MINISTRY OF EDUCATION AND RESEARCH OF THE REPUBLIC OF MOLDOVA
Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics
Department of Software and Automation Engineering

RUSNAC NICHITA FAF-242

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

of the "Algorithm analysis" course

Checked:
asist. univ. Cristofor Fistic
Department of Software and Automation Engineering,
FCIM Faculty, UTM

Chisinau – 2026

Оглавление

ALGORITHM ANALYSIS

Objective

Study and analyze multiple algorithms for computing the Fibonacci n-th term, including recursive, dynamic programming, closed-form, matrix-based, and advanced recursive-formula algorithms.

This report extends classical methods by including ten well-known algorithms and analyzing their theoretical and experimental behavior.

Tasks

1. Implement at least 10 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 theoretical complexity analysis is empirical analysis.
This approach is useful for:

- gaining initial insight into the complexity class of an algorithm;
- comparing the performance of two or more algorithms that solve the same problem;
- comparing different implementations of the same algorithm;
- evaluating how efficiently an algorithm runs on a specific computer system.

When performing empirical analysis, the following steps are typically followed:

1. Define the objective of the analysis.
2. Select the efficiency metric (for example, number of executed operations or execution time of the whole algorithm or a part of it).
3. Define the input data characteristics relevant to the analysis (such as data size or specific input properties).
4. Implement the algorithm in a programming language.
5. Generate multiple input data sets.
6. Execute the program for each input set.
7. Analyze the collected results.

The choice of efficiency metric depends on the goal of the analysis. For instance, if the objective is to estimate the complexity class or verify a theoretical prediction, counting the number of operations is usually appropriate. However, if the goal is to evaluate how well an implementation performs in practice, measuring execution time is more suitable.

After running the program with test data, the results are recorded. Then, depending on the analysis goals, either statistical summaries (such as mean, standard deviation, etc.) are computed, or graphs are plotted using pairs of values (for example, problem size versus efficiency metric).

Introduction

The Fibonacci sequence is defined by:

$F_n=F_{n-1}+F_{n-2}$
with base cases:
$F_0 = 0, F_1 = 1$

This sequence is widely used in algorithm education because it demonstrates multiple algorithmic paradigms such as:

- recursion vs iteration
- memoization
- dynamic programming
- matrix exponentiation
- closed-form formulas
- complexity growth patterns

The report presents ten algorithms, illustrating algorithmic concepts including exponential vs polynomial complexity, exact vs approximate computation, and repeated squaring optimization.

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, some algorithms will receive a series of numbers that will contain the order of the Fibonacci terms being looked up. The first series are small numbers, (5, 7, 10, 12, 15, 18, 20, 22, 25, 30), is small enough to run all algorithms, the second series will have a bigger scope to be able to compare the other algorithms between themselves (32, 35, 38, 40, 45, 50, 55, 60, 65, 70), the third series excludes the approximation algorithms (80, 120, 200, 300, 400, 500, 600, 700, 800, 900), and the last series focuses on the fastest algorithms only, excluding the slow algorithms, the recursive algorithms, and the approximate algorithms (1000, 1500, 2000, 3000, 4000, 5000, 6000, 7000, 8500, 10000).

IMPLEMENTATION

All ten 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 of memory of the device used.

Top-down Dynamic Programming (F1)

F1 is derived directly from the recursive definition of the Fibonacci sequence: $F_n=F_{n-1}+F_{n-2}$, $n \geq 2$, with the base values $F_0 = 0$, $F_1 = 1$.

Its time complexity T(n) with constant-time arithmetic can be expressed as $T(n) = T(n-1) + T(n-2) + 1$, $n \geq 2$, with the base values $T(0) = 1$ and $T(1) = 1$. The solution of this linear non-homogeneous recurrence relation implies that the time complexity is equal to $T(n) = O(F_n) = O(\phi^n)$, which is exponential in n.

Implementation:

```
#Top-down Dynamic Programming
def F1(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return F1(n-1) + F1(n-2)
```

Figure 1 - F1

Top-down Dynamic Programming with Memorization (F2)

F2 is equivalent to F1 with the main change being the so-called memorization. Memorization allows the caching of the already computed Fibonacci numbers so that F2 does not have to revisit already visited parts of the call tree. Memorization reduces the time and space complexities drastically; it leads to at most n additions. With constant-time arithmetic, the time complexity is O(n). The space complexity is also linear.

Implementation:

```
#Top-down Dynamic Programming with Memoization
def F2(n, F):
    if F[n] == None:
        if n == 0:
            F[n] = 0
        elif n == 1:
            F[n] = 1
        else:
            F[n] = F2(n-1, F) + F2(n-2, F)
    return F[n]
```

Figure 2 – F2

Iteration with Constant Storage (F3)

F3 uses the fact that each Fibonacci number depends only on the preceding two numbers in the Fibonacci sequence. This fact turns an algorithm designed top down, namely, F2, to one designed bottom up. This fact also reduces the space usage to constant, just a few variables. The time complexity is exactly the same as that of F2. The space complexity is O(1) with constant-time arithmetic and O(n) with non-constanttime arithmetic. Regarding the algorithmic concepts, F3 illustrates iteration, recursion to iteration conversion, bottom-up dynamic programming, and constant space.

Implementation:

```
#  Iteration with Constant Storage
def F3(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        a, b = 0, 1
        for _ in range(2, n+1):
            a, b = b, a + b
        return b
```

Figure 3 – F3

Closed-form Formula with the Golden Ratio (F4)

F4 uses the fact that the nth Fibonacci number has the following closedform formula(Binet):

$$F_n = \left[\frac{\varphi^n - \psi^n}{\varphi - \psi}\right] = \left[\frac{\varphi^n - \psi^n}{\sqrt{5}}\right]$$

This algorithm performs all its functions in floating-point arithmetic; the math functions in the algorithm map to machine instructions directly. Thus, this algorithm runs in constant time and space. For n > 71, this algorithm starts returning approximate results and the first value differs by 1, for n = 100 algorithm is wrong by 1196093, this is because of floating point precision and error accumulation(Numerical Analysis vibes).

Implementation:

```
#Binet's Formula
import math
def F4(n):
    phi = (1 + math.sqrt(5)) / 2
    psi = (1 - math.sqrt(5)) / 2
```

```
    return round((phi**n - psi**n) / math.sqrt(5))
```

Figure 4 – F4

Closed-form Formula with the Golden Ratio and Rounding (F5)

F5 relies on the same closed-form formula used by F4. However, it also uses the fact that ψ is less than 1, meaning its nth power for large n approaches zero, so Binet's formula reduces to:

$$F_n = \left[\frac{\varphi^n}{\sqrt{5}}\right]$$

For the same reasons as in F4, this algorithm also runs in constant time and space. The approximate behaviour of this algorithm is the same as that of F4. Regarding the algorithmic concepts, F5 illustrates the concept of the optimization of a closed-form formula for speed-up in addition to the algorithmic concepts illustrated by F4.

This algorithm gives wrong result from the start, but the error is not so big.

Implementation:

```
#optimized Binet's Formula
import math
def F5(n):
    sqrt5 = math.sqrt(5)
    phi = (1 + sqrt5) / 2
    return round((phi**n) / sqrt5)
```

Figure 5 – F5

The Power of a Certain 2x2 Matrix via Iteration (F6)

F6 uses the power of a certain 2x2 matrix for the Fibonacci sequence:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}$$

The complexity analyses here are similar to those of F2 or F3. With constant-time arithmetic, the time complexity is O(n). The space complexity in this case is also linear. Regarding the algorithmic concepts, F6 illustrates (simple) matrix algebra and iteration over closed-form equations.

```
#matrix exponentiation
def mat_mul_opt(m1):
    m = [[0,0],[0,0]]
    m[0][0] = m1[0][0] + m1[0][1]
    m[0][1] = m1[0][0]
```

```
    m[1][0] = m1[1][0] + m1[1][1]
    m[1][1] = m1[1][0]
    return m

def F6(n):
    if n == 0:
        r = 0
    else:
        m = [[1,0],[0,1]]  #identity matrix
        for _ in range(1, n):
            m = mat_mul_opt(m)
        r = m[0][0]
    return r
```

Figure 6 – F6

The Power of a Certain 2x2 Matrix via Repeated Squaring (F7 and F8)

F7 and F8 use the same equations used in F6 but while F6 uses iteration for exponentiation, F7 and F8 uses repeated squaring for speed up. Moreover, F7 uses a recursive version of repeated squaring while F8 uses an iterative version of it. Repeated squaring reduces time from linear to logarithmic. Hence, with constant time arithmetic, the time complexity is O(lg n). The space complexity is also logarithmic in n.

Implementation:

```
# F7
def mat_mul(m, n):
    r = [[0,0],[0,0]]
    r[0][0] = m[0][0]*n[0][0] + m[0][1]*n[1][0]
    r[0][1] = m[0][0]*n[0][1] + m[0][1]*n[1][1]
    r[1][0] = m[1][0]*n[0][0] + m[1][1]*n[1][0]
    r[1][1] = m[1][0]*n[0][1] + m[1][1]*n[1][1]
    return r

def mat_pow_recur(m, n):
    r = [[1,0],[0,1]]  #identity matrix
    if n>1:
        r = mat_pow_recur(m, n >> 1) # n//2
        r = mat_mul(r, r)
    if n % 2 == 1:
        r = mat_mul(r, m)
    return r

def F7(n):
    if n == 0:
        r = 0
    else:
        m = mat_pow_recur([[1,1],[1,0]], n-1)
```

```
        r = m[0][0]
    return r
```

Figure 7 – F7

And the F8 which is iterative

Implementation:

```
#Iteration with Constant Storage
def mat_pow_iter(m, n):
    n_bits = bin(n)[2:] #binary representation of n and exclude '0b' prefix
    r = [[1,0],[0,1]]  #identity matrix
    for bit in n_bits:
        r = mat_mul(r, r)
        if bit == '1':
            r = mat_mul(r, m)
    return r

def F8(n):
    if n == 0:
        return 0
    else:
        m = mat_pow_iter([[1,1],[1,0]], n-1)
        return m[0][0]
```

Figure 8 – F8

A Certain Recursive Formula (F9 and F10)

Both F9 and F10 use the following formulas for the Fibonacci sequence:

$$F_{2n+1} = {F_{n+1}}^2 + {F_n}^2 \; and \; F_{2n} = 2F_{n+1}F_n - {F_n}^2$$

where n ≥ 2, with the base values $F_2 = 1$, $F_1 = 1$, and $F_0 = 0$. Note that memoization is used for speed-up in such a way that only the cells needed for the final result are filled in the memoization table F. In F9, recursion takes care of identifying such cells. In F10, a cell marking phase, using F as a queue data structure, is used for such identification; then the values for these cells, starting from the base values, are computed in a bottom-up fashion. These algorithms behave like repeated squaring in terms of time complexity. Hence, with constant time arithmetic, the time complexity is O(lg n). The space complexity is also logarithmic in n.

Implementation:

```
# Squaring and recursive formula
def F9(n, F):
    if F[n] == None:
        if n == 0:
```

```
            F[n] = 0
        elif n == 1:
            F[n] = 1
        elif n == 2:
            F[n] = 1
        else:
            k = n >> 1
            f1 = F9(k, F)
            f2 = F9(k+1, F)
            if n % 2 == 0:
                F[n] = 2 * f1 * f2 - f1 * f1
            else:
                F[n] = f2 * f2 + f1 * f1
    return F[n]
```

Figure 9 – F9

And the second algorithm which is iterative

Implementation:

```
#iterative version of F9
def F10(n):
    F = {}

    # find indexes that need F values
    qinx = []  # queue of indexes
    qinx.append(n)
    F[n] = -1  # -1 to mark such values
    while qinx:
        k = qinx.pop() >> 1
        if k not in F:
            F[k] = -1
            qinx.append(k)
        if (k + 1) not in F:
            F[k + 1] = -1
            qinx.append(k + 1)

    # set base values
    F[0], F[1], F[2] = 0, 1, 1

    # fill the indexes that need values
    keys_sorted = sorted(F.keys())
    for k in keys_sorted[3:]:
        k2 = k >> 1
        f1, f2 = F[k2], F[k2 + 1]
        if k % 2 == 0:
            F[k] = 2 * f2 * f1 - f1 * f1
        else:
            F[k] = f2 * f2 + f1 * f1
```

```
    r = F[n]
    return r
```

Figure 10 – F10

Overall summary of complexity:

Algorithm	Type	Time Complexity
F1	Recursive	$O(\varphi^n)$
F2	Memorization	O(n)
F3	Iterative DP	O(n)
F4	Closed Form	O(1)
F5	Closed Form	O(1)
F6	Matrix Iteration	O(n)
F7	Matrix Fast Power	O(log n)
F8	Matrix Fast Power	O(log n)
F9	Doubling Recursive	O(log n)
F10	Doubling Iterative	O(log n)

RESULTS

Hope Mr. Fistic wouldn't downgrade me but I need to change a little the structure. I decided that it would be better to compare and plot all algorithms one against another(and perform like a battle royale, There Can Be Only One) and print some data just in case. So I have 10 algorithms and the first set, let's plot them.

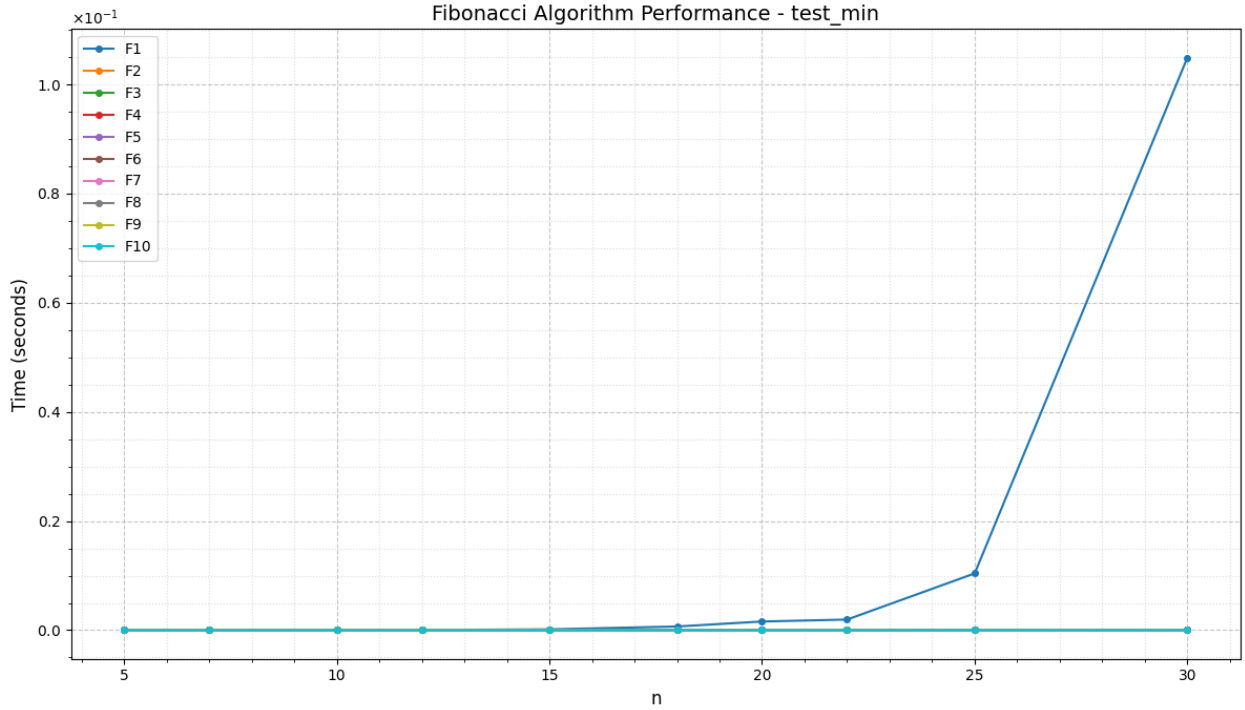

Figure 11 – results of the algorithm on minimal test array

As we can see they all performed great with exception of F1 which was the formula which just came from the definition of Fibonacci so nothing strange. Let's delete F1 from list and go to the second set of values.

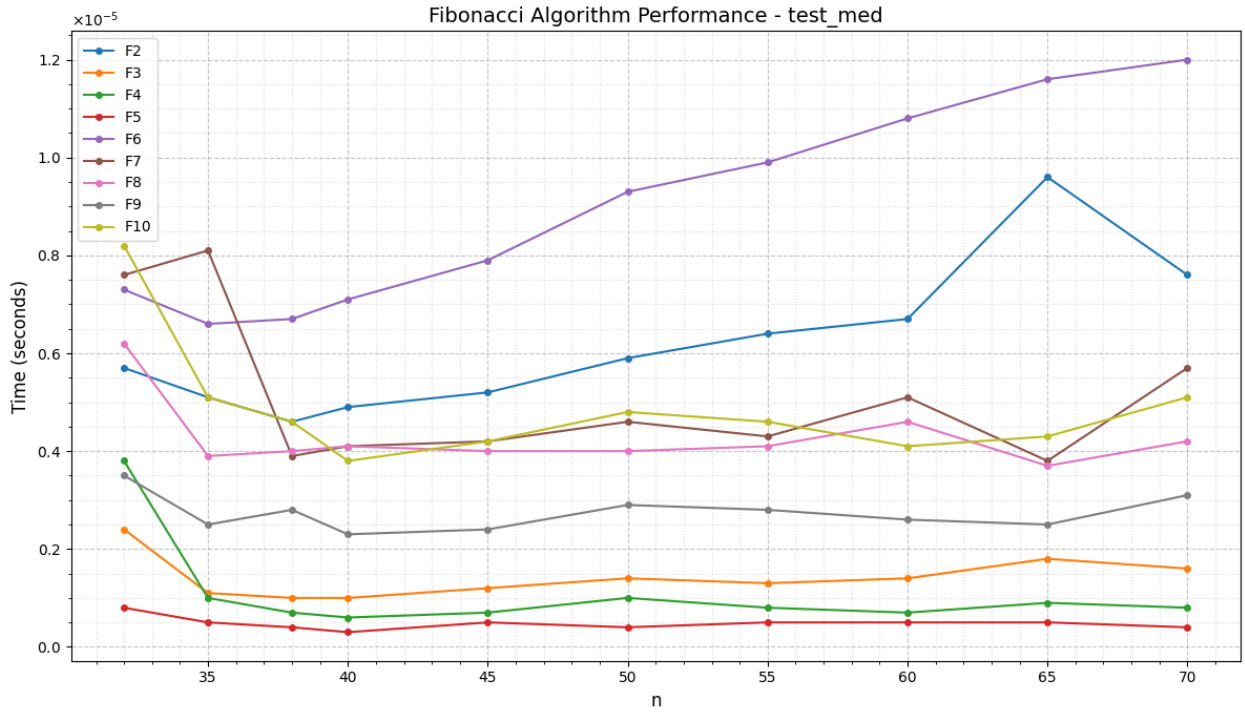

Figure 12 – results of the algorithm on medium test array

Here the results are very interesting! A clear sort of the best can be seen but none is performing a lot worse than others so let's go to the third set with same algorithms. You may notice that indeed the best are F4 and F5 because they are just an approximate computation, even though F4 is still giving exactly right result(F4 is not giving a right answer almost from the start) so let's leave them for a while.

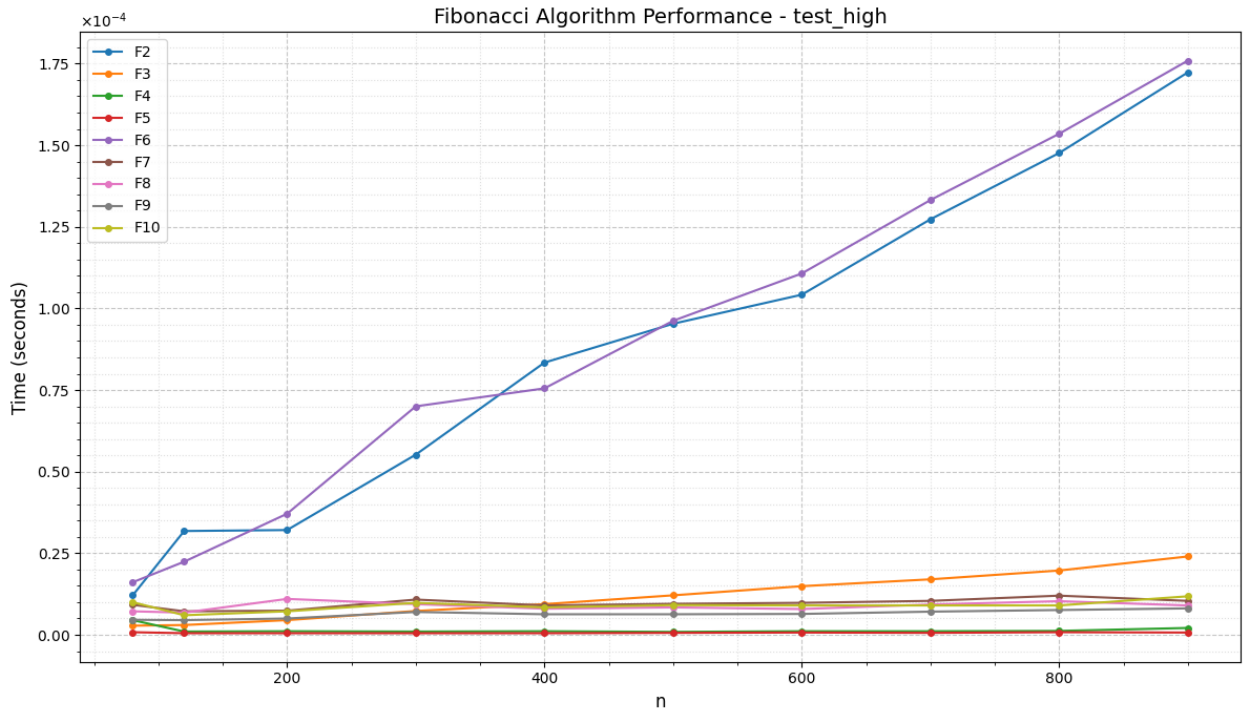

Figure 13 – results of the algorithm on high test array

Here the result are clear, algorithms F2 and F6 are worse because they are O(n) but why F3 is doing better even it has same complexity? Because it only does simple integer additions with no function calls or object creation, making it much faster in practice. So delete F2 and F6 and let's go to the final boss, but firstly let's delete also F4 and F5 because at this points they give too wrong values and it is not interesting to analyze complexity of a wrong algorithm.

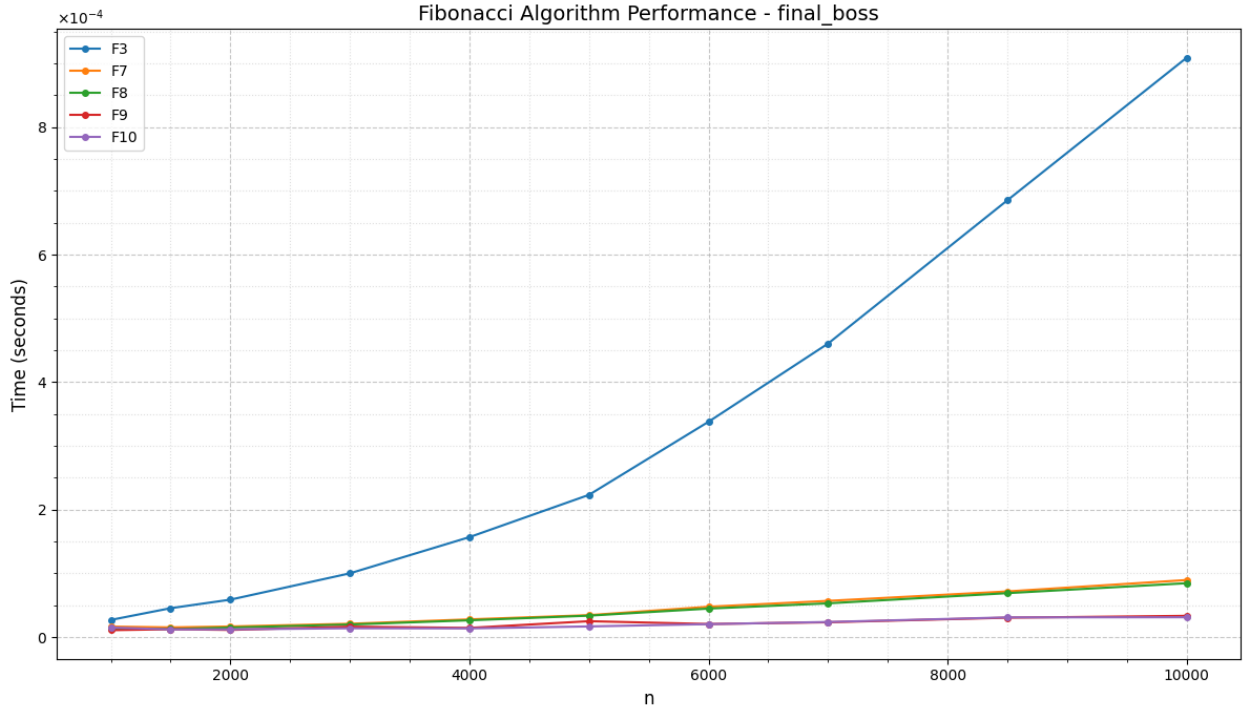

Figure 14 – results of the algorithm on final test array

At this point other algorithms are either O(lg n), the F3 is the last one with O(n) complexity, he did actually a great job, I wasn't expecting this. Now let's leave F3 and try one more time the same set of values but for remaining algorithms.

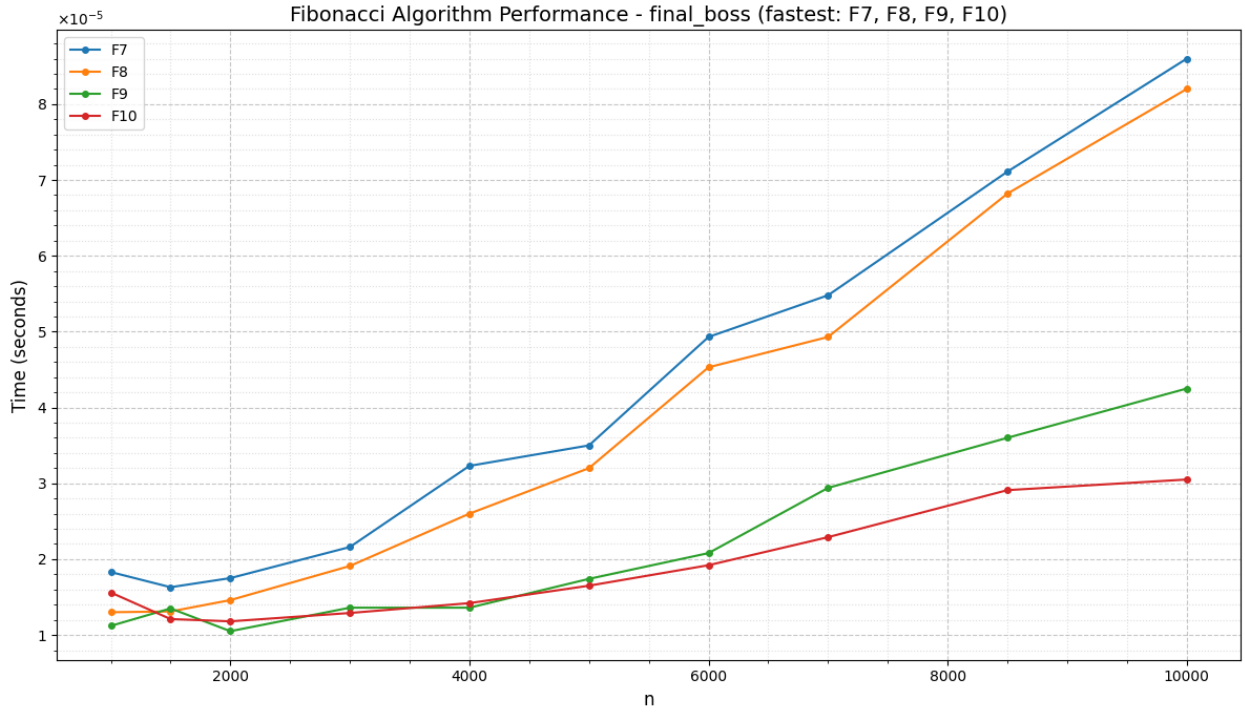

Figure 15 – results of the algorithm on final test array with best algorithms

So we can see clearly that the best algorithm is F10 and F9, why? F9/F10 use doubling formulas (F2k from Fk) achieving O(log n) with only 2-3 multiplications per step. Matrix methods (F7/F8) are also O(log n) but need 8 multiplications per step, making F9/F10 ~3-4x faster.

CONCLUSION

In this laboratory work, ten algorithms for computing the Fibonacci n-th term were studied and analyzed both theoretically and empirically. The algorithms represented multiple computational paradigms including naive recursion, dynamic programming, iterative methods, closed-form formulas, matrix exponentiation, and fast doubling recursive formulas.

The experimental analysis confirmed the theoretical time complexities of the algorithms. The naive recursive algorithm (F1) showed exponential growth and quickly became impractical for larger values of n. Dynamic programming algorithms (F2 and F3) demonstrated linear time complexity, with the iterative version (F3) performing better in practice due to reduced overhead and constant memory usage.

Closed-form algorithms (F4 and F5) demonstrated constant theoretical time complexity and extremely fast execution times. However, due to floating-point precision limitations, these algorithms produced inaccurate results for larger values of n, making them unsuitable for high-precision computations.

Matrix-based algorithms (F6, F7, F8) and fast doubling algorithms (F9, F10) demonstrated the best balance between performance and accuracy. In particular, the fast doubling algorithms achieved logarithmic time complexity while requiring fewer arithmetic operations compared to matrix methods, making them the fastest exact algorithms in practical scenarios.

The experimental results also highlighted that asymptotic complexity alone does not fully determine real-world performance. Implementation details such as memory allocation, function calls, and arithmetic operation cost significantly influence actual runtime.

Overall, fast doubling algorithms (F9 and F10) proved to be the most efficient and scalable exact methods for computing large Fibonacci numbers, while iterative dynamic programming (F3) remains the simplest efficient algorithm for moderate input sizes.

This laboratory work demonstrates how algorithm design choices strongly impact performance and shows the importance of combining theoretical analysis with empirical testing when evaluating algorithm efficiency.

Github repository: https://github.com/Nichita111/AA_labs