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-242 Bogdan Arama

Verified:
asist. univ. Fiștic Cristofor

Chișinău – 2026

Summary

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: The purpose of the analysis is established. Choose the efficiency metric to be used (number of executions or execution time). Establish the properties of the input data. The algorithm is implemented in a programming language. Generating multiple sets of input data. Run the program for each input data set. The obtained data are analyzed.

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... Mathematically we can describe this as: $x_n = x_{n-1} + x_{n-2}$.

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*, introducing the sequence to the Western world. Traditionally, the sequence was determined just by adding predecessors, however, with the evolution of computer science, several distinct methods for determination have been uncovered.

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 algorithms will be implemented in their naïve form in python and analyzed empirically based on the time required for their completion. 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 its 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 its execution time.

Algorithm Description

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

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Figure 1 Recursive Code

5	7	10	12	15	17	20	22	25	27	30	32	35	37	40	42	45
0.0000	0.0000	0.0000	0.0000	0.0001	0.0002	0.0010	0.0024	0.0101	0.0258	0.1158	0.2833	1.2007	3.0850	13.3579	35.8268	161.1964

Figure 2 Recursive Time Output

Method Analysis 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 may notice that the only function whose time was growing significantly was the Recursive Method. As seen in the generated graph,

the spike in time complexity happens after the 42nd term, leading us to deduce that the Time Complexity is exponential, $T(2^n)$.

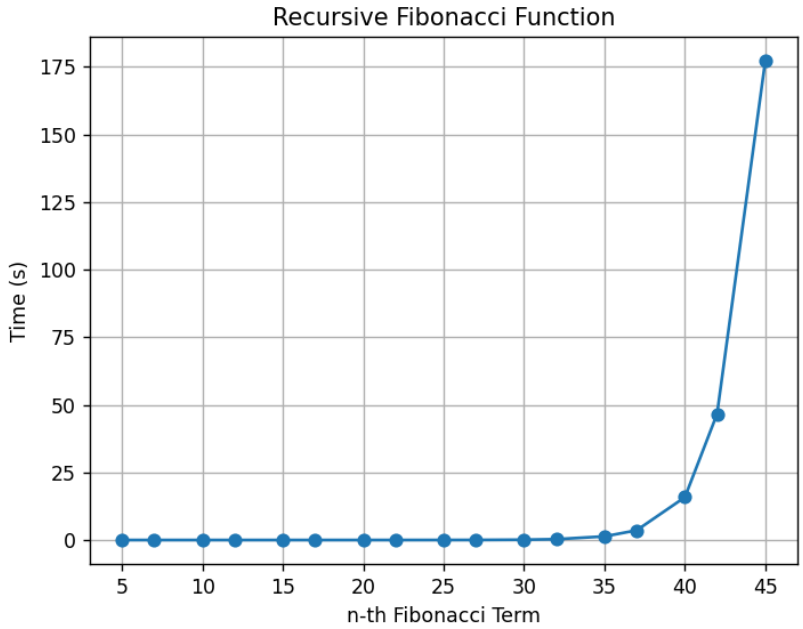

Figure 3 Recursive Graph

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

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

```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 4 Dynamic Programming Code

501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
0.0001	0.0001	0.0001	0.0001	0.0001	0.0002	0.0002	0.0004	0.0004	0.0006	0.0008	0.0015	0.0021	0.0052	0.0072	0.0107

Figure 5 Dynamic Programming Time Output

Method Analysis Results

After the execution of the function for each n Fibonacci term mentioned in the second set of Input Format, we obtained excellent results. The Dynamic Programming Method shows a time complexity denoted in a corresponding graph of $T(n)$, confirming its linear efficiency for large input sets.

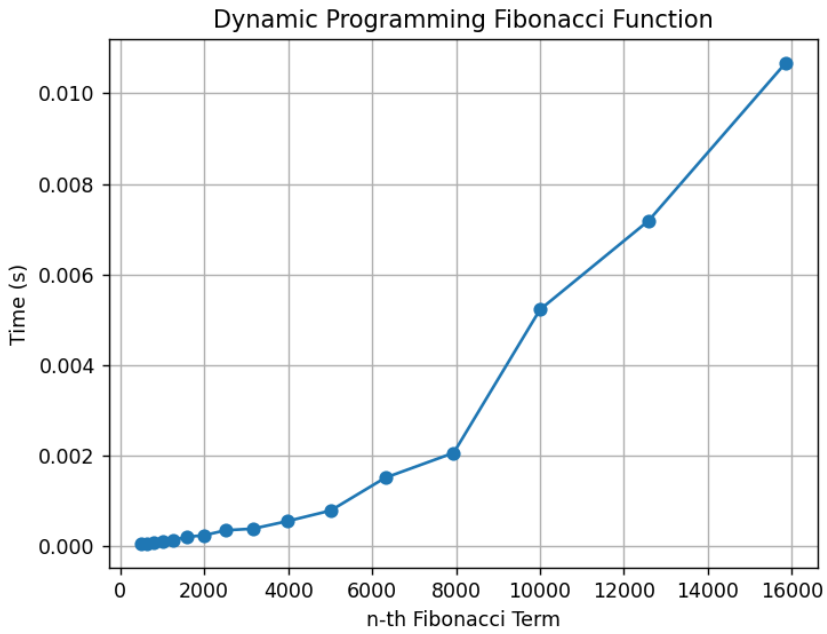

Figure 6 Dynamic Programming Graph

Matrix Power Method

The Matrix Power method of determining the n-th Fibonacci number is based on the multiple multiplication of a naïve Matrix with itself. This property of Matrix multiplication can be used to represent the growth of Fibonacci terms linearly through exponentiation.

Algorithm Description

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

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

Figure 7 Matrix Power Code Part 1

```python
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

def power(F, n):
    M = [[1, 1], [1, 0]]
    for i in range(2, n + 1):
        multiply(F, M)
```

Figure 8 Matrix Power Code Part 2

501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
0.0003	0.0004	0.0006	0.0007	0.0009	0.0012	0.0023	0.0021	0.0028	0.0041	0.0054	0.0080	0.0120	0.0175	0.0252	0.0362

Figure 9 Matrix Power Time Output

Method Analysis Results

After execution, the naïve Matrix method, although slightly slower than the Dynamic Programming one due to the overhead of matrix multiplication functions, still performs very well. The graph indicates a solid linear $T(n)$ time complexity.

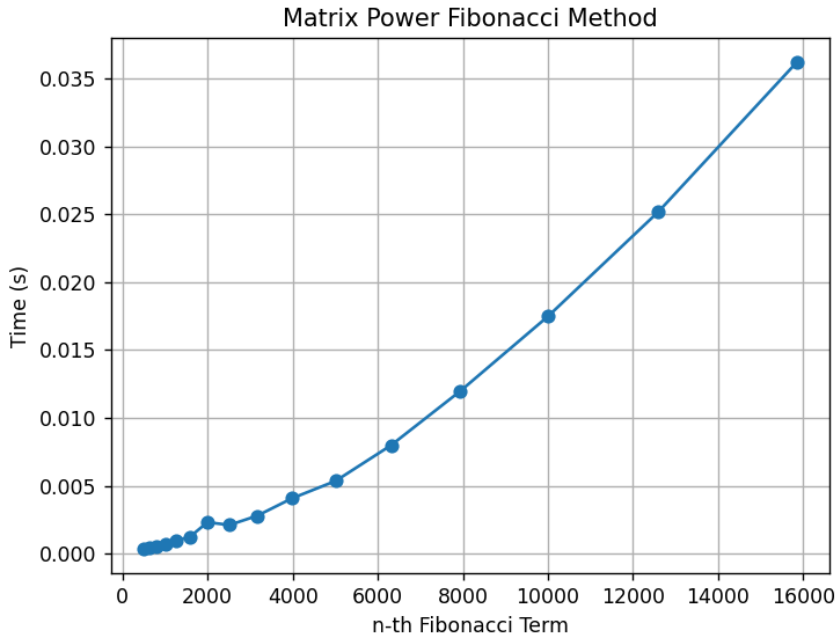

Figure 10 Matrix Power Graph

Binet Formula Method

The Binet Formula Method operates using the Golden Ratio formula, or phi. However, due to its nature of requiring decimal numbers, the rounding error of python that accumulates begins affecting the results significantly. The error starts appearing around the 70-th number, making it unusable in practice despite its speed.

Algorithm Description

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

```python
def fib(n):
    getcontext().prec = 60
    getcontext().rounding = ROUND_HALF_EVEN

    sqrt5 = Decimal(5).sqrt()
    phi = (Decimal(1) + sqrt5) / Decimal(2)
    psi = (Decimal(1) - sqrt5) / Decimal(2)

    value = (phi ** n - psi ** n) / sqrt5
    return int(value.to_integral_value())
```

Figure 11 Binet Formula Code

501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
0.0001	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0001	0.0001	0.0001	0.0002

Figure 12 Binet Formula Time Output

Method Analysis Results

Although it is the most performant regarding execution time, the Binet Formula is not accurate enough to be considered within the analyzed limits for large terms. Its time complexity is almost constant, but accuracy issues mean it is only recommended for Fibonacci terms up to 80 in its naïve form.

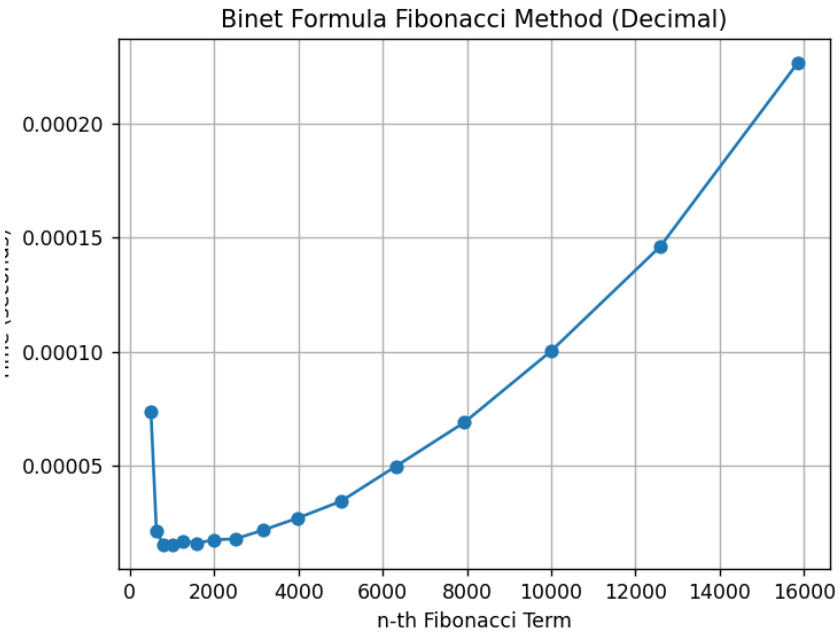

Figure 13 Binet Formula Graph

Fast Doubling Method

The Fast Doubling method is an optimization of the Matrix Power method that reduces the time complexity to logarithmic $O(\log n)$. By using specific jumping formulas based on the binary representation of n, it bypasses full iterative or matrix multiplication steps.

Algorithm Description

```
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 odd:
        return (d, c + d)
    else:
        return (c, d)
```

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

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

Figure 14 Fast Doubling Code

501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0001

Figure 15 Fast Doubling Time Output

Method Analysis Results

The Fast Doubling method represents the peak of optimization for this analysis. The empirical results show that it handles the largest input terms with almost zero noticeable increase in time, confirming its $T(\log n)$ efficiency.

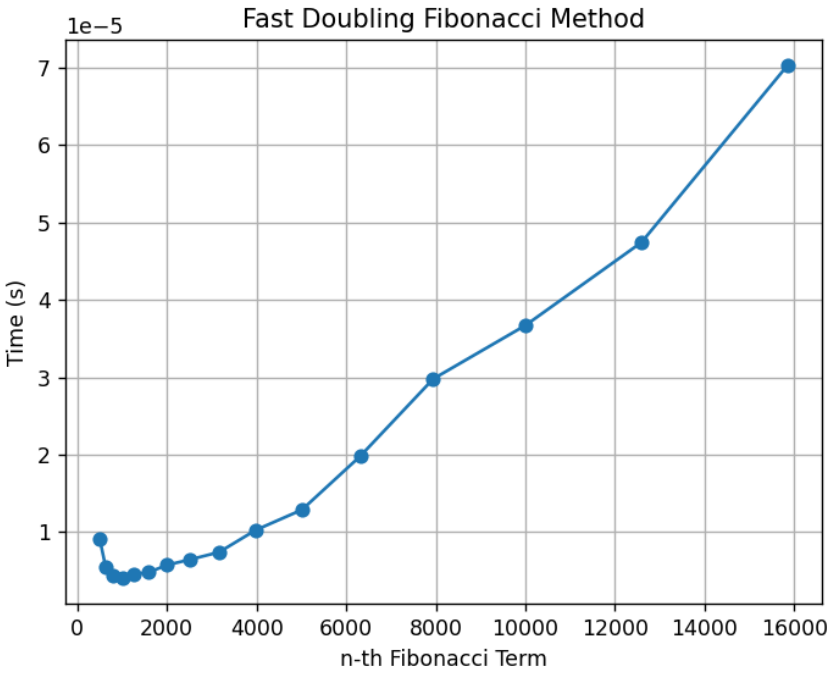

Figure 16 Fast Doubling Graph

CONCLUSION

The empirical analysis conducted in this laboratory work provided a comprehensive evaluation of five distinct algorithmic approaches to determining the n-th term of the Fibonacci sequence. By testing these methods against varying scopes of input, we successfully mapped their theoretical complexities to real-world execution times, highlighting the critical importance of algorithm selection in computational efficiency.

The Recursive method, while mathematically elegant and simple to implement, proved to be highly unfeasible for any $n > 40$ due to its exponential time complexity, $O(2^n)$. The redundant calculations inherent in its structure led to a rapid exhaustion of computational resources. In stark contrast, the Dynamic Programming and Matrix Multiplication methods demonstrated that shifting from a top-down to a bottom-up or algebraic approach can drastically reduce execution time to linear growth, $O(n)$. These methods provide the best balance of simplicity and accuracy for general-purpose use.

Furthermore, the Binet Formula method showcased the power of closed-form mathematical solutions, offering nearly constant-time execution, $O(1)$. However, the empirical results also revealed its primary weakness: the reliance on floating-point arithmetic leads to precision loss and rounding errors after the 70th-80th term in standard Python implementations. This makes it a specialized tool rather than a general solution.

Finally, the Fast Doubling method emerged as the most technologically superior approach. By optimizing the matrix representation to a logarithmic complexity, $O(\log n)$, it exhibited near-flat execution times even for the largest values in the test series. In conclusion, for practical applications at the Technical University of Moldova or in professional software development, the Dynamic Programming method is recommended for its reliability and precision, while the Fast Doubling method is the optimal choice for high-performance systems requiring the calculation of extremely large Fibonacci terms.

Access the full implementation and analysis on GitHub:
https://github.com/bogdan456tech/aa-labs/tree/main/lab1