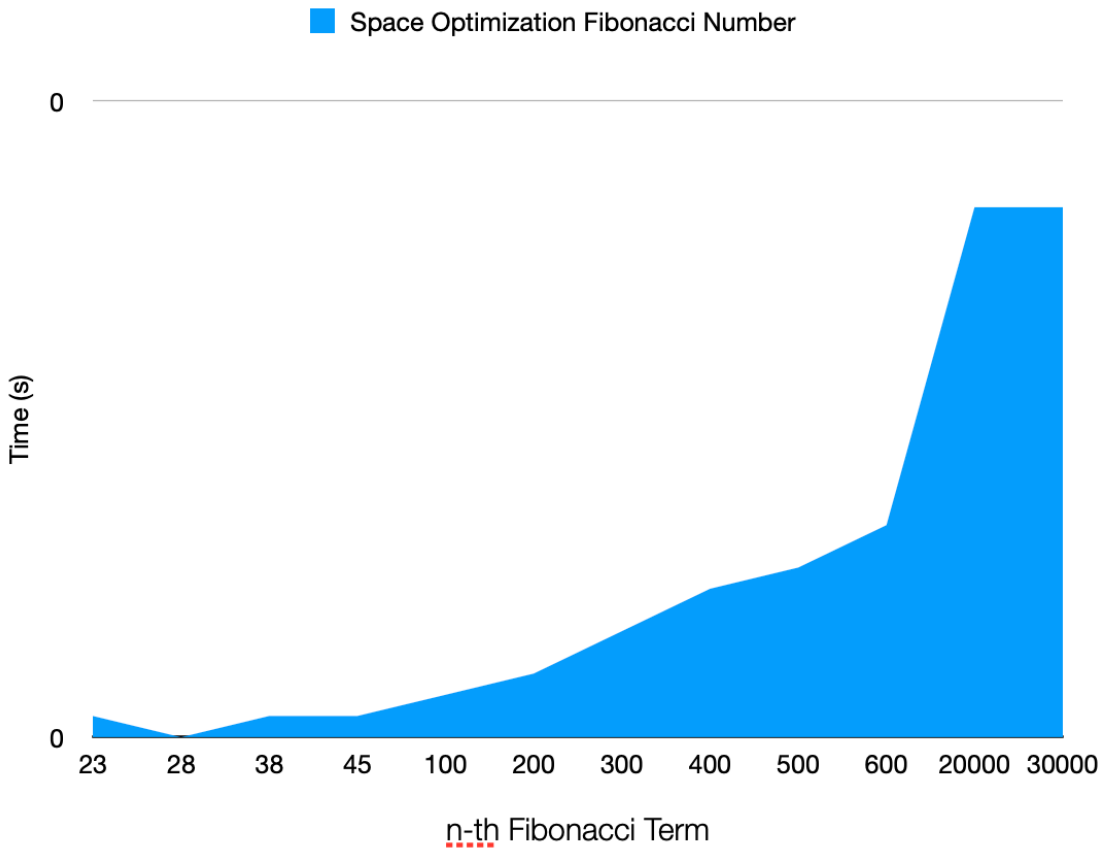

Figure 16 Graph of Space Optimization Function

Matrix Exponentiation Method:

The exponentiation method for finding Fibonacci numbers utilizes the matrix exponentiation technique to efficiently compute Fibonacci numbers in logarithmic time complexity. By representing Fibonacci numbers as matrix exponentiation problems, this method leverages the properties of matrices to compute Fibonacci numbers using repeated squaring. It exploits the fact that the Fibonacci sequence can be represented as a linear recurrence relation and utilizes exponentiation by squaring to efficiently compute the nth Fibonacci number. This approach offers a logarithmic time complexity, making it particularly efficient for large values of n.

Algorithm Description:

Here's the pseudocode for the exponentiation method:

```
function fibonacci_exponentiation(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    base_matrix = [[1, 1], [1, 0]]  // Initialize the base matrix
```

```
    result_matrix = matrix_exponentiation(base_matrix, n-1)  //
Calculate (n-1)th power of the base matrix
    return result_matrix[0][0]

function matrix_exponentiation(matrix, exponent):
    if exponent == 1:
        return matrix
    if exponent % 2 == 0:
        half_power_matrix = matrix_exponentiation(matrix, exponent/2)
        return multiply_matrices(half_power_matrix, half_power_matrix)
    else:
        half_power_matrix = matrix_exponentiation(matrix,
(exponent-1)/2)
        squared_matrix = multiply_matrices(half_power_matrix,
half_power_matrix)
        return multiply_matrices(matrix, squared_matrix)

function multiply_matrices(matrix1, matrix2):
    result_matrix = new matrix with appropriate dimensions
    // Perform matrix multiplication
    return result_matrix
```

In this pseudocode:

- fibonacci_exponentiation is the main function that calculates the nth Fibonacci number using exponentiation.
- matrix_exponentiation calculates the exponentiation of a matrix using recursion and exponentiation by squaring.
- multiply_matrices multiplies two matrices to compute their product.

```swift

import Foundation

func multiply(_ a: [[Double]], _ b: [[Double]]) -> [[Double]] {
    let x = a[0][0] * b[0][0] + a[0][1] * b[1][0]
    let y = a[0][0] * b[0][1] + a[0][1] * b[1][1]
    let z = a[1][0] * b[0][0] + a[1][1] * b[1][0]
    let w = a[1][0] * b[0][1] + a[1][1] * b[1][1]
    return [[x, y], [z, w]]
}

func power(_ matrix: [[Double]], _ n: Int) -> [[Double]] {
    if n == 1 {
        return matrix
    } else if n % 2 == 0 {
        let halfPower = power(matrix, n / 2)
        return multiply(halfPower, halfPower)
    } else {
        return multiply(matrix, power(matrix, n - 1))
    }
}

func fibonacciMatrix(n: Int) -> Double {
    if n == 0 { return 0.0 }
    let matrix = [[1.0, 1.0], [1.0, 0.0]]
    let result = power(matrix, n - 1)
    return result[0][0]
}
```

Fig 17 Matrix Exponentiation Method in Swift

Results:

```
Term 23 | Term 28 | Term 38 | Term 45 | Term 100 | Term 200 | Term 300 | Term 400 | Term 500 | Term 600 | Term 20000 | Term 30000
-------------------------------------------------------------------------------------------------------------------------------
0.00002 s | 0.00000 s | 0.00001 s | 0.00000 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00001 s
0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00001 s
0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00001 s | 0.00001 s
Program ended with exit code: 0
```

Fig 18 Matrix Exponentiation Method Results after first Input

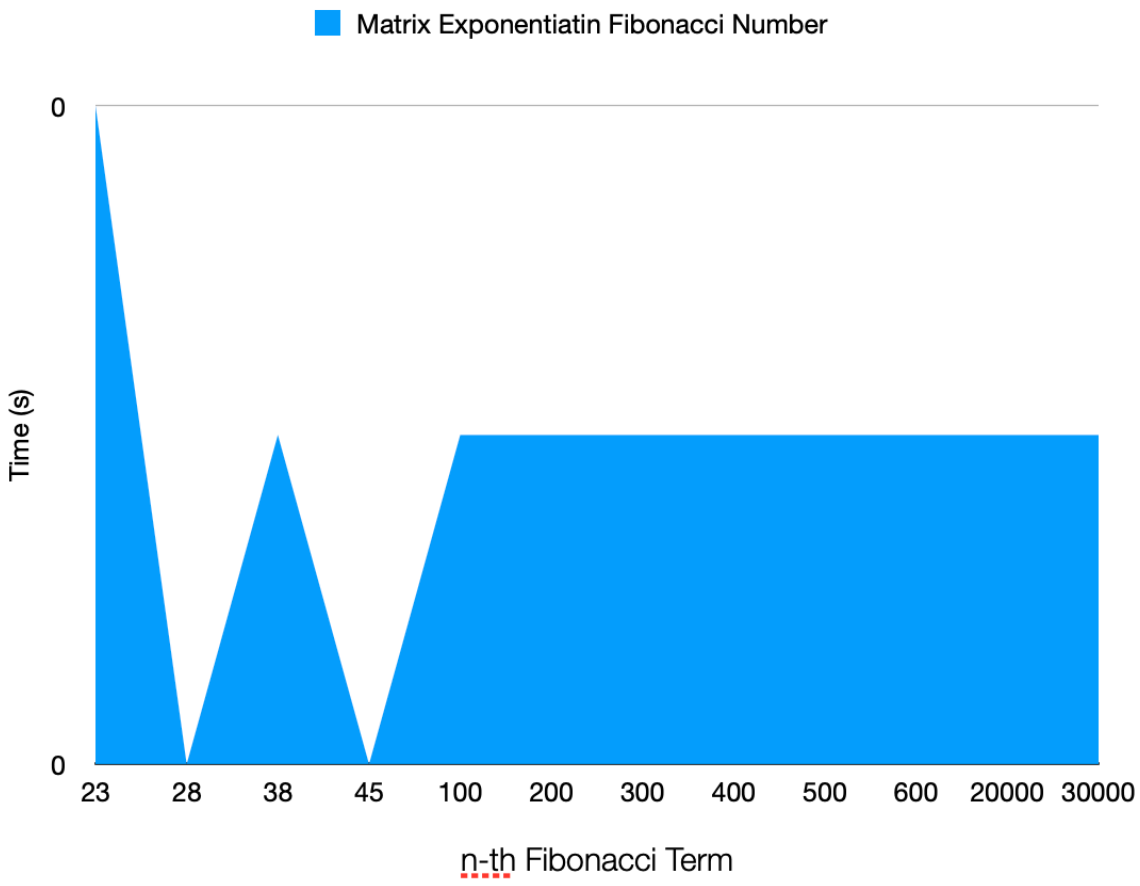

Fig 19 Matrix Exponentiation Method Results in a Graph

So far, the most efficient method. Stopped due to the fact that length of the number being to high.

Golden Ratio Method:

The golden ratio method for finding Fibonacci numbers exploits the mathematical relationship between consecutive Fibonacci numbers and the golden ratio, φ (approximately 1.6180339887). This method leverages the fact that the ratio of two consecutive Fibonacci numbers approaches the golden ratio as n increases. By using the formula $F(n) = \varphi^n/\sqrt{5}$, where φ is the golden ratio and n is the index of the

Fibonacci number, it is possible to directly compute Fibonacci numbers without resorting to iterative or recursive methods. This approach offers a simple and efficient way to calculate Fibonacci numbers, particularly for large values of n, with a time complexity of O(1).

Algorithm Description:
Here's the pseudocode for the golden ratio method:

```
function fibonacci_golden_ratio(n):
    phi = (1 + sqrt(5)) / 2  // Calculate the golden ratio
    return round((phi ** n) / sqrt(5))  // Compute the nth Fibonacci
number using the golden ratio formula
```

Implementation:

```swift
import Foundation

func fibonacciGoldenRatio(n: Int) -> Double {
    let phi = (1 + sqrt(5)) / 2
    return Double(round(pow(phi, Double(n)) / sqrt(5)))
}
```

Fig 20 Golden Ratio in Swift

Results:

```
Term 23 | Term 28 | Term 38 | Term 45 | Term 100 | Term 200 | Term 300 | Term 500 | Term 600 | Term 20000 | Term 100000 | Term 100000000
--------------------------------------------------------------------------------------------------------------------------------------------
0.00001 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s
0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s
0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s | 0.00000 s
Program ended with exit code: 0
```

Fig 21 Golden Ratio in Swift after first input

Due to the fact that this method calculates the nth term of the Fibonacci using a formula, doesn't matter which one you want to find, it will be easy for the computer, because it doesn't have to find the rest terms.

CONCLUSION

In this laboratory work, we embarked on a comprehensive exploration of various computational methods to determine the \(n\)th term of the Fibonacci sequence, a series intrinsically linked to both natural phenomena and mathematical theory. Our investigation spanned six distinct approaches: recursive, memoization, tabulation, space optimization, matrix exponentiation, and the golden ratio method. Each method provided unique insights into the trade-offs between computational efficiency, memory usage, and algorithm complexity.

The recursive method, while straightforward and intuitive, demonstrated significant limitations in terms of computational efficiency due to its exponential time complexity. This approach, however, served as a foundational comparison point for the other, more sophisticated methods.

Memoization and tabulation, both dynamic programming strategies, offered considerable improvements in execution time by avoiding redundant computations. Memoization, with its top-down approach, and tabulation, through a bottom-up technique, both significantly reduced the computational complexity from exponential to polynomial time. However, they differed in memory usage and implementation complexity, with memoization being more intuitive but potentially more memory-intensive due to recursive stack calls, and tabulation being more space-efficient but requiring a more iterative thought process.

The space-optimized method further refined the dynamic programming approach by reducing the memory requirement to a minimal, constant amount, thus addressing one of the primary limitations of memoization and tabulation without sacrificing computational speed.

Matrix exponentiation emerged as a powerful technique, offering a quantum leap in efficiency by reducing the problem to logarithmic time complexity. This method, based on linear algebra principles, highlighted the deep mathematical underpinnings of the Fibonacci sequence and demonstrated the potential for significant optimization when leveraging mathematical properties.

Lastly, the golden ratio method, a formula derived from Binet's formula, provided a direct, non-iterative calculation of the \(n\)th Fibonacci number. This approach, while impressively concise and elegant, introduced challenges related to floating-point arithmetic and precision, especially for large values of \(n\).

In conclusion, this laboratory work not only illuminated the Fibonacci sequence's fascinating characteristics but also underscored the importance of selecting appropriate algorithms based on the specific requirements and constraints of a computational problem. From the simplicity of recursion to the efficiency of matrix exponentiation and the elegance of the golden ratio, each method contributes uniquely to our understanding and computational toolkit.

As we continue to delve into the vast expanse of algorithmic strategies, the insights gained from this comparative study will undoubtedly enrich our approach to solving complex computational problems, demonstrating the power and beauty of algorithmic diversity and optimization.