MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE AND AUTOMATION ENGINEERING

ANALYSIS OF ALGORITHMS

LABORATORY WORK #1

Development of elaborations through the use of state diagrams and activity diagrams.

Author:

Andrei Chicu

std. gr. FAF–233

Verified:

Fistic

Department of SEA, FCIM UTM

Chișinău, 2024

1 Algorithms Analysis

github url: `https://github.com/andyp1xe1/aa_labs/tree/main/lab1`

1.1 Objective

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

1.2 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.

1.3 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.

1. The properties of the input data in relation to which the analysis is performed are established

(data size or specific properties).

1. The algorithm is implemented in a programming language.
2. Generating multiple sets of input data.

3. Run the program for each input data set.

4. 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.

1.4 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 6 algorithms empirically.

1.5 Comparison Metric

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

1.6 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).

2 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.

2.1 Helper Code

We will be plotting the performance of algorithms using `matplotlib`, by taking the time taken to compute numbers at even inrervals. For that we define a helper function:

```
import time
import matplotlib.pyplot as plt

def plotFibPerformance(fibFunc, numList):
    times = []
    for n in numList:
        start = time.perf_counter()
        fibFunc(n)
        times.append(time.perf_counter() - start)

    plt.figure(figsize=(10, 6))
    plt.plot(numList, times, 'bo-')
    plt.xlabel('n-th Fibonacci Number')
    plt.ylabel('Time (seconds)')
```

```
    plt.title(f'Performance of {fibFunc.__name__}')
    plt.grid(True)

    #return times
    return plt.gcf()
```

And we define our input lists:

```
input1 = [5, 7, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35,]
input2 = [501, 631, 794, 1000, 1259, 1585, 1995, 2512, 3162, 3981, 5012, 6310,
↪  7943,]
```

2.2 Recursive Method

The recursive Fibonacci implementation directly mirrors the mathematical recurrence relation Fn = Fn-1 + Fn-2. For each n, it recursively calculates F(n-1) and F(n-2) until reaching base cases of n=0 or n=1. While elegant, this creates an exponential time complexity of $O(2^n)$ as it recomputes the same Fibonacci numbers many times. For example, computing F(4) requires computing F(3) and F(2), but F(3) also requires computing F(2) again.

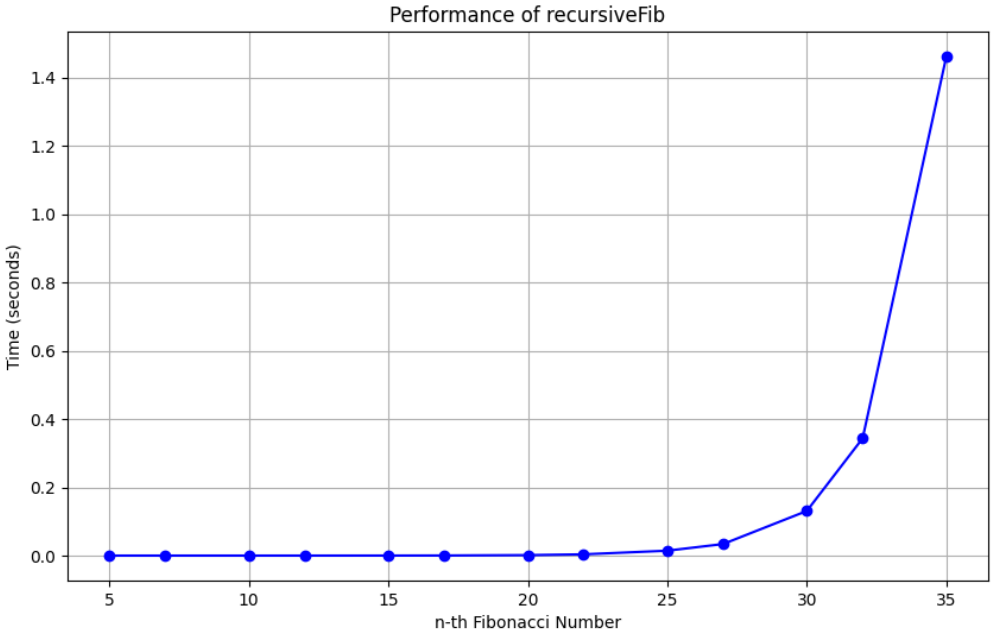
2.2.1 Implementation

```
def recursiveFib(n):
    if n <= 1:  # Base cases
        return n
    return recursiveFib(n-1) + recursiveFib(n-2)
```

2.2.2 Results

Here we are running the function for every number in the list:

```
plotFibPerformance(recursiveFib, input1)
```



2.3 Memoized Recursive

Memoization caches previously computed values to avoid redundant calculations.

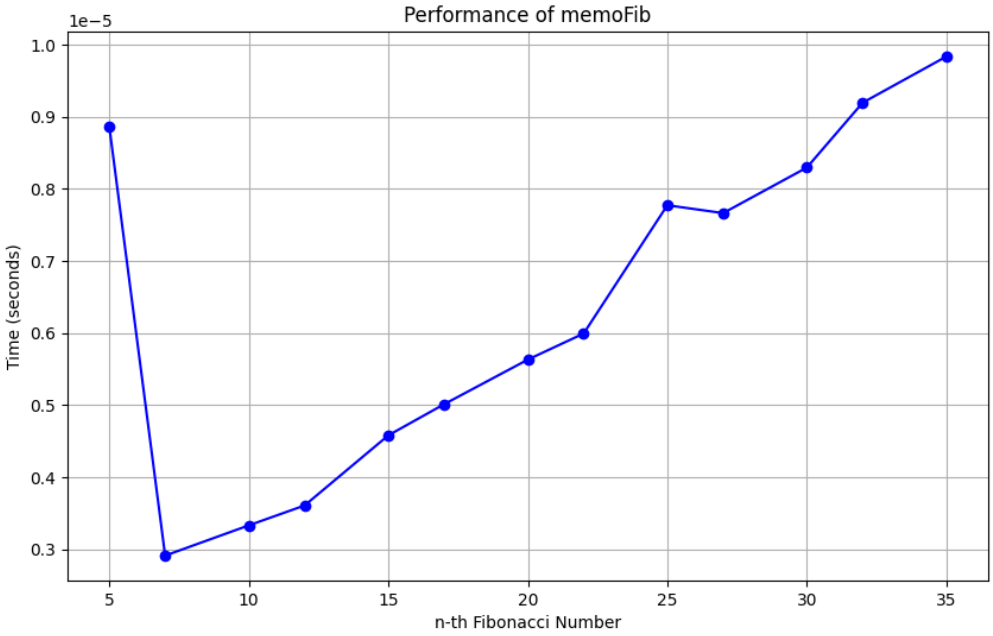
2.3.1 Implementation

```
def memoFib(n, memo=None):
    if memo is None: memo = {}
    if n <= 1: return n
    if n not in memo:
        memo[n] = memoFib(n-1, memo) + memoFib(n-2, memo)
    return memo[n]
```

2.3.2 Results

As expected, we achieve T(n) time complexity:

```
plotFibPerformance(memoFib, input1)
```



2.4 Matrix Exponentiation

The matrix exponentiation method computes Fibonacci numbers in T(log n) time by exploiting the relationship between consecutive Fibonacci numbers and matrix multiplication. The key insight is that:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

Therefore, F_n can be computed by raising the base matrix to the nth power using fast exponentiation.

2.4.1 Implementation

This will allow us to deal with overflows

```
MOD = 10**9 + 7
```

Function to multiply two 2x2 Matrices:

```
def multiply(A, B):
    # Matrix to store the result
    C = [[0, 0], [0, 0]]

    # Matrix Multiply
    C[0][0] = (A[0][0] * B[0][0] + A[0][1] * B[1][0]) % MOD
    C[0][1] = (A[0][0] * B[0][1] + A[0][1] * B[1][1]) % MOD
```

```
    C[1][0] = (A[1][0] * B[0][0] + A[1][1] * B[1][0]) % MOD
    C[1][1] = (A[1][0] * B[0][1] + A[1][1] * B[1][1]) % MOD

    # Copy the result back to the first matrix
    A[0][0] = C[0][0]
    A[0][1] = C[0][1]
    A[1][0] = C[1][0]
    A[1][1] = C[1][1]
```

Function to find (Matrix M^{expo})

```
def power(M, expo):
    # Initialize result with identity matrix
    ans = [[1, 0], [0, 1]]

    # Fast Exponentiation
    while expo:
        if expo & 1:
            multiply(ans, M)
        multiply(M, M)
        expo >>= 1

    return ans
```

And the fibonacci function per-ce:

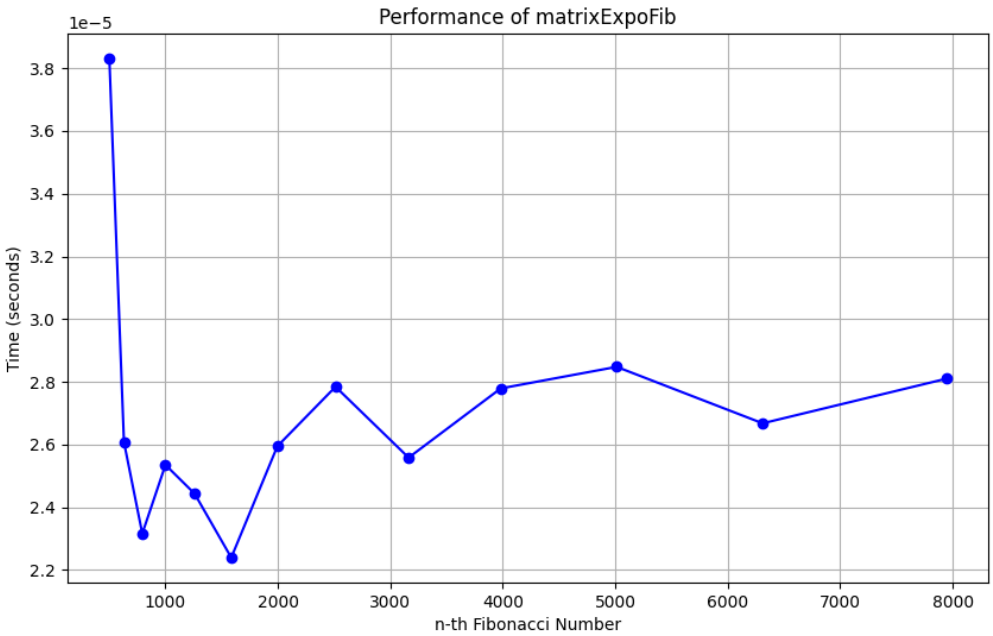
```
def matrixExpoFib(n):
    # Base case
    if n == 0 or n == 1:
        return 1

    M = [[1, 1], [1, 0]]
    # F(0) = 0, F(1) = 1
    F = [[1, 0], [0, 0]]

    # Multiply matrix M (n - 1) times
    res = power(M, n - 1)

    # Multiply Resultant with Matrix F
    multiply(res, F)
```

```
    return res[0][0] % MOD
```

2.4.2 Results

Here we measure the function, Roughly seeing the T(log n) complexity.

```
plotFibPerformance(matrixExpoFib, input2)
```



2.5 Fast Doubling method

The Matrix Exponentiation Method is already discussed before. The Doubling Method can be seen as an improvement to the matrix exponentiation method to find the N-th Fibonacci number although it doesn't use matrix multiplication itself.

The Fibonacci recursive sequence is given by

$$F(n+1) = F(n) + F(n-1)$$

The Matrix Exponentiation method uses the following formula The fast doubling method leverages the relationships between consecutive Fibonacci numbers using these identities:

$$F_{2n} = F_n(2F_{n+1} - F_n) \tag{1}$$

$$F_{2n+1} = F_{n+1}^2 + F_n^2 \tag{2}$$

Like matrix exponentiation, this achieves T(log n) time complexity through recursive doubling.

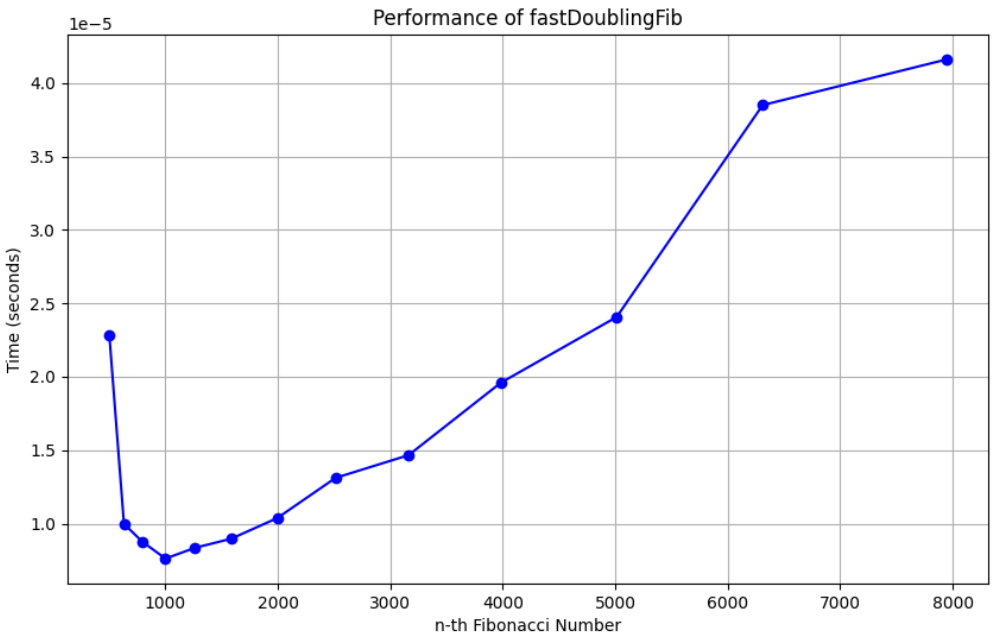
2.5.1 Implementation

```
def fastDoublingFib(n):
    return _fib(n)[0]

# Auxiliary method for Fast Doubling
def _fib(n):
    if n == 0:
        return 0, 1
    else:
        a, b = _fib(n // 2)
        c = a * (b * 2 - a)
        d = a * a + b * b
        if n % 2 == 0:
            return c, d
        else:
            return d, c + d
```

2.5.2 Results

Here we measure the function. As expected, it is much faster than the matrix exponentiation method, whilst still having resemblence of the $log(n)$ graph.

```
plotFibPerformance(fastDoublingFib, input2)
```



2.6 Dinamic Programming

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, it stores just the previous two values. The recurrence relation is:

$$F_n = F_{n-1} + F_{n-2}$$

2.6.1 Implementation

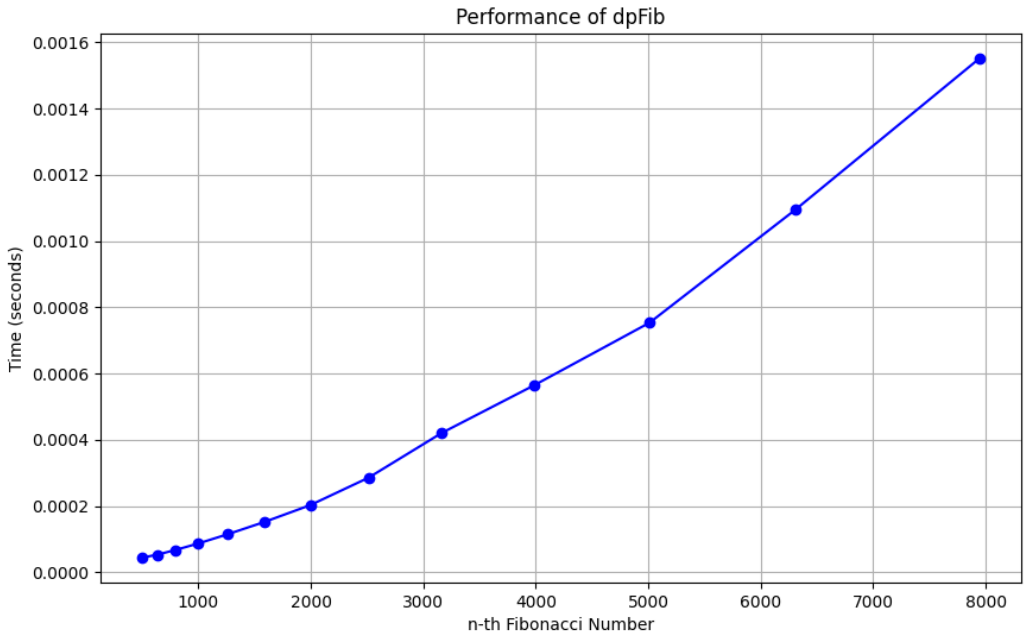
```
def dpFib(n):
    if n <= 1: return n
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b  # Store only previous two values
    return b
```

This is optimal for computing a single Fibonacci number with respect to time complexity.

2.6.2 Results

Showing excellent results with a time complexity denoted in a corresponding graph of T(n):

```
plotFibPerformance(dpFib, input2)
```



2.7 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.

2.7.1 Implementation

```
import math
from decimal import Decimal, Context, ROUND_HALF_EVEN



input3 = [5, 7, 10, 12, 15, 17, 20, 22, 25, 27, 30, 32, 35, 47, 50, 55, 65, 70,
↪  80, 100, 150, 166, 200]

def binetFib(n):
    ctx = Context(prec=60, rounding=ROUND_HALF_EVEN)
    phi = Decimal((1 + Decimal(5**(1/2))))
    psi = Decimal((1 - Decimal(5**(1/2))))
    return int((ctx.power(phi, Decimal(n)) - ctx.power(psi, Decimal(n))) / (2 **
    ↪  n * Decimal(5 * (1/2))))
```

2.7.2 Results

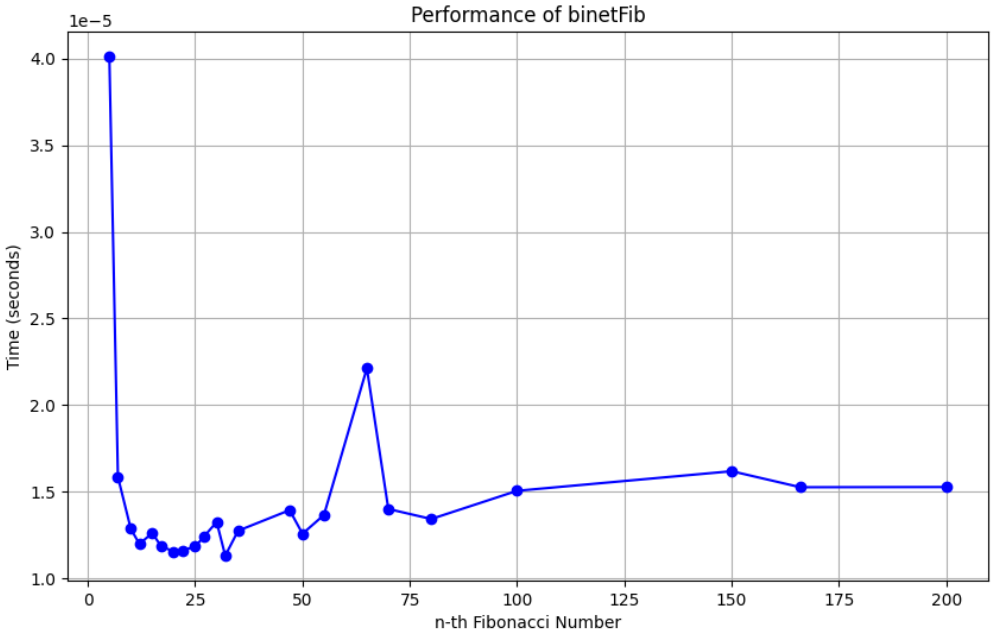
The errors starting with around 75-th number make the algorithm unusable in practice, despite its speed:

```
numBinet = binetFib(75)
numDoubl = fastDoublingFib(75)
err = abs(numBinet - numDoubl)
f"binet:\t\t{numBinet}\nvs fast doubling:\t{numDoubl}\nThe error:\t{err}"
```

```
binet:  1888569667134150
vs fast doubling: 2111485077978050
The error: 222915410843900
```

And here is the performance graph:

```
plotFibPerformance(binetFib, input3)
```



3 Conclusions

In this laboratory, we analyzed various algorithms for calculating Fibonacci numbers, focusing on their empirical performance. Six distinct methods were implemented and tested: the Recursive Method, Memoized Recursive Method, Matrix Exponentiation, Fast Doubling Method, Dynamic Programming, and the Binet Formula. The main comparison metric was the execution time of each algorithm, measured across different input sizes.

The basic **Recursive Method** demonstrated poor performance as the input size increased due to its exponential time complexity of $O(2^n)$. This inefficiency was evident in the execution times, particularly for larger values of n, where the algorithm required excessive computation due to recalculating the same Fibonacci numbers multiple times.

The **Memoized Recursive Method** showed significant improvement by storing previously computed values, which reduced the time complexity from $O(2^n)$ to $O(n)$. This method proved to be much faster and more efficient than the plain recursive approach, particularly when dealing with larger values of n.

The **Matrix Exponentiation Method** used matrix multiplication to calculate Fibonacci numbers in $O(\log n)$ time. This logarithmic time complexity resulted in a considerable speedup compared to the recursive methods, and it was much faster for larger input sizes. The graph data confirmed the expected $O(\log n)$ relationship between input size and execution time, making it a viable option for more extensive computations.

The **Fast Doubling Method** further optimized the matrix exponentiation approach by leveraging the relationships between consecutive Fibonacci numbers. Like matrix exponentiation, it achieved $O(\log n)$

time complexity but outperformed it in terms of execution speed. This method was the most efficient for large input sizes, providing both faster execution and maintaining accuracy in the results.

The **Dynamic Programming Method** stored only the last two Fibonacci numbers at each step, which made it highly efficient. With $O(n)$ time complexity and minimal space usage, this method performed excellently across all input sizes, being both fast and memory-efficient. It stands out as a reliable and optimal choice for computing Fibonacci numbers in practice.

Lastly, the **Binet Formula** provided a fast way to calculate Fibonacci numbers using the Golden Ratio. However, this method became unreliable for large input sizes due to rounding errors that accumulated as the Fibonacci numbers grew. While it demonstrated fast execution for smaller numbers, the lack of accuracy with larger n makes it unsuitable for practical applications at scale.

In conclusion, the **Dynamic Programming** and **Fast Doubling** methods emerged as the most efficient algorithms for calculating Fibonacci numbers, especially for large input sizes. The Memoized Recursive Method also offered a substantial improvement over the plain recursive method. Although the **Matrix Exponentiation** method is a strong candidate for moderately large Fibonacci numbers, **Fast Doubling** proved to be the best option overall in terms of both speed and accuracy. The **Binet Formula**, while fast, is not recommended for larger Fibonacci numbers due to its accuracy limitations.