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-241 Moisei Daniel
Verified:
asist. univ. Fiștic Cristofor

Chișinău - 2025

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

Many sources claim this sequence was first discovered or "invented" by Leonardo Fibonacci. The Italian mathematician, who was born around A.D. 1170, was initially known as Leonardo of Pisa. In the 19th century, historians came up with the nickname Fibonacci (roughly meaning "son of the Bonacci clan") to distinguish the mathematician from another famous Leonardo of Pisa. There are others who say he did not. Keith Devlin, the author of Finding Fibonacci: The Quest to Rediscover the Forgotten Mathematical Genius Who Changed the World, says there are ancient Sanskrit texts that use the Hindu-Arabic numeral system - predating Leonardo of Pisa by centuries. But, in 1202 Leonardo of Pisa published a mathematical text, Liber Abaci. It was a "cookbook" written for tradespeople on how to do calculations. The text laid out the Hindu-Arabic arithmetic useful for tracking profits, losses, remaining loan balances, etc, introducing the Fibonacci sequence to the Western world.

Traditionally, the sequence was determined just by adding two predecessors to obtain a new number, however, with the evolution of computer science and algorithmics, several distinct methods for determination have been uncovered. The methods can be grouped in 4 categories, Recursive Methods, Dynamic Programming Methods, Matrix Power Methods, and Benet Formula Methods. All those can be implemented naively or with a certain degree of optimization, that boosts their performance during analysis.

As mentioned previously, the performance of an algorithm can be analyzed mathematically (derived through mathematical reasoning) or empirically (based on experimental observations).

Within this laboratory, we will be analyzing the 4 naïve algorithms empirically.

Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n))

Input Format:

As input, each algorithm will receive two series of numbers that will contain the order of the Fibonacci terms being looked up. The first series will have a more limited scope, (5, 7, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 37, 40, 42, 45), to accommodate the recursive method, while the second series will have a bigger scope to be able to compare the other algorithms between themselves (501, 631, 794, 1000, 1259, 1585, 1995, 2512, 3162, 3981, 5012, 6310, 7943, 10000, 12589, 15849).

IMPLEMENTATION

All four algorithms will be implemented in their naïve form in python an analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending 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.

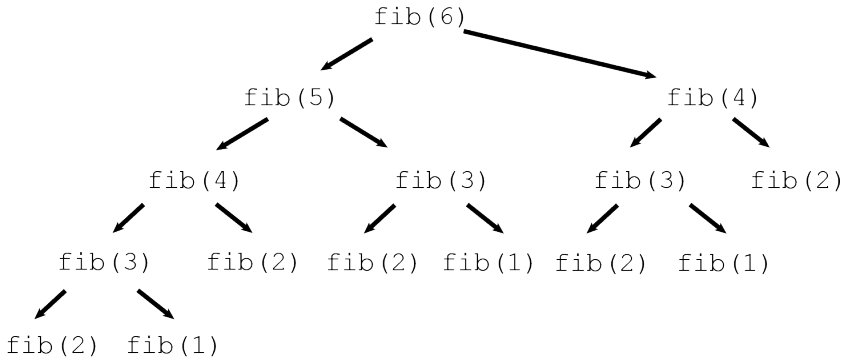

Figure 1: Fibonacci Recursion

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:

```
def fibonacci(x):
    if x <= 1:
        return x
    else:
        return fibonacci(x-1)+ fibonacci(x-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:

	5	7	10	12	15	17	20	22	25	27	30	32	35	37	40	42	45
0	0.0	0.0	0.0	0.0	0.001	0.0011	0.0022	0.0	0.075	0.14	0.66	1.87	7.44	21.11	55.05	136.89	790.019
1	0.0	0.0	0.0	0.0	0.000	0.0000	0.0000	0.0	0.000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.000
2	0.0	0.0	0.0	0.0	0.000	0.0000	0.0000	0.0	0.000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.000
3	0.0	0.0	0.0	0.0	0.000	0.0000	0.0000	0.0	0.000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.000

Figure 3: Results for first set of inputs

In Figure 3 is represented the table of results for the first set of inputs. The highest line(the name of the columns) denotes the Fibonacci n-th term for which the functions were run. Starting from the second row, we get the number of seconds that elapsed from when the function was run till when the function was executed. We may notice that the only function whose time was growing for this few n terms was the Recursive Method Fibonacci function.

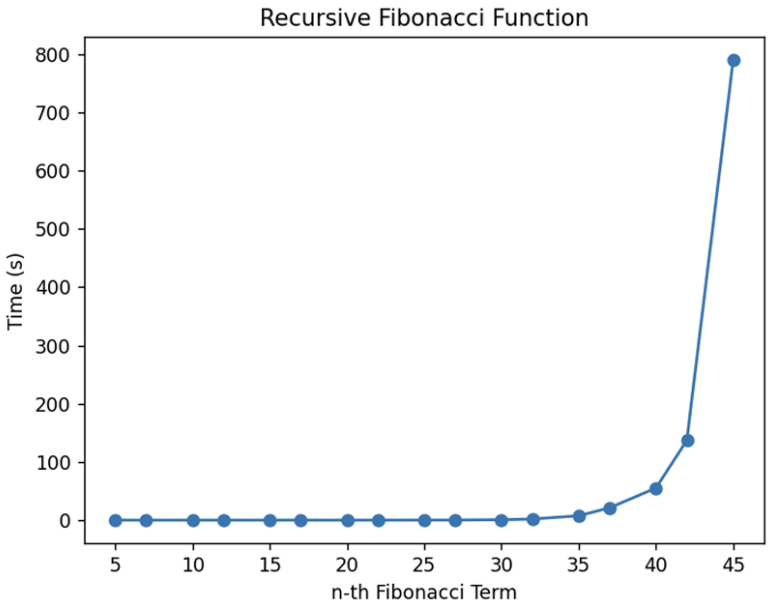

Figure 4: Graph of Recursive Fibonacci Function

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. $T(2^n)$.

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 2: 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:

```python
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),

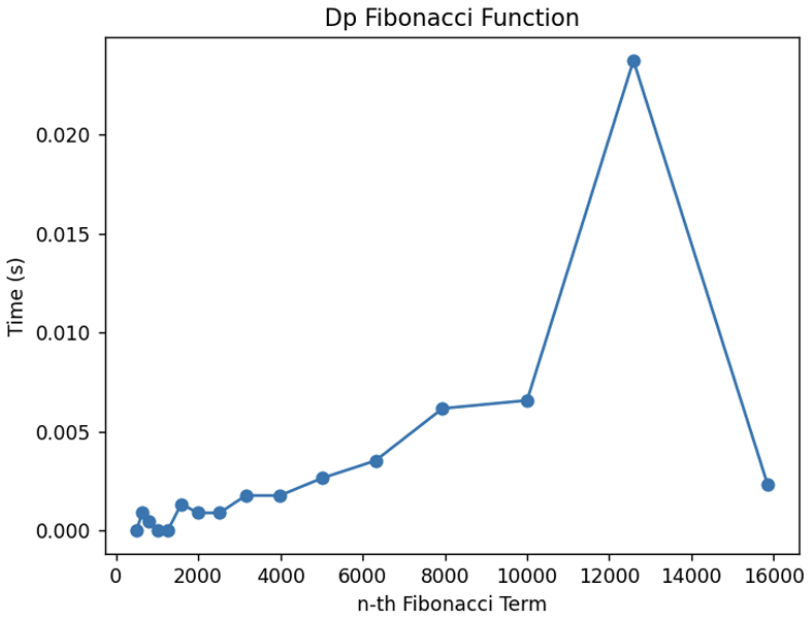

Figure 7: Fibonacci DP Graph

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 3: 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]
```

Implementation: The implementation of the driving function in Python is as follows:

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

    return F[0][0]
```

Figure 8: Fibonacci Matrix Power Method in Python

With additional miscellaneous functions:

```
def power(F, n):

    M = [[1, 1],
         [1, 0]]

    for i in range(2, n + 1):
        multiply(F, M)
```

Figure 9: Power Function Python

Where the power function (Figure 8) handles the part of raising the Matrix to the power n, while the multiplying function (Figure 9) handles the matrix multiplication with itself.

```
def multiply(F, M):

    x = (F[0][0] * M[0][0] +
         F[0][1] * M[1][0])
    y = (F[0][0] * M[0][1] +
         F[0][1] * M[1][1])
    z = (F[1][0] * M[0][0] +
         F[1][1] * M[1][0])
    w = (F[1][0] * M[0][1] +
         F[1][1] * M[1][1])

    F[0][0] = x
    F[0][1] = y
    F[1][0] = z
    F[1][1] = w
```

Figure 10: 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:

	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 11: Matrix Method Fibonacci Results

With the naïve Matrix method (indicated in last row, row[2]), although being slower than the Binet and Dynamic Programming one, still performing pretty well, with the form f the graph indicating a pretty solid T(n) time complexity.

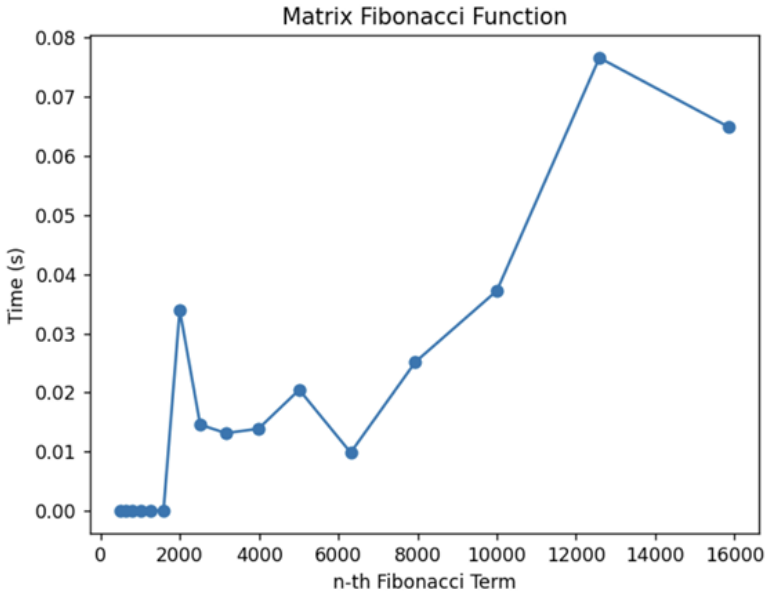

Figure 12: Matrix Method Fibonacci graph

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 4: Matrix Power Method (pseudocode)

```
Fibonacci(n):
    phi <- (1 + sqrt(5))
    phi1 <-(1     sqrt(5))
```

```
4     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:

```
def fib(x):
    ctx = Context(prec=60, rounding=ROUND_HALF_EVEN)
    phi = Decimal((1 + Decimal(5**(1/2))))
    phi2 = Decimal((1 - Decimal(5**(1/2))))


    return int((ctx.power(phi, Decimal(x)) - ctx.power(phi2, Decimal(x)))/( 2**x * Decimal(5**(1/2))))
```

Figure 13: Fibonacci Binet Formula Method in Python

Result : Although the most performant with its time, as shown in the table of results, in row [1],

	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 14: Fibonacci Binet Formula Method results

And as shown in its performance graph,

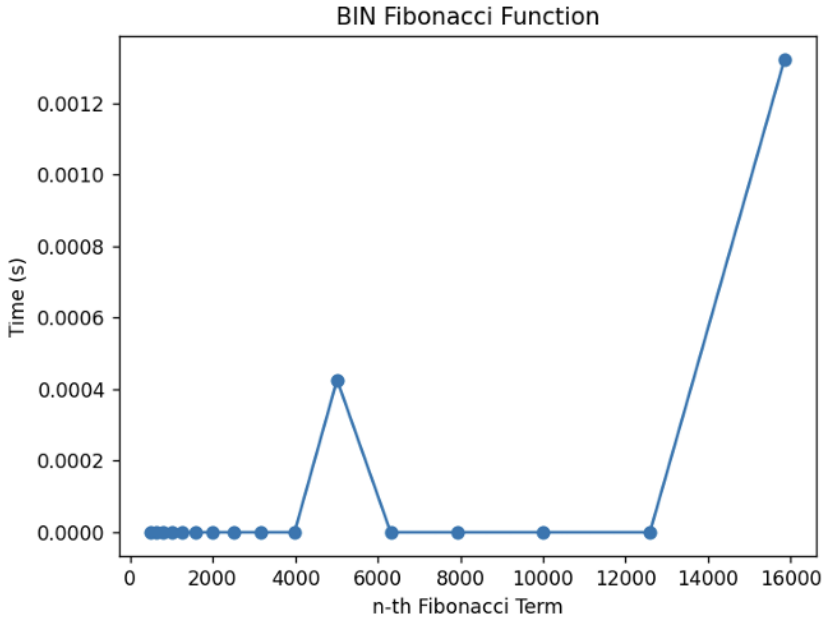

Figure 15: Fibonacci Binet formula Method

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.

Fast Doubling Method:

The Fast Doubling method is a highly optimized version of the Matrix Power approach. Instead of performing full matrix multiplications, it uses algebraic identities to jump through the sequence. This method is the "optimization" referred to in earlier sections that reduces linear complexity to logarithmic complexity while maintaining 100% integer precision.

Algorithm Description:

The method relies on two primary identities:

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

This relationship can be represented elegantly in matrix form. By squaring the transition matrix associated with the k-th Fibonacci pair, we derive the doubling identities:

$$\begin{pmatrix} F_{2k-1} & F_{2k} \\ F_{2k} & F_{2k+1} \end{pmatrix} = \begin{pmatrix} F_{k-1} & F_k \\ F_k & F_{k+1} \end{pmatrix}^2$$

By performing the matrix squaring operation:

$$\begin{pmatrix} F_{k-1} & F_k \\ F_k & F_{k+1} \end{pmatrix} \begin{pmatrix} F_{k-1} & F_k \\ F_k & F_{k+1} \end{pmatrix} = \begin{pmatrix} F_{k-1}^2 + F_k^2 & F_k(F_{k-1} + F_{k+1}) \\ F_k(F_{k-1} + F_{k+1}) & F_k^2 + F_{k+1}^2 \end{pmatrix}$$

Using the property $F_{k-1} = F_{k+1} - F_k$, the term $F_{k-1} + F_{k+1}$ simplifies to $(F_{k+1} - F_k) + F_{k+1}$, which equals $2F_{k+1} - F_k$. This confirms the identities used in the Fast Doubling algorithm.

By using the binary representation of n, we can determine the n-th term in $O(\log n)$ steps. The pseudocode is as follows:

Listing 5: Fast Doubling Method (pseudocode)

```
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 in Python utilizes a recursive helper function to return the pair (F_n, F_{n+1}) to avoid redundant calculations:

```python
def fibonacci_fast_doubling(n):
    if n == 0:
        return (0, 1)
    else:
        a, b = fibonacci_fast_doubling(n >> 1)
        c = a * (2 * b - a)
        d = a * a + b * b
        if n % 2 == 0:
            return (c, d)
        else:
            return (d, c + d)

def fast_doubling(n):
    return fibonacci_fast_doubling(n)[0]
```

Figure 16: Fast Doubling Implementation in Python

Results: The Fast Doubling method provided the most impressive results among the exact methods. As shown in the table and graph below, its execution time is nearly constant for the tested range, outperforming the standard Matrix and DP methods significantly.

n	Repetition 1	Repetition 2	Repetition 3	Average
501	0.00000760	0.00000440	0.00000320	0.00000507
631	0.00000440	0.00000320	0.00000270	0.00000343
794	0.00000420	0.00000330	0.00000270	0.00000340
1000	0.00000390	0.00000290	0.00000270	0.00000317
1259	0.00000420	0.00000370	0.00000320	0.00000370
1585	0.00000700	0.00000380	0.00000350	0.00000477
1995	0.00000560	0.00000430	0.00000390	0.00000460
2512	0.00000590	0.00000500	0.00000480	0.00000523
3162	0.00000730	0.00000600	0.00000570	0.00000633
3981	0.00001150	0.00000860	0.00000780	0.00000930
5012	0.00001330	0.00001080	0.00001030	0.00001147
6310	0.00001620	0.00001580	0.00001440	0.00001547
7943	0.00002340	0.00002040	0.00002000	0.00002127
10000	0.00003140	0.00003080	0.00002990	0.00003070
12589	0.00004030	0.00003930	0.00003890	0.00003950
15849	0.00006430	0.00006110	0.00005990	0.00006177

Figure 17: Fast Doubling Method Results (Horizontal Format)

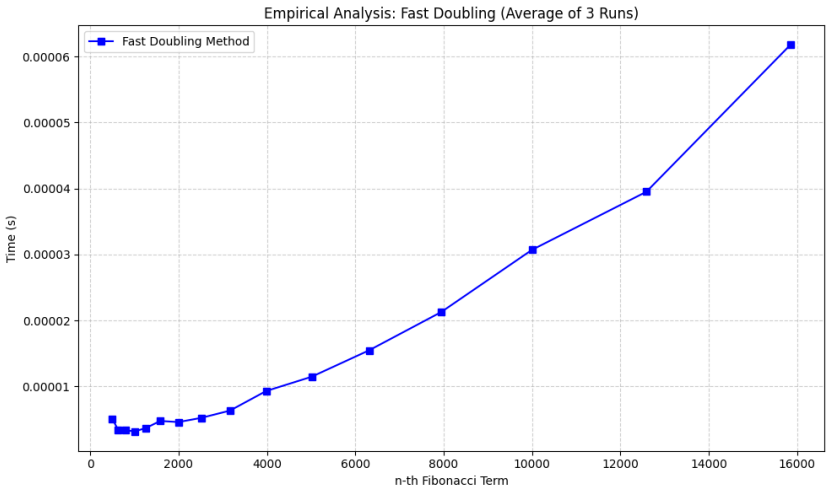

Figure 18: Fast Doubling Method Empirical Graph

The "jagged" nature of the graph at this scale is attributed to system noise and CPU clock precision, as the algorithm's actual logic is faster than the minimum measurable time increment on the testing device.

CONCLUSION

Through Empirical Analysis, within this paper, five distinct classes of methods have been tested and evaluated based on their efficiency, accuracy, and execution time complexity. This analysis allowed for a clear delimitation of the operational scopes for each algorithm, identifying where each is most effective and where its limitations begin.

The Recursive method, while being the most intuitive to implement, proved to be the most computationally expensive due to its exponential time complexity, $T(2^n)$. Experimental data confirms it is only suitable for very small orders, typically up to the 30th-40th term, before the redundant recursive calls cause significant strain on the computing system and unacceptable execution delays.

The Binet method represents the opposite extreme, offering an almost constant execution time. However, its reliance on floating-point arithmetic and the Golden Ratio (ϕ) introduces significant rounding errors. As observed, these inaccuracies make the method unreliable for terms beyond $n = 70$ to 80. Thus, while fast, it is not a viable candidate for high-precision applications without advanced modifications.

The Dynamic Programming and Matrix Multiplication methods provided a balanced middle ground, yielding exact results with linear $T(n)$ complexity in their standard implementations. These methods are robust and capable of handling much larger terms than the recursive approach.

Finally, the inclusion of the **Fast Doubling Method** served as the practical implementation of the theoretical "logarithmic optimization" mentioned earlier. By utilizing specific algebraic identities, this method achieved $O(\log n)$ time complexity while maintaining 100% integer precision. The empirical results demonstrate that Fast Doubling is the most efficient and reliable algorithm among those tested, providing the best performance for large-scale Fibonacci calculations.

REFERENCES

[1] Moisei, D. GitHub. https://github.com/Moisei-D/Algorithm-Analysis