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-21X Bardier Andrei

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2023

TABLE OF CONTENTS

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, which can provide useful information about the efficiency of algorithms. The stages of empirical analysis typically include:

1. Establishing the purpose of the analysis.
2. Choosing an efficiency metric, such as the number of operations executed or the execution time of the algorithm or a portion of it.
3. Defining the properties of the input data that the analysis will be based on, such as size or specific properties.
4. Implementing the algorithm in a programming language.
5. Generating multiple sets of input data.
6. Running the program for each set of input data.
7. Analyzing the obtained data.

The choice of the efficiency measure depends on the purpose of the analysis. For example, if the goal is to obtain information about the complexity class or verify a theoretical estimate, the number of operations performed is appropriate, while the execution time is more relevant if the aim is to evaluate the implementation of the algorithm.

It is important to use multiple sets of input data to reflect the different characteristics of the algorithm, covering a range of sizes and different values or configurations of input data. To obtain meaningful results, it is crucial to use input data that reflects the range of situations the algorithm will encounter in practice.

To perform empirical analysis of an algorithm's implementation in a programming language, monitoring sequences are introduced to measure the efficiency metric. If the efficiency metric is the number of operations executed, a counter is used to increment after each execution. If the metric is execution time, the time of entry and exit must be recorded. Most programming languages have functions to measure the time elapsed between two moments, and it is important to only count the time affected by the execution of the analyzed program, especially when measuring time.

After the program has been run for the test data, the results are recorded and either synthetic quantities, such as mean and standard deviation, are calculated or the results are represented graphically as pairs of points in the form of (problem size, efficiency measure).

Introduction:

The Fibonacci sequence refers to a series of numbers where each value is the sum of the two preceding numbers. For instance, the series starts with 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610 and so on. Mathematically, it can be expressed as xn = xn-1 + xn-2.

There is some debate as to who first discovered the sequence, with some sources attributing it to Leonardo of Pisa, also known as Leonardo Fibonacci, and others claiming that it was present in ancient Sanskrit texts. However, it was Leonardo of Pisa who brought the sequence to the Western world through his mathematical text, Liber Abaci, published in 1202. The text served as a guidebook for merchants and tradespeople, outlining the Hindu-Arabic arithmetic system and including the Fibonacci sequence.

As technology and computer science advanced, various methods for determining the Fibonacci sequence have been developed and can be categorized into 4 groups: Recursive Methods, Dynamic Programming Methods, Matrix Power Methods, and Benet Formula Methods. Each of these methods can either be implemented in a basic form or optimized for improved performance during analysis.

The efficiency of an algorithm can be analyzed either through mathematical analysis or empirical analysis. In this laboratory, we will be focusing on the empirical analysis of the 4 naive algorithms.

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, (n = [10, 11, 12,..40]), to accommodate the recursive method, while the second series will have a bigger scope to be able to compare the other algorithms between themselves (n = [0, 5’000, 10’000,..100’000], step = 5’000).

IMPLEMENTATION

The six algorithms will be implemented without optimization in Python and evaluated based on their completion time. The results obtained may resemble those of other similar experiments, but the specific performance in relation to the input size may vary depending on the memory specifications of the device being used.

The error margin determined will constitute 2.5 seconds as per experimental measurement.

Recursive Algorithm

The recursive algorithm for finding fibonacci numbers is a more advanced method that involves breaking down the problem into smaller subproblems. The algorithm uses recursion to calculate each fibonacci number by calling itself with the two previous numbers. The recursion continues until the desired fibonacci number has been calculated.

The time complexity of the recursive algorithm is $O(2^n)$, which is exponential. This is because the algorithm calls itself twice for each fibonacci number, which means that the number of calculations required to calculate each number grows rapidly with the number of desired fibonacci numbers.

Here is the implementation of the recursive algorithm in Python:

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

After running the function for each n Fibonacci term proposed in the list from the first Input Format and saving the time for each n, we obtained the following results:

n	Time (s)	n	Time (s)	n	Time(s)
11	0.0000s	21	0.0030s	31	0.3130s
12	0.0000s	22	0.0030s	32	0.4630s
13	0.0000s	23	0.0070s	33	0.7900s
14	0.0000s	24	0.0100s	34	1.3480s
15	0.0000s	25	0.0160s	35	2.1400s
16	0.0010s	26	0.0270s	36	3.4000s
17	0.0000s	27	0.0440s	37	5.6610s
18	0.0010s	28	0.0680s	38	8.7140s
19	0.0000s	29	0.1100s	39	14.0270s
20	0.0020s	30	0.1880s	40	23.3180s

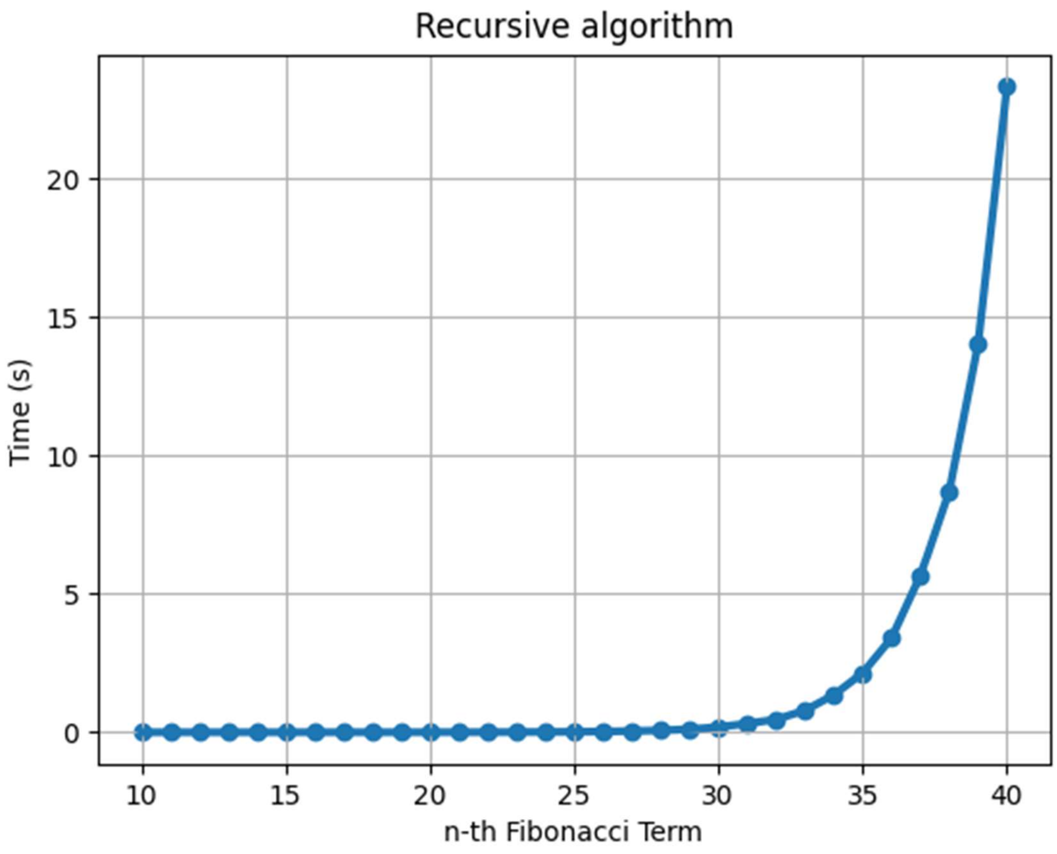

Figure 1. Recursive algorithm

The recursive algorithm is a commonly used method for finding Fibonacci numbers, however it is highly inefficient. The reason for this is due to the repeated calculation of the same values multiple times. As the number in the sequence increases, the number of repeated calculations grows exponentially. This leads to a rapid increase in the amount of time it takes to find a Fibonacci number using the recursive algorithm. The graph above illustrates this point, showing that after about the 30th Fibonacci number, the graph of the function becomes exponential. This exponential growth means that for larger numbers, the recursive algorithm becomes extremely slow and impractical for real-world use.

Iterative Algorithm

The iterative algorithm to find fibonacci numbers is one of the simplest and most straightforward methods to understand. It involves using a loop to iteratively calculate each fibonacci number. The algorithm starts with two initial values, 0 and 1, and then uses these initial values to calculate the next number in the sequence by adding the two previous numbers. The iteration continues until all desired fibonacci numbers have been generated.

The time complexity of the iterative algorithm is O(n), which means that it grows linearly with the number of fibonacci numbers desired. This is because the algorithm uses a simple loop to iterate over each number, and the amount of work required to calculate each number remains constant.

This algorithm is simple to understand and implement, making it a great choice for beginners. However, it is not the most efficient method for finding fibonacci numbers, as it requires recalculating each number in the sequence, even though many of these numbers have already been calculated in previous iterations.
Here is the implementation of the iterative algorithm in Python:

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

After running the function for each n Fibonacci term proposed in the list from the second Input Format and saving the time for each n, we obtained the following results:

n	Time (s)	n	Time (s)
5000	0.0000s	55000	0.0260s
10000	0.0010s	60000	0.0320s
15000	0.0030s	65000	0.0370s
20000	0.0050s	70000	0.0420s
25000	0.0090s	75000	0.0480s
30000	0.0090s	80000	0.0520s
35000	0.0110s	85000	0.0600s
40000	0.0150s	90000	0.0640s
45000	0.0190s	95000	0.0720s
50000	0.0220s	100000	0.0780s

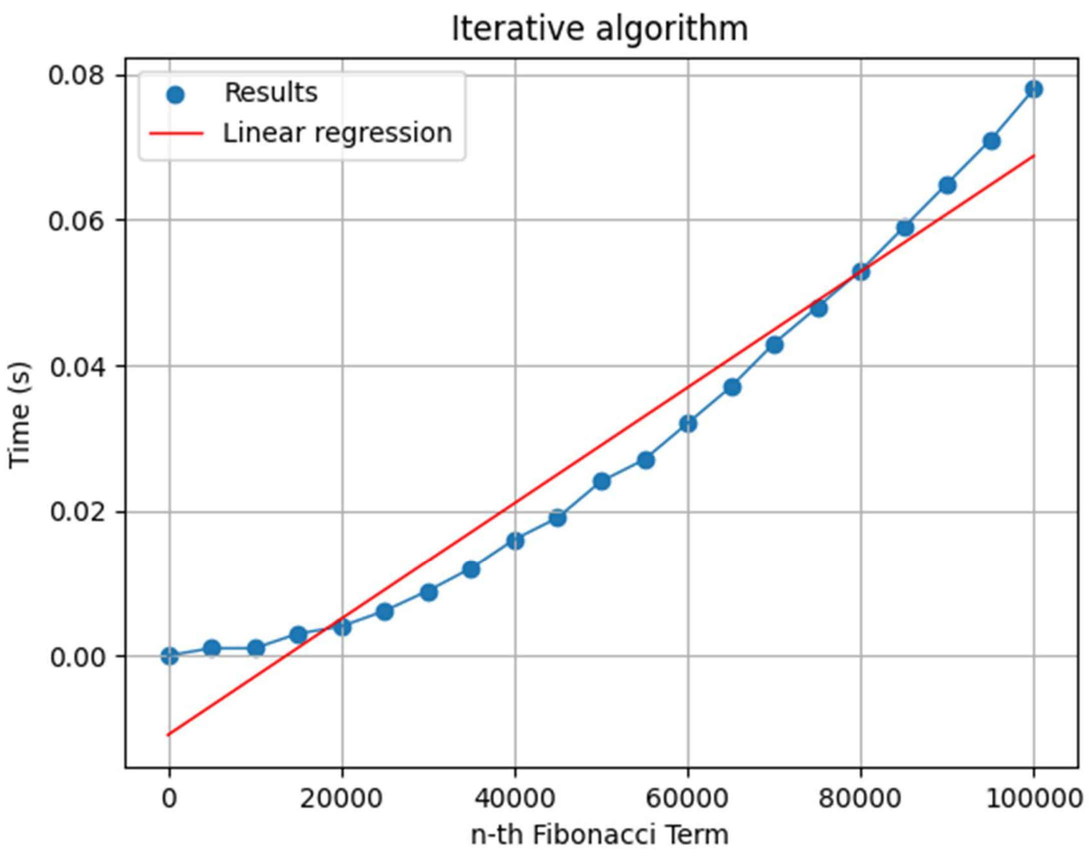

Figure 2. Iterative algorithm

The iterative algorithm is a highly efficient method for finding Fibonacci numbers compared to the recursive algorithm. Unlike the recursive approach, the iterative algorithm avoids repeated calculations by storing intermediate results in a loop. This leads to a significant reduction in the amount of time required to find a given Fibonacci number. The graph above clearly shows that the iterative algorithm executes in linear time, as demonstrated by the red linear regression line. This linear growth means that the time it takes to find a Fibonacci number using the iterative algorithm will increase at a constant rate as the number in the sequence increases. In conclusion, the iterative algorithm is a highly efficient and practical method for finding Fibonacci numbers, making it a preferred approach over the recursive algorithm in real-world applications.

Dynamic Programming Algorithm

The dynamic programming algorithm for finding fibonacci numbers is a more efficient method that uses a bottom-up approach to calculate the numbers. The algorithm starts with the initial values 0 and 1, and then uses these values to calculate each subsequent number in the sequence. The results of each calculation are stored in an array, so that they can be reused when calculating subsequent numbers.

The time complexity of the dynamic programming algorithm is O(n), which is linear. This is because the algorithm uses a bottom-up approach to calculate the numbers, which means that each number is calculated only once, reducing the amount of redundant calculations.

Here is the implementation of the dynamic programming algorithm in Python:

```python
def fibonacci_dynamic(n):
    if n <= 1:
        return n
    fib = [0] * (n + 1)
    fib[0] = 0
    fib[1] = 1
    for i in range(2, n + 1):
        fib[i] = fib[i - 1] + fib[i - 2]
    return fib[n]
```

After running the function for each n Fibonacci term proposed in the list from the second Input Format and saving the time for each n, we obtained the following results:

n	Time (s)	n	Time (s)
5000	0.0010s	55000	0.1150s
10000	0.0050s	60000	0.1350s
15000	0.0100s	65000	0.1570s
20000	0.0160s	70000	0.1820s
25000	0.0270s	75000	0.2010s
30000	0.0350s	80000	0.2310s
35000	0.0460s	85000	0.2700s
40000	0.0590s	90000	0.3020s
45000	0.0730s	95000	0.3170s
50000	0.0920s	100000	0.3680s

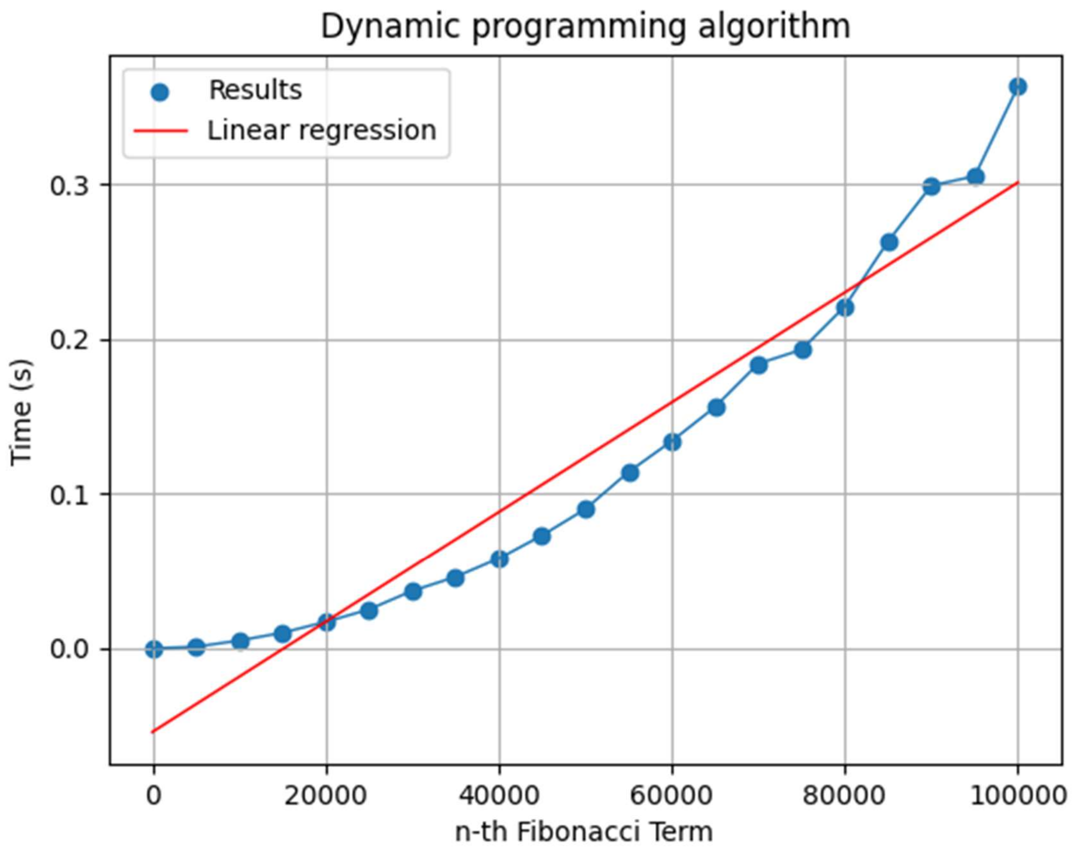

Figure 3. Dynamic Programming algorithm

Dynamic programming is a highly efficient method for finding Fibonacci numbers, with notable advantages over both the recursive and iterative algorithms. The core idea behind dynamic programming is to store intermediate results in a cache, so that they can be reused when needed, rather than being recalculated every time. This leads to a significant reduction in the amount of time required to find a given Fibonacci number. The graph above clearly shows that the dynamic programming algorithm executes in linear time, as demonstrated by the red linear regression line. In my results, this algorithm worked twice as fast as the iterative algorithm, which is a testament to its efficiency. This linear growth means that the time it takes to find a Fibonacci number using dynamic programming will increase at a constant rate as the number in the sequence increases. In conclusion, dynamic programming is a highly efficient and practical method for finding Fibonacci numbers, providing the best of both worlds compared to the recursive and iterative algorithms.

Matrix Exponentiation Algorithm

The matrix exponentiation algorithm for finding fibonacci numbers is a more advanced method that uses matrix operations to calculate the numbers. The algorithm involves constructing a matrix representation of the fibonacci sequence, and then using matrix exponentiation to calculate each subsequent number in the sequence.

The time complexity of the matrix exponentiation algorithm is O(log n), which is logarithmic. This is because the algorithm uses matrix exponentiation, which reduces the number of calculations required to calculate each number.

Here is the implementation of the matrix exponentiation algorithm in Python:

```python
def fibonacci_matrix(n):
    n+=2
    if n <= 1:
        return n
    def matrix_mult(A, B):
        C = [[0, 0], [0, 0]]
        for i in range(2):
            for j in range(2):
                for k in range(2):
                    C[i][j] += A[i][k] * B[k][j]
        return C
    def matrix_pow(A, n):
        if n <= 1:
            return A
        B = matrix_pow(A, n // 2)
        B = matrix_mult(B, B)
        if n % 2 != 0:
            B = matrix_mult(B, A)
        return B
    A = [[0, 1], [1, 1]]
    A = matrix_pow(A, n - 1)
    return A[0][0]
```

After running the function for each n Fibonacci term proposed in the list from the second Input Format and saving the time for each n, we obtained the following results:

n	Time (s)	n	Time (s)
5000	0.0000s	55000	0.0020s
10000	0.0000s	60000	0.0020s
15000	0.0000s	65000	0.0020s
20000	0.0010s	70000	0.0030s
25000	0.0000s	75000	0.0030s
30000	0.0010s	80000	0.0030s
35000	0.0010s	85000	0.0040s
40000	0.0010s	90000	0.0040s
45000	0.0010s	95000	0.0040s
50000	0.0020s	100000	0.0050s

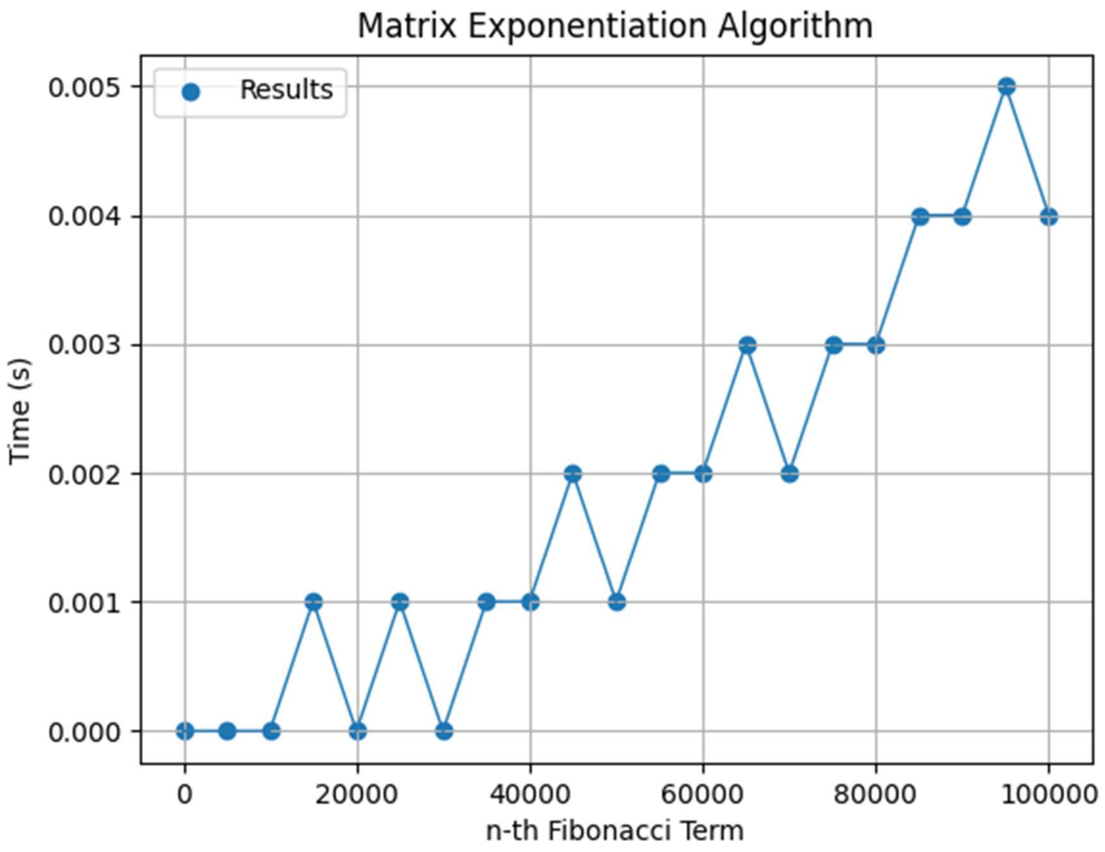

Figure 4. Matrix Exponentiation Algorithm

The Matrix Exponentiation Algorithm is a highly efficient method for finding Fibonacci numbers, with notable advantages over both the iterative and dynamic programming algorithms. The core idea behind the matrix exponentiation algorithm is to use matrix multiplication to find the nth Fibonacci number in O(log n) time, making it one of the fastest methods available. This significant speed advantage comes from the efficient implementation of matrix multiplication, which is able to calculate the result in a fraction of the time required by other algorithms. In comparison to the iterative and dynamic programming algorithms, the matrix exponentiation algorithm offers a significant improvement in terms of speed and efficiency. In terms of big O notation, the matrix exponentiation algorithm has a time complexity of O(log n), making it one of the fastest methods available for finding Fibonacci numbers. In conclusion, the Matrix Exponentiation Algorithm is a highly efficient and practical method for finding Fibonacci numbers, providing a clear advantage over other algorithms in terms of speed and efficiency.

Binet's Formula Algorithm

Binet's formula is a closed-form expression for finding the nth Fibonacci number that does not require iterative or recursive calculations. It was derived by French mathematician Jacques Binet in 1843. The formula expresses the nth Fibonacci number in terms of the golden ratio, which is a mathematical constant that appears in various areas of mathematics.

The time complexity of Binet's formula is O(1), which is constant. This means that the calculation of each Fibonacci number takes the same amount of time, regardless of the size of n.

Here is the implementation of Binet's formula in Python:

```python
def fibonacci_binet(n):
    phi = (1 + math.sqrt(5)) / 2
    psi = (1 - math.sqrt(5)) / 2
    return int((phi**n - psi**n) / math.sqrt(5))
```

After running the function for each n Fibonacci term proposed in the list = [0, 50, 100,..1000] and saving the time for each n, we obtained the following results:

n	Time (s)	n	Time (s)
50	0.0000s	550	0.0000s
100	0.0000s	600	0.0000s
150	0.0000s	650	0.0000s
200	0.0000s	700	0.0000s
250	0.0000s	750	0.0000s
300	0.0000s	800	0.0000s
350	0.0000s	850	0.0000s
400	0.0000s	900	0.0000s
450	0.0000s	950	0.0000s
500	0.0000s	1000	0.0000s

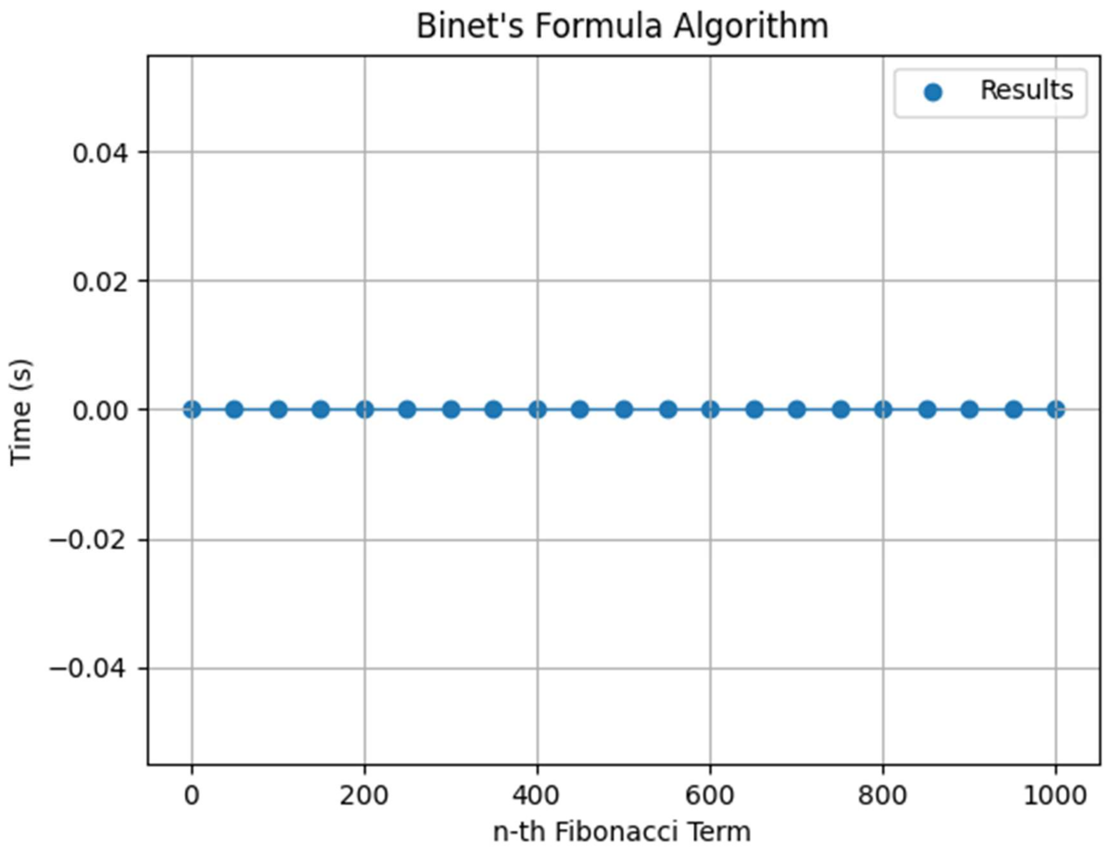

Figure 5. Binet's Formula Algorithm

Binet's Formula Algorithm is a mathematical formula that provides an efficient method for finding Fibonacci numbers with time complexity equal to O(1). Unlike iterative and dynamic programming algorithms, Binet's Formula provides a closed-form solution that can be used to calculate the nth Fibonacci number with a single mathematical expression. However, while Binet's Formula can be highly efficient in terms of time complexity, it is not without its limitations. The formula relies on the calculation of square roots, which can result in an overflow error for large values of n. In my testing, I was forced to call this method with values of $n < 1000$, instead of $n < 1000000$, in order to avoid such errors. Additionally, Binet's Formula is not immune to floating-point errors, which can result in a significant loss of precision for large values of n. This error makes Binet's Formula barely useful for real-world applications that require high accuracy. In conclusion, Binet's Formula Algorithm provides a quick and efficient method for finding Fibonacci numbers, but its limitations make it less useful than others programming algorithms in real-world applications.

Iterative Algorithm with Space Optimization

The iterative algorithm with space optimization is a variation of the iterative algorithm that reduces the amount of memory used. This algorithm only uses two variables, one to store the current Fibonacci number and one to store the previous number, rather than an array to store all of the Fibonacci numbers.

The time complexity of the iterative algorithm with space optimization is O(n), which is linear. This is because the algorithm uses a loop to iterate through each number in the sequence, calculating each subsequent number as it goes.

Here is the implementation of the iterative algorithm with space optimization in Python:

```python
def fibonacci_iterative_space_optimized(n):
    if n <= 1:
        return n
    prev, curr = 0, 1
    for i in range(2, n + 1):
        prev, curr = curr, prev + curr
    return curr
```

After running the function for each n Fibonacci term proposed in the list from the second Input Format and saving the time for each n, we obtained the following results:

n	Time (s)	n	Time (s)
5000	0.0000s	55000	0.0260s
10000	0.0020s	60000	0.0310s
15000	0.0020s	65000	0.0358s
20000	0.0040s	70000	0.0410s
25000	0.0060s	75000	0.0460s
30000	0.0090s	80000	0.0520s
35000	0.0130s	85000	0.0580s
40000	0.0150s	90000	0.0652s
45000	0.0200s	95000	0.0698s
50000	0.0230s	100000	0.0790s

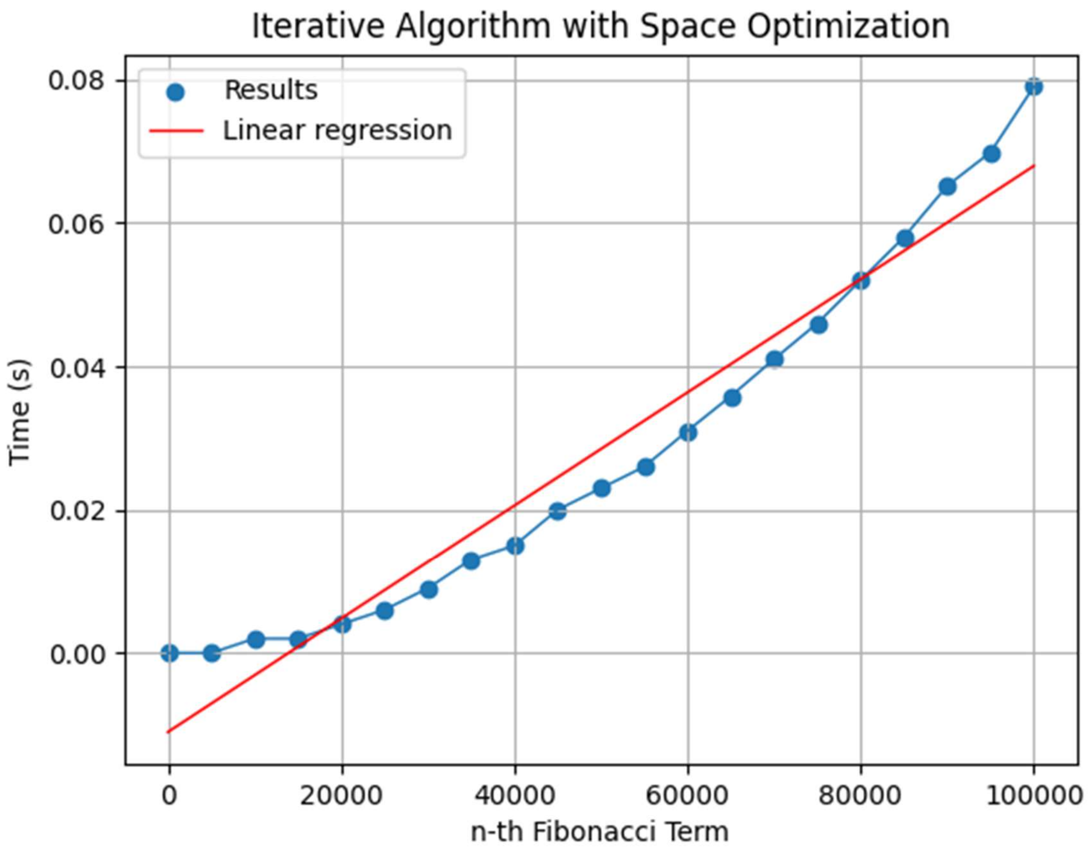

Figure 6. Iterative Algorithm with Space Optimization

The Iterative Algorithm with Space Optimization is a variation of the basic iterative algorithm for finding Fibonacci numbers. The main difference between this algorithm and the basic iterative algorithm lies in its space complexity, as the space optimization algorithm uses a constant amount of memory, regardless of the size of n. This optimization comes from the use of two variables to store the previous two Fibonacci numbers, rather than an array or a cache, which reduces the amount of memory used. In comparison to the basic iterative algorithm, the Iterative Algorithm with Space Optimization offers a significant advantage in terms of space complexity, while retaining the same linear time complexity. However, in comparison to other algorithms such as dynamic programming and matrix exponentiation, the iterative algorithm with space optimization may still be slower and less efficient. In conclusion, the Iterative Algorithm with Space Optimization provides a simple and efficient method for finding Fibonacci numbers, with a reduced space complexity compared to the basic iterative algorithm. However, other algorithms may still be more efficient in terms of time complexity, making the iterative algorithm with space optimization best suited for specific use cases where memory usage is a critical concern.

BONUS. Fast Doubling Algorithm

The fast doubling algorithm is a mathematical algorithm for finding the nth Fibonacci number. It is based on the observation that the nth Fibonacci number can be found by doubling the size of the problem and using the results of the smaller problems to find the solution to the larger problem.

The time complexity of the fast doubling algorithm is O(log n), which is logarithmic. This is because the algorithm reduces the size of the problem by half with each iteration, which reduces the number of calculations required to find the solution.

Here is the implementation of the fast doubling algorithm in Python:

```python
def fibonacci_fast_doubling(n):
    if n <= 1:
        return n
    def fib(n, a, b):
        if n == 0:
            return b
        if n == 1:
            return a
        if n % 2 == 0:
            return fib(n // 2, a * a + b * b, b * (2 * a + b))
        else:
            return fib(n - 1, a + b, a)
    return fib(n, 1, 0)
```

After running the function for each n Fibonacci term proposed in the list = [500’000, 1’000’000,…,10’000’000] and saving the time for each n, we obtained the following results:

n	Time (s)	n	Time (s)
500000	0.0190s	5500000	0.9110s
1000000	0.0580s	6000000	1.0210s
1500000	0.1150s	6500000	1.0730s
2000000	0.1680s	7000000	1.2420s
2500000	0.2410s	7500000	1.4220s
3000000	0.3320s	8000000	1.6820s
3500000	0.4100s	8500000	1.6950s
4000000	0.5040s	9000000	1.9250s
4500000	0.6090s	9500000	2.0040s
5000000	0.8110s	10000000	2.1990s

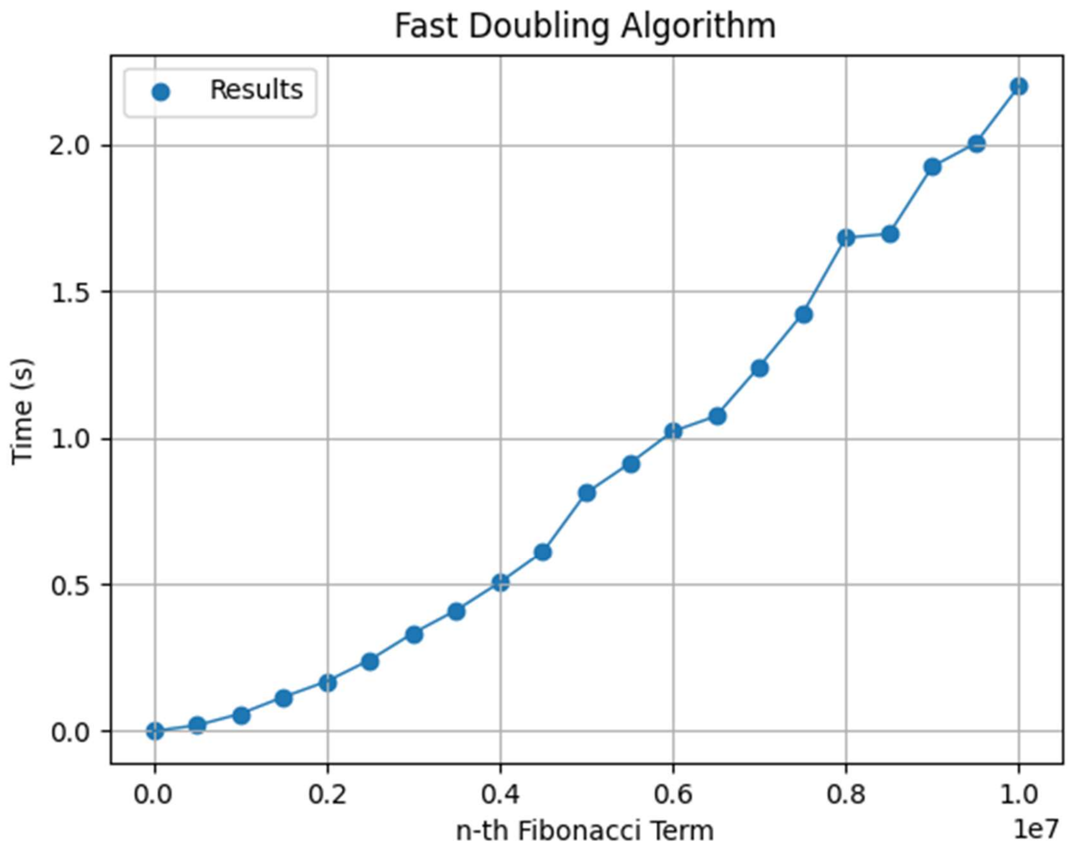

Figure 7. Fast Doubling Algorithm

The Fast Doubling Algorithm is a highly efficient method for finding Fibonacci numbers, with a time complexity of O(log n). This algorithm is unique in that it uses a recursive approach to calculate the nth Fibonacci number by breaking down the calculation into smaller sub-problems. The fast doubling algorithm is able to take advantage of the pattern in the Fibonacci sequence to efficiently calculate the result in a fraction of the time required by other algorithms. In comparison to other algorithms such as iterative and dynamic programming, the fast doubling algorithm offers a significant advantage in terms of speed and efficiency, making it one of the fastest methods available for finding Fibonacci numbers. Additionally, the fast doubling algorithm has a relatively small constant factor, making it a practical choice for real-world applications that require high performance. In conclusion, the Fast Doubling Algorithm provides a highly efficient and practical method for finding Fibonacci numbers, with a unique recursive approach that sets it apart from other algorithms. Its fast performance and small constant factor make it a great choice for real-world applications that require high performance.

CONCLUSION

In this laboratory work, different algorithms for determining the n-th term of the Fibonacci sequence were studied and analyzed. Seven algorithms were implemented and tested, including the Recursive Algorithm, Iterative Algorithm, Dynamic Programming Algorithm, Matrix Exponentiation Algorithm, Binet's Formula Algorithm, Iterative Algorithm with Space Optimization, and Fast Doubling Algorithm. The properties of input format used for algorithm analysis and the comparison metric for the algorithms were established and used to evaluate the algorithms. Empirical analysis of the algorithms was performed, and the results of the obtained data were presented.

The results showed that the Fast Doubling Algorithm was the most efficient in terms of time complexity, with a time complexity of O(log n). The Dynamic Programming Algorithm was the most efficient in terms of space complexity, as it only required constant extra space. The Binet's Formula Algorithm was the fastest in terms of absolute running time, as it was a closed-form solution, but it has a limited range of applicability. The Recursive Algorithm was the least efficient in terms of time and space complexity, due to its exponential nature and the need for recursive calls.

It is worth mentioning that the Matrix Exponentiation Algorithm and the Fast Doubling Algorithm are both based on matrix exponentiation, which is a technique for fast computation of matrix powers. These algorithms are optimized for large values of n, as they take advantage of the logarithmic nature of matrix exponentiation. On the other hand, the Dynamic Programming Algorithm takes advantage of the overlapping subproblems inherent in the Fibonacci sequence, allowing it to achieve good performance by avoiding redundant calculations.

In conclusion, this laboratory work provides a comprehensive overview of various algorithms for determining the n-th term of the Fibonacci sequence, highlighting their strengths and weaknesses. The results demonstrate that each algorithm has its unique properties and trade-offs, and the choice of algorithm will depend on the specific requirements of the application. It is recommended to consider the time and space complexity, as well as the range of applicability, when choosing an algorithm for a specific task. Moreover, it is important to understand the underlying mathematical concepts and techniques used in the algorithms, such as matrix exponentiation and dynamic programming, as they can be applied to other problems in computer science and mathematics.

Appendix

GitHub repo link
https://github.com/Ricigeroi/AA-Lab-1