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 Lupan Lucian

Verified:

asist. univ. Fiștic Cristofor

Chişinău – 2024

Contents

ALGORITHM ANALYSIS

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

Tasks:

1. Implement at least 3 algorithms for determining the n-th Fibonacci number;
2. Decide properties of input format that will be used for algorithm analysis;
3. Decide the comparison metric for the algorithms;
4. Analyse 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 timeexecution of all or part of the algorithm.
3. The properties of the input data with 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 analysed.

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 to check the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess thebehavior 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 twopreceding 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. TheItalian 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 Sanskrittexts 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 Westernworld.

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 fordetermination 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 6 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, (np.unique(np.linspace(5, n, 17, dtype=int))); n = 40, to accommodate the recursive method, while the second and third algorithms willhave a bigger scope to be able to compare the other algorithms between themselves, but still limited due to recursion limit(np.unique(np.linspace(5, n, 17, dtype=int))); n = 2000. The 4-6th algorithms compute much higher terms - (np.unique(np.linspace(5, n, 17, dtype=int))); n = 10000.

IMPLEMENTATION

All six algorithms are implemented in Python and analyzed empirically based on the time required for their completion, using the matplotlib and timeit libraries (as well as the numpy, sys, decimal and math to optimize/work with higher terms of the Fibonacci sequence).

Recursive Algorithm:

Algorithm Description:

The recursive algorithm is one of the most inefficient algorithms for determining Fibonacci numbers, as it has a time complexity of O(2^n), making numbers >40 taking minutes/hours to compute. This algorithm's weakness is that it never uses the memory for storage, meaning it doesn't store the number that have already been computed, and therefore it must compute them again through recursion. In figure 1, we can notice the problem with this algorithm.

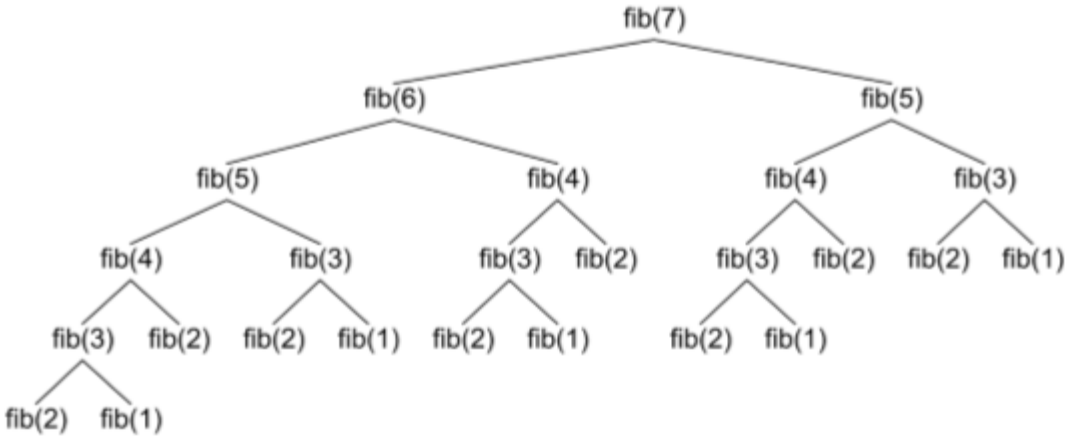

Figure 1. Fibonacci recursion

Implementation:

```
def recursive_fibonacci(nr):
    if nr == 0:
        return 0
    elif nr == 1:
        return 1
    else:
        return recursive_fibonacci(nr - 1) + recursive_fibonacci(nr - 2)
```

Figure 2. Fibonacci recursion code

Results:

After running the algorithm in Python, the time it took to obtain the terms above 40 was very high (taking minutes to compute 1 term), as can be observed in the table below.

N	5	7	9	11	13	15	18	20	22	24	26	29	31	33	35	37	40
Milliseconds	~0	~0	~0	~0	~0	0.998	1	3	10.99	28.86	78	576	984	2663	7206	27520	74868

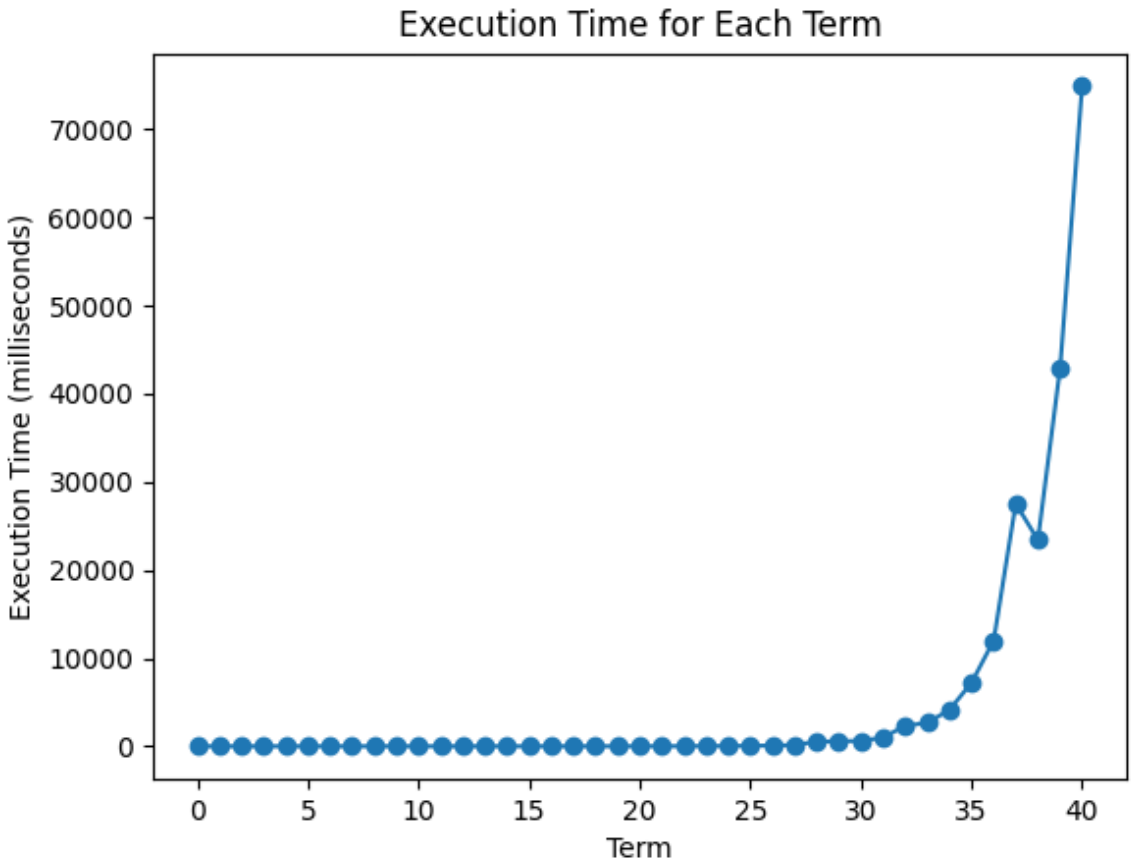

Figure 3. Fibonacci recursion time execution

In Figure 3, we can see the time it took to compute the nth term through the use of a graph. There's a sudden spike at n=40, and each of the terms after that approximately double the time taken of the previous term, due to its time complexity of O(2^n).

Dynamic programming (Recursion) Algorithm:

Algorithm Description:

This method generally functions the same as the recursive method, however we add a "cache", that stores the values of the already computed Fibonacci numbers, thus decreasing the runtime and making finding numbers that are higher than 40 less demanding.

Implementation:

```
def dynamic_fibonacci(nr, cache=None):
    if cache is None:
        cache = {}
    if nr == 0:
        return 0
    elif nr == 1:
        return 1
    else:
        if cache.get(nr - 1) is None:
            cache[nr - 1] = dynamic_fibonacci(nr - 1, cache)
        if cache.get(nr - 2) is None:
            cache[nr - 2] = dynamic_fibonacci(nr - 2, cache)
        return cache[nr - 1] + cache[nr - 2]
```

Figure 4. Dynamic programming code

Results:

After the execution, we can notice a massive decrease in the time required to compute each of the numbers.

N	5	129	254	379	503	628	753	877	1002	1127	1251	1376	1501	1625	1750	1875	2000
Milliseconds	0.03	0.25	2.19	0.74	1.60	1.96	1.69	1.99	2.29	1.81	1.93	2.22	2.59	3.26	3.83	3.02	3.42

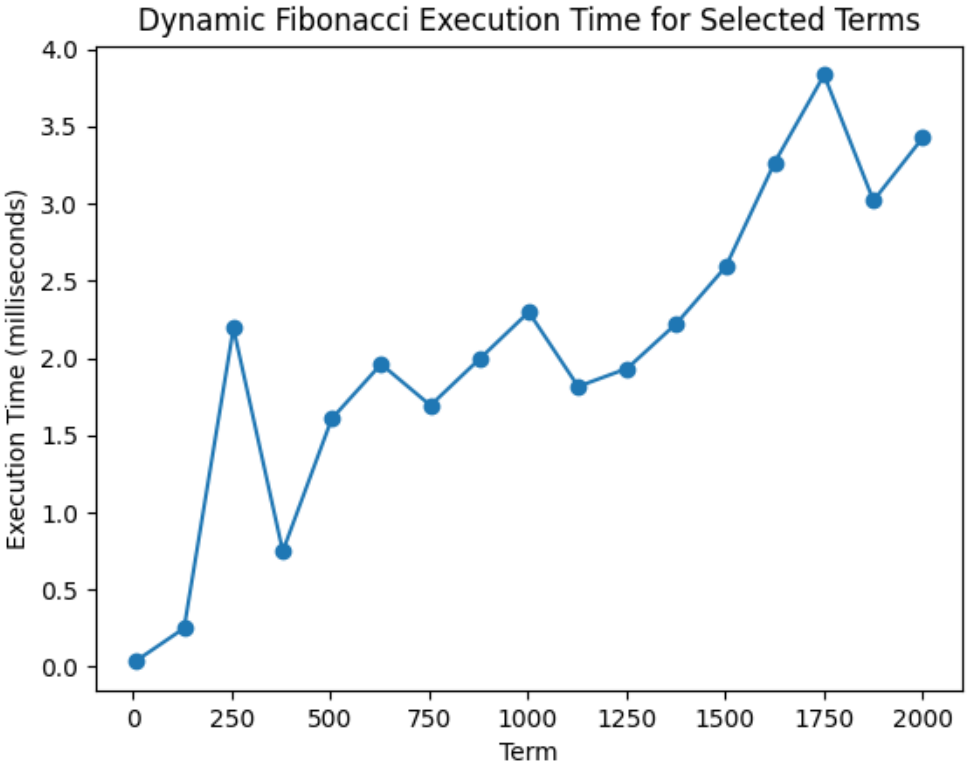

Figure 5. Dynamic programming graph

With the optimized recursive algorithm, we can notice the graph in Figure 5 that higher Fibonacci terms took far less time to compute, as this algorithm only has a time complexity of O(n). This algorithm is still relatively slow compared to the next algorithms, and because it uses recursion, we encounter the max recursion limit error, which makes it so that we have trouble computing terms past 2000, even through the use of system functions which increase recursion limit, as we encounter the stack overflow error.

Iterative Algorithm:

Algorithm Description:

The Iterative algorithm is alike the dynamic programming approach, storing the values of the already computed terms; however, it doesn't utilize recursion, but a single for loop to determine the Fibonacci terms. It returns the cache[nr] once it's done computing it.

Implementation:

```
def iterative_fibonacci(nr, cache=None):
    if cache is None:
        cache = [0, 1]

    if nr == 0:
        return cache[0]
    elif nr == 1:
        return cache[1]
    else:
        for i in range(2, nr + 1):
            cache.append(cache[i - 1] + cache[i - 2])
        return cache[nr]
```

Figure 6. Iteration method code

Below, we can see the results displayed in a table:

Results:

N	5	129	254	379	503	628	753	877	1002	1127	1251	1376	1501	1625	1750	1875	2000
Milliseconds	0.01	0.03	0.05	0.07	0.09	0.12	0.16	0.24	0.22	0.24	0.27	0.31	0.31	0.36	0.38	0.45	0.61

Figure 7. Iteration method graph

As we can notice in Figure 7, the execution time of the iterative algorithm is better than the recursion dynamic programming approach, despite having a similar time complexity of O(n). This is due to not having to prioritize other checks when going through the recursion, saving time by computing the terms efficiently.

Binet Formula Algorithm:

Algorithm Description:

The Binet Algorithm is a very performant algorithm that directly computes the nth term of the Fibonacci sequence by utilizing the golden ration formula. However, this is done at the expense of accuracy. Because square root has to be computed (and since it's not an integer, it has rounding errors), and the division doesn't return an int, we have to deal with rounding errors, and when comparing with actual Fibonacci terms, it's pretty clear to see that the error is very significant.

Implementation:

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

Figure 8. Binet formula

```
from math import *


def fibonacci_binet(nr):
    sqrt_5 = sqrt(5)
    golden_ratio = (1 + sqrt_5) / 2

    fib_n = (golden_ratio**nr - (-1 / golden_ratio)**nr) / sqrt_5
    return round(fib_n)
```

Figure 9. Binet formula code

Results:

N	5	629	1254	1879	2503	3128	3753	4377	5002	5627	6251	6876	7501	8125	8750	9375	10000
Milliseconds	1.28	1.22	1.29	1.42	1.50	1.39	1.53	1.56	1.66	1.97	1.81	1.94	1.99	2.27	2.12	2.42	2.16

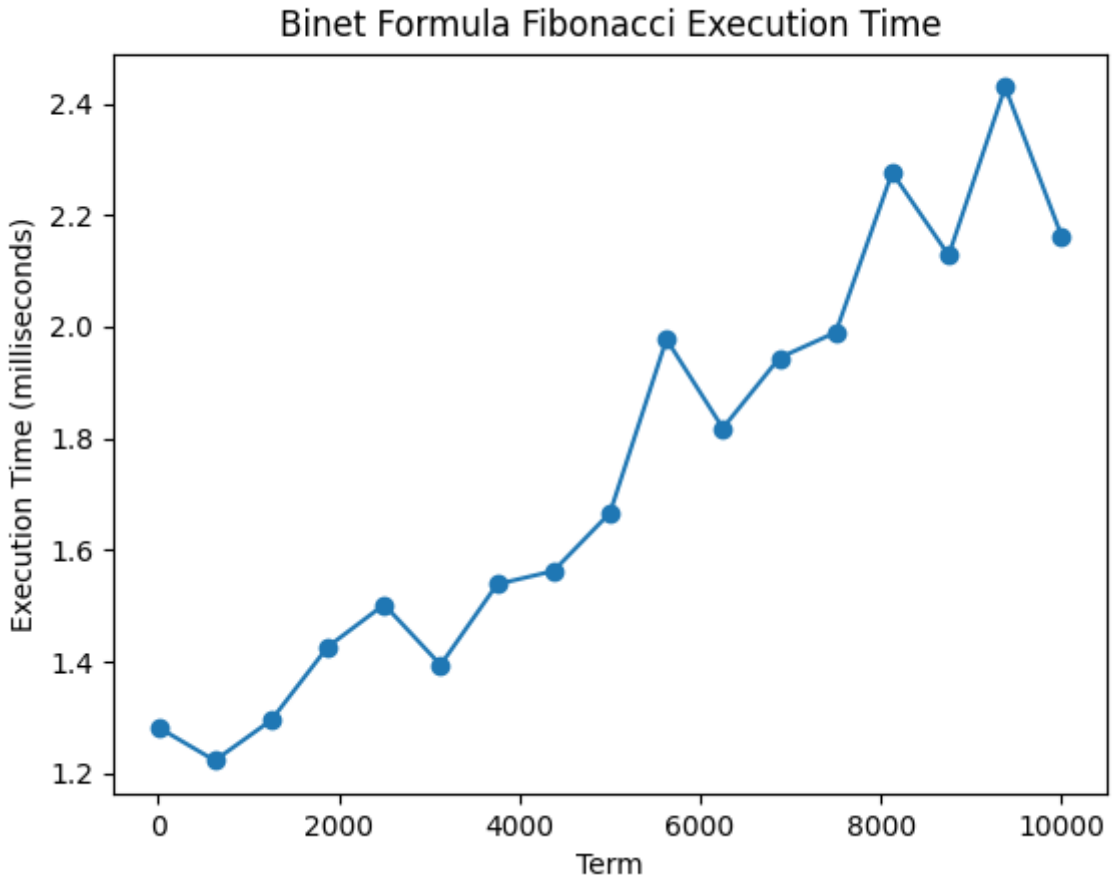

Figure 10. Binet formula graph

In figure 10, we can see the graph describing Binet formula's performance. Due to its time complexity of only O(1), the results were as predicted – the algorithm functioned very fast, taking only 2.16 ms to compute the 10000th term. However, the graph doesn't show the errors, that spike at around the 70th term, and increase to an absurd amount, making this an unreliable algorithm for accurately determining the values of the Fibonacci sequence.

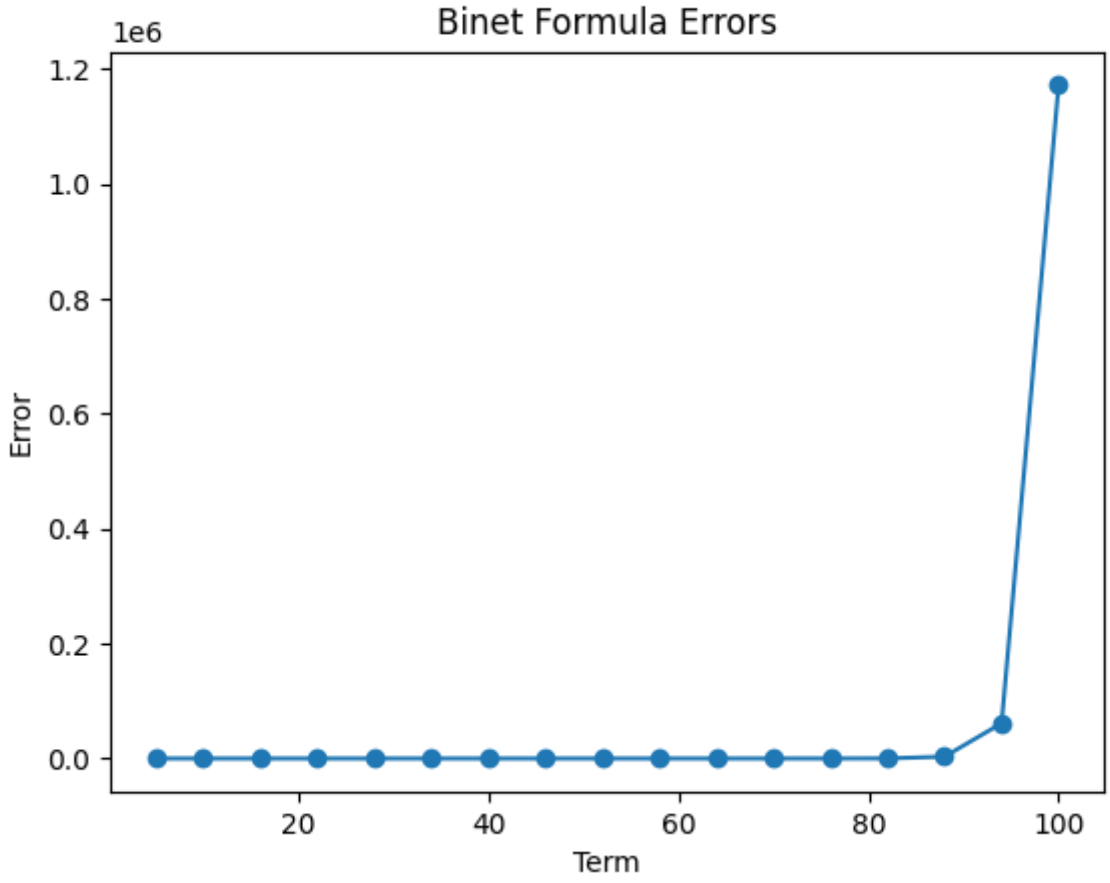

Figure 11. Binet formula error graph

In figure 11, we can see the error representation of the Binet formula. Even though we considered only the first 100th terms, we have an incredible loss of accuracy (which continues to grow, down to the point we can no longer convert integers to floats in python), reaching almost 1.2 * 10^6 (1172004, to be exact). This makes the formula unreliable for doing any sort of computation which requires accurate data.

Matrix Exponentiation Algorithm:

Algorithm Description:

Matrix exponentiation is a technique where a matrix is raised to a power efficiently using divide-and-conquer. In the context of finding the Fibonacci numbers, it is used to optimize the computation by expressing the recurring relation as a matrix and efficiently computing its power, leading to a more efficient algorithm compared to previously explored methods. The time complexity of this algorithm is only O(log(n)).

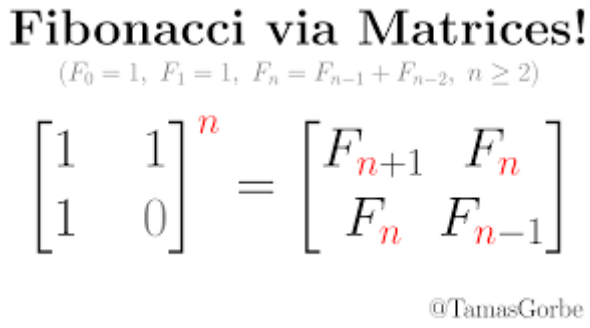

Figure 12. Matrix exponentiation formula

Implementation

```
def multiply(a, b):
    mul = [[0 for x in range(3)]
           for y in range(3)]
    for i in range(3):
        for j in range(3):
            mul[i][j] = 0
            for k in range(3):
                mul[i][j] += a[i][k] * b[k][j]

    for i in range(3):
        for j in range(3):
            a[i][j] = mul[i][j]
    return a


def power(F, n):
    M = [[1, 1, 1], [1, 0, 0], [0, 1, 0]]
    if (n == 1):
        return F[0][0] + F[0][1]
    power(F, int(n / 2))
    F = multiply(F, F)
    if (n % 2 != 0):
        F = multiply(F, M)
    return F[0][0] + F[0][1]

def findNthTerm(n):
    F = [[1, 1, 1], [1, 0, 0], [0, 1, 0]]
    return power(F, n - 2)
```

Figure 13. Matrix exponentiation code

Results:

N	5	629	1254	1879	2503	3128	3753	4377	5002	5627	6251	6876	7501	8125	8750	9375	10000
Milliseconds	0.05	1.08	0.52	0.32	0.48	0.49	0.54	0.58	0.69	0.98	0.77	0.87	1.06	1.12	1.14	1.28	1.43

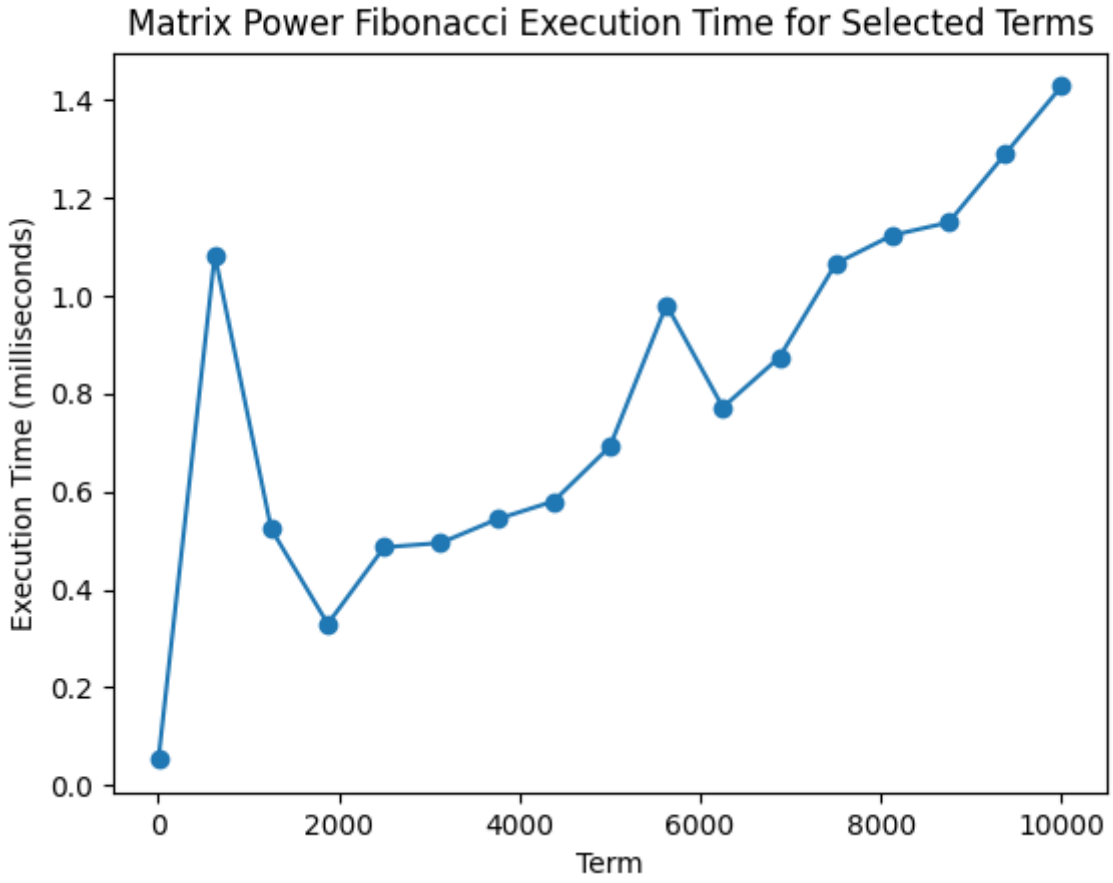

Figure 14. Matrix exponentiation graph

In figure 14, we can notice that the matrix power method is very fast. Completing the 10000th term took only 1.43 ms, which is very fast and efficient, boasting an incredible O(log n) time complexity, beating basically every algorithm we've used beforehand. It's also very important to note that the algorithm didn't lose any accuracy during the computation, providing exact results.

Fast Doubling Algorithm:

Algorithm Description:

The Fast Doubling Algorithm serves as an improvement to the Matrix Exponentiation algorithm, avoiding the computing Fn (as in the Matrix Exponentiation). Instead, it uses the simplified formula in Figure 14. Although it has a time complexity of O(log n), it is faster than the previous method.

$$\begin{bmatrix} F(2n+1) \\ F(2n) \end{bmatrix} = \begin{bmatrix} F(n+1)^2 + F(n)^2 \\ 2F(n+1)F(n) - F(n)^2 \end{bmatrix}$$

Figure 15. Fast Doubling Algorithm

Implementation:

```
MOD = 1000000007

def FastDoubling(n, res):
    if (n == 0):
        res[0] = 0
        res[1] = 1
        return

    FastDoubling((n // 2), res)
    a = res[0]
    b = res[1]
    c = 2 * b - a
    if (c < 0):
        c += MOD
    c = (a * c) % MOD
    d = (a * a + b * b) % MOD
    if (n % 2 == 0):
        res[0] = c
        res[1] = d
    else:
        res[0] = d
        res[1] = c + d
```

Figure 16. Fast Doubling code

Results:

N	5	629	1254	1879	2503	3128	3753	4377	5002	5627	6251	6876	7501	8125	8750	9375	10000
Milliseconds	0.018	0.016	0.013	0.012	0.012	0.012	0.012	0.013	0.013	0.013	0.013	0.013	0.013	0.013	0.014	0.014	0.014

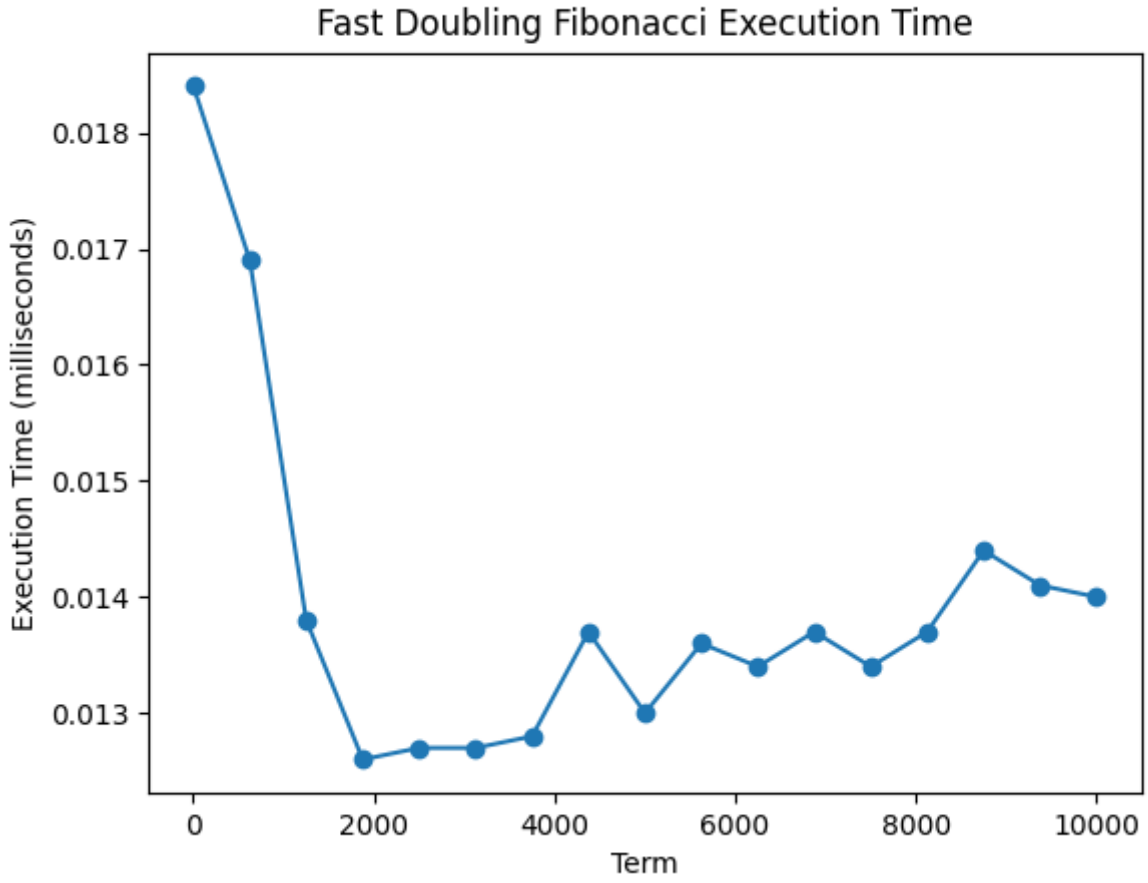

Figure 17. Fast Doubling time complexity graph

Despite having a time complexity similar to the matrix exponentiation algorithm, of O(log n), the Fast Doubling algorithm performs much better than its predecessor. As observed in Figure 17, the fast doubling algorithm outperforms the matrix exponentiation algorithm. The downside to this is that this program will not work for Fibonacci terms that return very high values - higher than ~10^10, as they're reduced by the modulus operator.

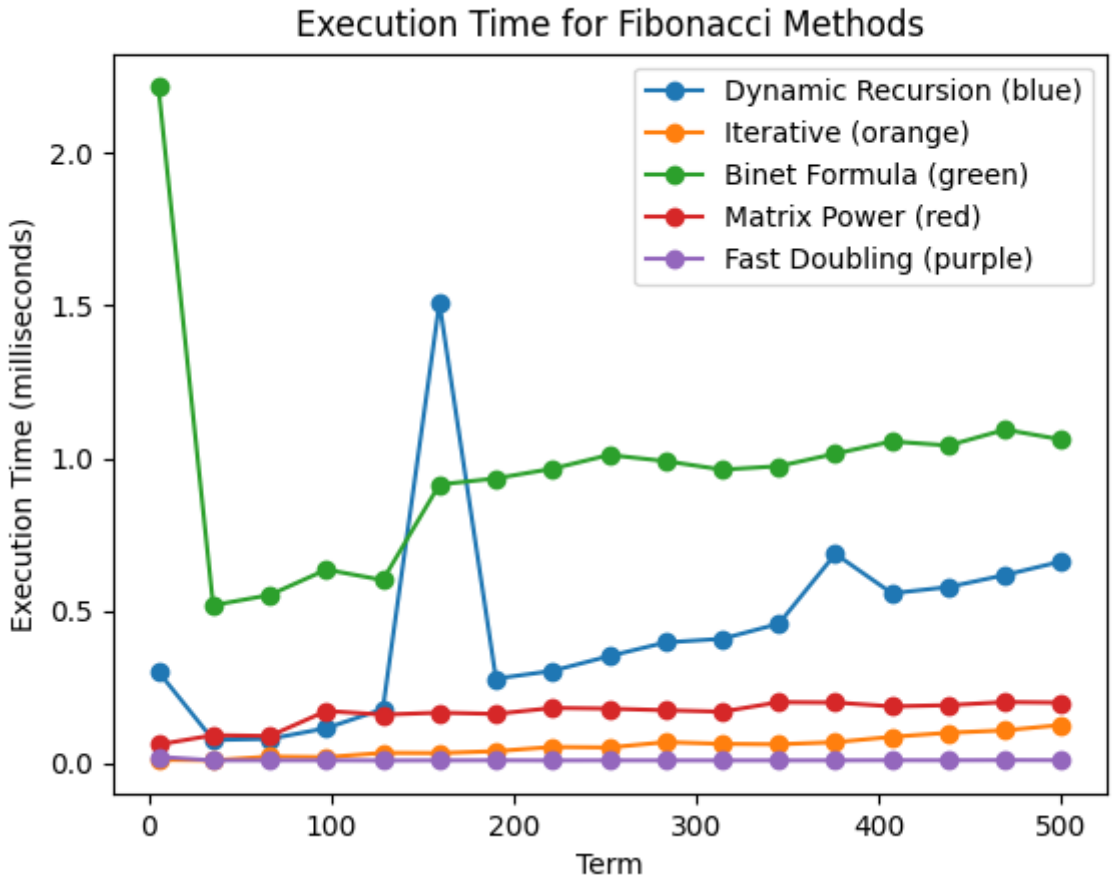

Figure 18. Time complexity comparison for lower Fibonacci terms

In figure 18, we can observe how the algorithms behave for a lower Fibonacci term. Despite the matrix exponentiation and Binet formula being slower in this graph, this is due to the term we're using - 500, and leaves room for errors.

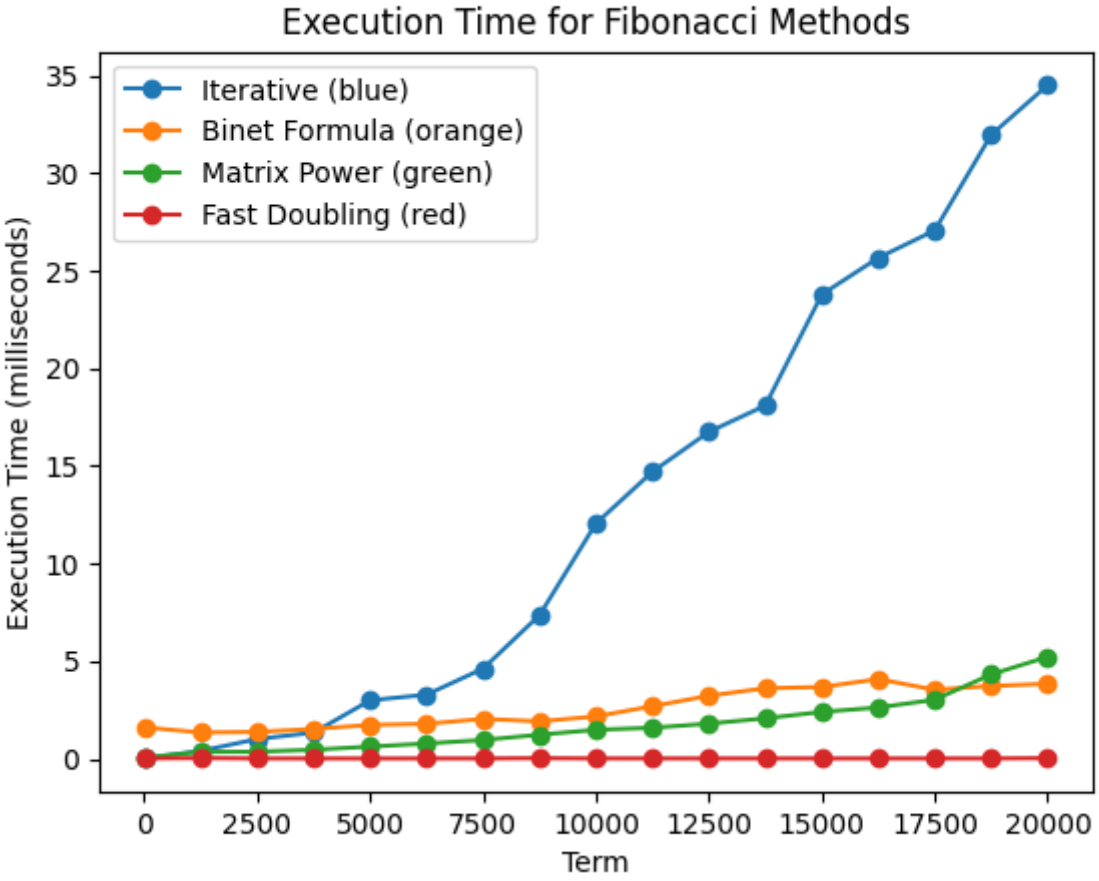

Figure 19. Time complexity comparison for higher Fibonacci terms

In Figure 19, we operate with much higher Fibonacci terms. We can notice that the iterative algorithm starts to slow down, while the Binet formula, matrix exponentiation and fast doubling maintain their pace. The dynamic programming (recursion) algorithm was removed due to recursion limitations in Python.

CONCLUSION

After finishing this laboratory work, I learnt about 6 different ways to compute the Fibonacci sequence. The first one – the recursive approach – although it's the slowest, it's good enough to find the lower terms (up to 40) of the Fibonacci sequence relatively quickly. After 40, however, this method becomes increasingly inefficient. The second one – the dynamic programming approach, is much better, but we're still limited by the maximum recursion limit, which is predefined by the system, and changing it to a higher number leads to a stack overflow error. The third one, the iterative approach, combines simplicity with efficiency. For a couple of lines of code, we can efficiently obtain the higher numbers of the Fibonacci sequence very quickly (~0.6 ms for the 10000^{th} term).

The Binet formula is the 4^{th} method used to obtain the Fibonacci sequence values. Although it's relatively fast, having a complexity of O(1) on paper, overall this algorithm came in third by execution time. This is probably due to constant rounding the decimals and multiple computations which lead to slower runtime and computation of terms. It also proved to generate very high errors (Figure 11), which makes it unreliable. The 5^{th} method was the matrix exponentiation algorithm. It was the 2^{nd} fastest algorithm, falling short only to the fast doubling algorithm. It provided very good execution time AND accurate results, proving to be a very efficient O(log n) algorithm. It could be used to compute very high Fibonacci terms, without a significant increase in computational time. The last algorithm was also the fastest – the fast doubling algorithm – although it had its own drawbacks. This algorithm managed to outperform the matrix exponentiation method by a significant amount, however its downside is that it falls short in terms of computing big numbers (>10^10), since the program won't work for those numbers, and it will only return the remaining number after the modulus operation; thus if we want to obtain the true values of higher numbers, we must use another method.

The most important lesson I learnt from this laboratory work is that different approaches to a problem can solve it far more efficiently. Additionally, algorithms that are at first inefficient can be optimized – i.e., using dynamic programming & optimizing the total amount of recursions by storing values in a cache. The big O notation is a great indication of how fast an algorithm could run, but it doesn't determine everything. Applying an algorithm might require other modules, which might make them take far longer than predicted – as noticed in the Binet formula vs matrix exponentiation algorithm (O(1) vs O(log n)). Lastly, depending on the Fibonacci term we need to compute, we can choose more efficient algorithms, such as the fast doubling algorithm, as they allow us to compute the terms lower than 10^10 in a very quick runtime.

GitHub repository link - https://github.com/Lucian-Lu/AA-Labs