Ministry of Education of Republic of Moldova
Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics

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

Author:
st. gr. FAF-242 Pancenco Ina
Verified:
asist. univ. Fiștic Cristofor

Chișinău - 2026

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.

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 a numerical series where each term is the sum of the two preceding values, defined mathematically as $x_n = x_{n-1} + x_{n-2}$. Although named after the 12th-century mathematician Leonardo of Pisa, known as Fibonacci, historical evidence suggests the sequence appeared in ancient Sanskrit texts using the Hindu-Arabic numeral system centuries earlier.

Modern computer science has expanded Fibonacci determination into four primary methodologies: Recursive, Dynamic Programming, Matrix Power, and Binet's Formula. These approaches can be implemented naively or optimized to enhance performance, which is analyzed either mathematically through reasoning or empirically via observation. This laboratory focuses on the empirical analysis of four naive algorithmic implementations to evaluate their experimental efficiency. In this laboratory work we will be doing the Empirical Analysis for 5 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, (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 five algorithms will be implemented in their naïve form and some of them in an optimized form to compare both in python and empirically analyzed 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 o memory of the device used.

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

Recursive Method:

The recursive method, also considered the most inefficient method, follows a straightforward approach of computing the n-th term by computing it's predecessors first, and then adding them. However, the method does it by calling upon itself a number of times and repeating the same operation, for the same term, at least twice, occupying additional memory and, in theory, doubling it's execution time.

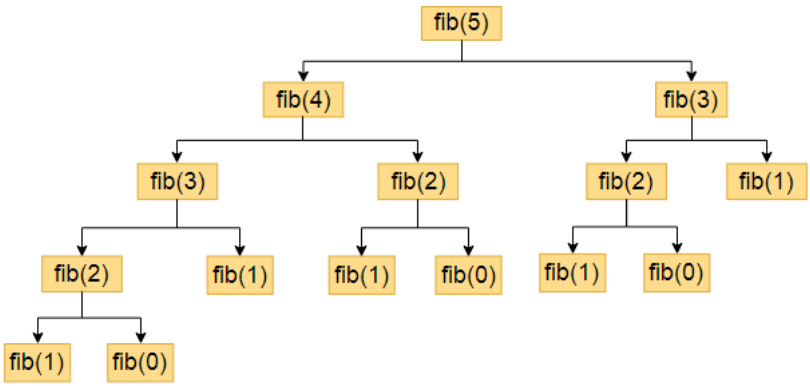


Figure 1: Fibonacci Recursion

Algorithm Description:

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

Listing 1: Recursive Fibonacci (pseudocode)

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

Implementation:

```python
def fibonacci_naive(n):
    global CALL_COUNT
    CALL_COUNT += 1
    if n <= 1:
        return 1
    return fibonacci_naive(n - 1) + fibonacci_naive(n - 2)
```

Figure 2: Fibonacci recursion in Python

Results:

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	run1(ms)	run2(ms)	run3(ms)	avg(ms)
5	0.010	0.004	0.002	0.005
10	0.038	0.036	0.036	0.037
15	0.280	0.249	0.250	0.260
20	3.113	2.883	3.411	3.136
25	42.835	32.291	32.137	35.754
30	400.203	387.086	395.926	394.405

Figure 3: Results for first set of inputs

Figure 3 represents the table of results for the first set of inputs. The first row (the column headers) indicates the values of the Fibonacci n-th term for which the algorithm was executed. Starting from the second row, the table shows the execution time in milliseconds recorded for three runs, as well as the average execution time for each value of n.

Therefore, these results clearly illustrate the exponential growth of the execution time, which is characteristic of the Recursive Method for computing the Fibonacci sequence. As the value of n increases, the number of repeated computations grows rapidly, leading to a dramatic increase in the total running time.

Memoized Recursion Algorithm Description:

Listing 2: Memoized recursion (pseudocode)

```
Fibonacci(n):
    create array M of size n+1
    for i <- 0 to n do
        M[i] <- -1
    return FibMemo(n, M)

FibMemo(n, M):
    if M[n] != -1 then
        return M[n]
    if n = 0 then
        M[0] <- 0
    else if n = 1 then
        M[1] <- 1
    else
        M[n] <- FibMemo(n-1, M) + FibMemo(n-2, M)
    return M[n]
```

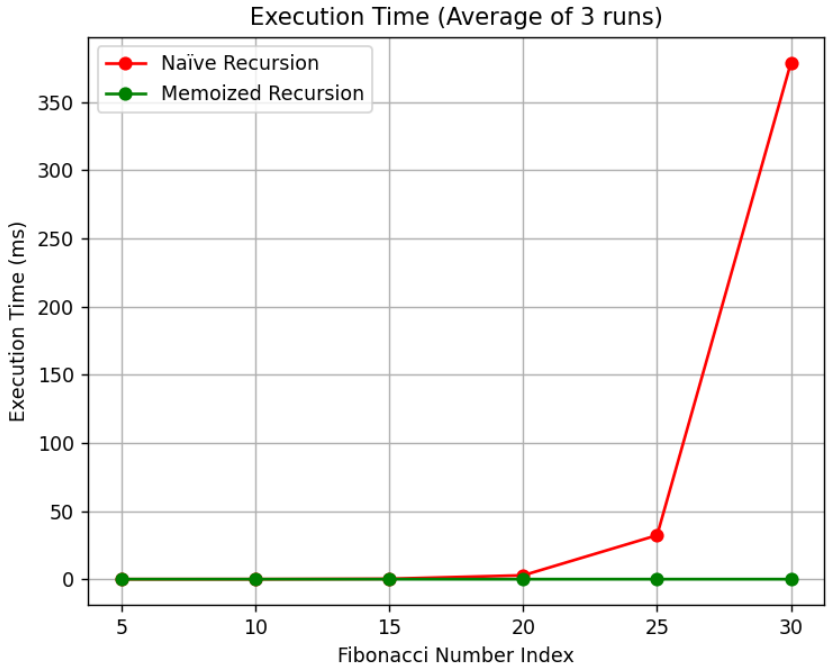


Figure 4: Graph of Recursive Fibonacci Function

In the graph in *Figure 4* that shows the growth of the time needed for the operations, we may easily see the dramatic spike in time complexity for the Naïve Recursion that begins to accelerate after the 20th term and climbs sharply by the 30th term. This behavior leads us to deduce that its time complexity is exponential, denoted as $O(2^n)$. In stark contrast, the Memoized Recursion maintains an execution time consistently near

0 ms across all tested indices, demonstrating the efficiency of a linear time complexity, $O(n)$, by avoiding redundant calculations.

Dynamic Programming Method:

The Dynamic Programming method, similar to the recursive method, takes the straightforward approach of calculating the n-th term. However, instead of calling the function upon itself, from top down it operates based on an array data structure that holds the previously computed terms, eliminating the need to recompute them.

Algorithm Description:

The naïve DP algorithm for Fibonacci n-th term follows the pseudocode:

Listing 3: Dynamic Programming Method (pseudocode)

```
Fibonacci(n):
    Array A;
    A[0]<-0;
    A[1]<-1;
    for i <- 2 to n      1 do
        A[i]<-A[i-1]+A[i-2];
    return A[n-1]
```

Implementation:

```
def f(x):
    l1 = [0, 1]

    for i in range(2, x + 1):
        l1.append(l1[i-1] + l1[i-2])

    return l1[x]
```

Figure 5: Fibonacci DP in Python

Results:

After the execution of the function for each n Fibonacci term mentioned in the second set of Input Format we obtain the following results:

	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
0	0.0	0.0	0.000726	0.0	0.000758	0.0007	0.000360	0.003646	0.001101	0.001459	0.001831	0.002550	0.004351	0.005489	0.019695	0.014626
1	0.0	0.0	0.000000	0.0	0.000000	0.0000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001074	0.000000	0.000727
2	0.0	0.0	0.000000	0.0	0.000000	0.0000	0.033937	0.014543	0.013128	0.013862	0.020456	0.009831	0.025155	0.037205	0.076604	0.064923

Figure 6: Fibonacci DP Results

With the Dynamic Programming Method (first row, row[0]) showing excellent results with a time complexity denoted in a corresponding graph of T(n),

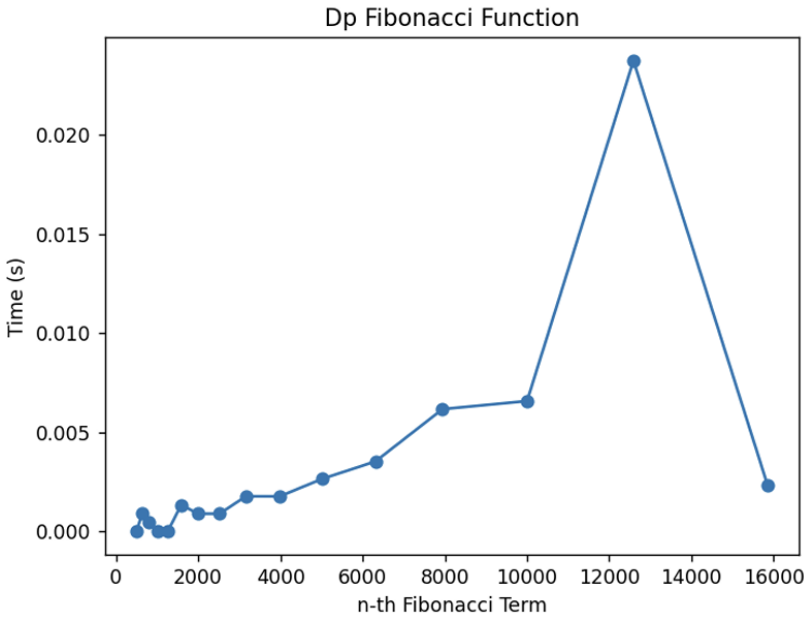


Figure 7: Fibonacci DP Graph

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).

Matrix Power Method:

The Matrix Power method of determining the n-th Fibonacci number is based on, as expected, the multiple multiplication of a naïve Matrix $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ with itself.

Algorithm Description:

It is known that

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} b \\ a+b \end{pmatrix}$$

This property of Matrix multiplication can be used to represent

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \end{pmatrix}$$

And similarly:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_2 \\ F_3 \end{pmatrix}$$

Which turns into the general:

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

This set of operation can be described in pseudocode as follows:

Listing 4: Matrix Power Method (pseudocode)

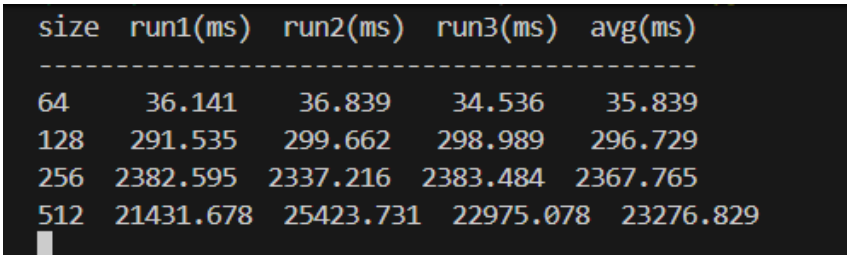
```
Fibonacci(n):
    F<- []
    vec <- [[0], [1]]
    Matrix <- [[0, 1],[1, 1]]
    F <-power(Matrix, n)
    F <- F * vec
    Return F[0][0]
```

The Matrix Power Method presented in Listing 4 utilizes linear transformations to compute the n^{th} Fibonacci term. It defines a transformation matrix $M = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ and calculates M^n to transition from the base case $[0, 1]$ to the desired index. By employing the `power` function, likely via exponentiation by squaring, the algorithm achieves a logarithmic time complexity, $O(\log n)$, making it one of the most efficient approaches for very large values of n.

```python
def matrix_multiply_naive(A, B):
    n = len(A)
    C = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]
    return C
```

Figure 8: Naive Multiply Function Python

Result : After the execution of the function for each n Fibonacci term mentioned in the second set of Input Format we obtain the following results:



size	run1(ms)	run2(ms)	run3(ms)	avg(ms)
64	36.141	36.839	34.536	35.839
128	291.535	299.662	298.989	296.729
256	2382.595	2337.216	2383.484	2367.765
512	21431.678	25423.731	22975.078	23276.829

Figure 9: Matrix Method Fibonacci Results

The experimental results for the Matrix Method, as detailed in *Figure 9*, illustrate the impact of increasing input size on the algorithm's execution time. This progression suggests that while the method is powerful, the computational overhead grows substantially with input scale, reflecting the underlying complexity of the matrix operations performed during each run.

Implementation of Numpy Optimized method:

```
def matrix_multiply_numpy(A, B):
    return np.dot(A, B)
```

Figure 10: Numpy Optimized Matrix Method

Figure 10 shows the implementation of matrix multiplication utilizing the NumPy library's `np.dot()` function. Unlike standard Python loops, this approach employs vectorized operations that drastically reduce computational overhead. When integrated into the Matrix Power Method, it ensures that the constant-time factors of the $O(\log n)$ complexity remain as low as possible, facilitating the rapid calculation of extremely high-order Fibonacci terms.

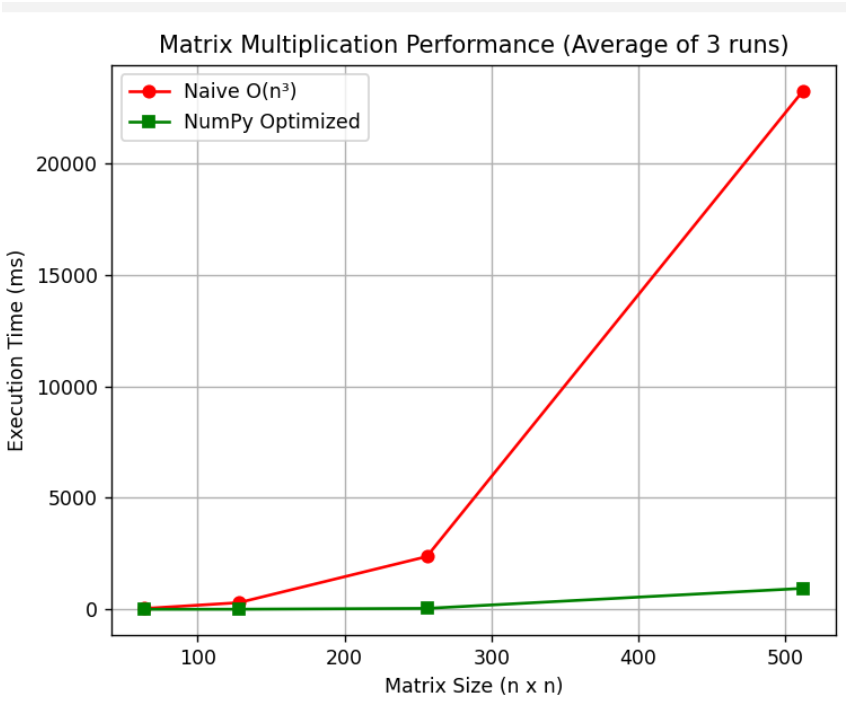


Figure 11: Matrix Method Fibonacci graph

Binet Formula Method:

The Binet Formula Method is another unconventional way of calculating the n-th term of the Fibonacci series, as it operates using the Golden Ratio formula, or phi. However, due to its nature of requiring the usage of decimal numbers, at some point, the rounding error of python that accumulates, begins affecting the results significantly. The observation of error starting with around 70-th number making it unusable in practice, despite its speed.

Algorithm Description:

The set of operation for the Binet Formula Method can be described in pseudocode as follows:

Listing 5: Binet Formula Method(pseudocode)

```
Fibonacci(n):
    phi <- (1 + sqrt(5))
    phi1 <-(1     sqrt(5))
    return pow(phi, n)- pow(phi1, n)/(pow(2, n)*sqrt(5))
```

Implementation:

The implementation of the function in Python is as follows, with some alterations that would increase the number of terms that could be obtain through it:

```python
def fibonacci_binet(n):
    phi = (1 + math.sqrt(5)) / 2
    phi1 = (1 - math.sqrt(5)) / 2
    return round((phi**n - phi1**n) / math.sqrt(5))
```

Figure 12: Fibonacci Binet Formula Method in Python

Result :

```
=== BINET FORMULA METHOD ===
n   run1(ms)   run2(ms)   run3(ms)   avg(ms)
-------------------------------------
5         0.021      0.007      0.002      0.010
-------------------------------------
5         0.021      0.007      0.002      0.010
10        0.006      0.003      0.002      0.004
15        0.005      0.001      0.001      0.002
5         0.021      0.007      0.002      0.010
10        0.006      0.003      0.002      0.004
15        0.005      0.001      0.001      0.002
10        0.006      0.003      0.002      0.004
15        0.005      0.001      0.001      0.002
20        0.003      0.001      0.002      0.002
20        0.003      0.001      0.002      0.002
25        0.002      0.001      0.001      0.001
30        0.002      0.001      0.001      0.001
30        0.002      0.001      0.001      0.001
35        0.003      0.001      0.001      0.002
40        0.003      0.001      0.001      0.002
50        0.003      0.001      0.002      0.002
100       0.003      0.001      0.001      0.002
500       0.002      0.001      0.001      0.002
1000      0.003      0.001      0.001      0.002
```

Figure 13: Fibonacci Binet Formula Method results

The empirical results for the Binet Formula Method, as shown in the experimental data table, demonstrate exceptional performance consistency across all tested values of n. Unlike recursive or matrix-based approaches, the Binet method maintains an average execution time near 0.001 to 0.002 ms, even as n increases from 5 to 1000. For instance, at $n = 5$, the average recorded time is 0.010 ms, while at $n = 1000$, it remains remarkably low at 0.002 ms. This stability confirms that the closed-form mathematical approach provides a near $O(1)$ time complexity, as it relies on a direct calculation rather than iterative or recursive steps.

And as shown in its performance graph,

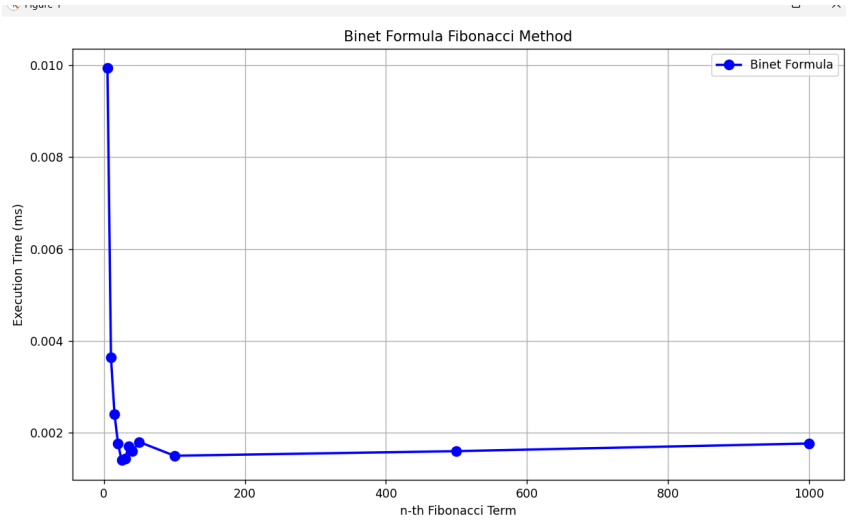


Figure 14: Fibonacci Binet formula Method

The Binet Formula Function is not accurate enough to be considered within the analysed limits and is recommended to be used for Fibonacci terms up to 80. At least in its naïve form in python, as further modification and change of language may extend its usability further.

Doubling Method:

The Fast Doubling method calculates the n^{th} Fibonacci number by recursively halving the index. Given the values of F_k and F_{k+1}, the values for the doubled indices are calculated as:

$$F_{2k} = F_k(2F_{k+1} - F_k)$$
$$F_{2k+1} = F_{k+1}^2 + F_k^2$$

Listing 6: Fast Doubling Algorithm

```
FastDoubling(n):
    if n == 0:
        return (0, 1)
    (a, b) = FastDoubling(n >> 1)
    c = a * (2 * b - a)
    d = a * a + b * b
    if n is even:
        return (c, d)
    else:
        return (d, c + d)
```

Implementation: The implementation of the function in Python is as follows, with some alterations that would increase the number of terms that could be obtain through it:

```
def fib_doubling(k):
    if k == 0:
        return (0, 1)
    f_m, f_m1 = fib_doubling(k // 2)
    c = f_m * (2 * f_m1 - f_m)
    d = f_m * f_m + f_m1 * f_m1

    if k % 2 == 0:
        return (c, d)
    else:
        return (d, c + d)

return fib_doubling(n)[0]
```

Figure 15: Fibonacci Doubling Method

Result :

n	run1(ms)	run2(ms)	run3(ms)	avg(ms)
10	0.014	0.006	0.003	0.008
20	0.007	0.005	0.004	0.005
50	0.011	0.006	0.005	0.007
100	0.011	0.006	0.024	0.014
200	0.015	0.008	0.007	0.010
500	0.016	0.017	0.009	0.014
1000	0.017	0.011	0.010	0.013
2000	0.015	0.011	0.008	0.011
5000	0.038	0.024	0.022	0.028
10000	0.102	0.093	0.086	0.094

Figure 16: Fibonacci Doubling Method results

The logarithmic nature of the algorithm, $O(\log n)$, is clearly evidenced by the performance at higher ranges. While the Naïve Recursion model showed an exponential spike, the Fast Doubling method handles $n = 5000$ in 0.028 ms and reaches $n = 10000$ with an average time of only 0.094 ms. These results confirm that the Fast Doubling approach is vastly superior to the Matrix Method for large-scale calculations, providing stable and rapid performance without the significant computational overhead observed in iterative or basic matrix implementations.

And as shown in its performance graph,

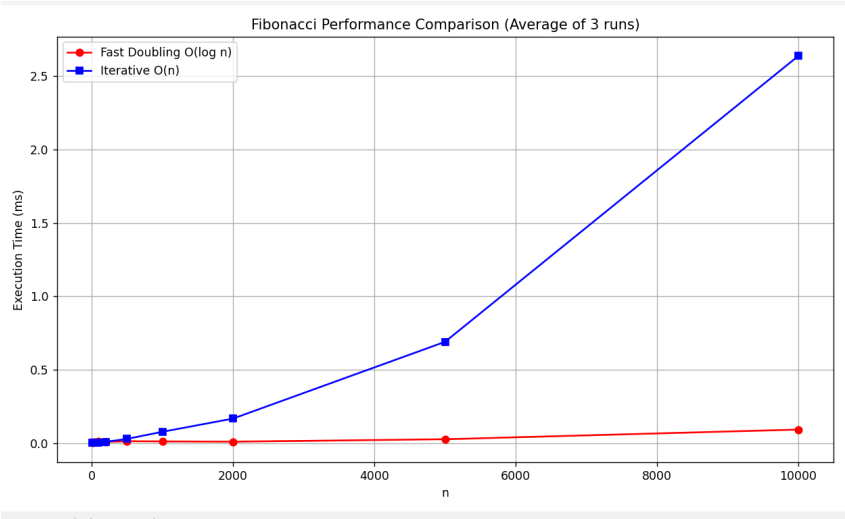


Figure 17: Fibonacci Fast Doubling Method

I have chosen to compare the Fast Doubling Method to the simple Iterative Fibonnaci Method. The performance comparison highlights the significant efficiency and gains provided by logarithmic complexity. The Iterative method, characterized by an $O(n)$ time complexity, displays a clear linear growth in execution time. Starting from near-zero values for small n, it climbs steadily to approximately 0.7 ms at $n = 5000$ and reaches over 2.5 ms at $n = 10000$.

Conversely, the Fast Doubling method, which operates with $O(\log n)$ complexity, maintains a nearly flat execution profile across the entire tested range. Even as the input size n scales to 10000, the execution time remains exceptionally low at 0.094 ms. This stark contrast in the slopes of the two functions empirically validates that the Fast Doubling approach is the more robust solution for calculating high-order Fibonacci numbers in large-scale computational tasks.

CONCLUSION

Through Empirical Analysis, within this paper, five classes of methods have been tested in their efficiency at both their providing of accurate results, as well as at the time complexity required for their execution, to delimit the scopes within which each could be used, as well as possible improvements that could be further done to make them more feasible.

The Recursive method, being the easiest to write, but also the most difficult to execute with an exponential time complexity, can be used for smaller order numbers, such as numbers of order up to 30 with no additional strain on the computing machine and no need for testing of patience.

The Binet method, the easiest to execute with an almost constant time complexity, could be used when computing numbers of order up to 80, after the recursive method becomes unfeasible. However, its results are recommended to be verified depending on the language used, as there could rounding errors due to its formula that uses the Golden Ratio.

The Dynamic Programming and Matrix Multiplication Methods can be used to compute Fibonacci numbers further then the ones specified above, both of them presenting exact results and showing a linear complexity in their naivety that could be, with additional tricks and optimisations, reduced to logarithmic.

The Fast Doubling Method represents the most robust solution for large-scale computations, leveraging specific mathematical identities to achieve a logarithmic time complexity of $O(\log n)$. Avoiding both the exponential growth of recursion and the precision issues inherent in Binet's formula, this method is ideal for calculating high-order Fibonacci numbers where both speed and exactness are required.

REFERENCES

[1] **P., I. (2026).** *Analysis of Algorithms*. GitHub Repository. https://github.com/inap235/AALab

[2] GeeksforGeeks. (2023). *Binet's Formula for Fibonacci Numbers*. https://www.geeksforgeeks.org/binets-formula-for-fibonacci-numbers/

[3] CP-Algorithms. (2022). *Fibonacci Numbers - Fast Doubling Method*. https://cp-algorithms.com/algebra/fibonacci-numbers.html#fast-doubling-method

[4] Khan Academy. (2024). *Recursive Fibonacci and its Time Complexity*. https://www.khanacademy.org/computing/computer-science/algorithms/recursive-algorithms/a/recursive-fibonacci

[5] Towards Data Science. (2021). *Fibonacci Sequence With Matrix Exponentiation*. https://towardsdatascience.com/fibonacci-sequence-with-matrix-exponentiation-c7f70415ee5

[6] Programiz. (2024). *Dynamic Programming: Memoization and Tabulation*. https://www.programiz.com/dsa/dynamic-programming