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

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

Elaborated:
st. gr. FAF-232 Vremere Adrian

Verified:
asist. univ. Fiștic Cristofor

Chișinău - 2025

Content

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 the algorithms empirically;
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: $x_n = x_{n-1} + x_{n-2}$.

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. 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).

Therefore, within this laboratory, we will analyze seven algorithms for determining the Fibonacci N-th term 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 are 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 seven algorithms will be implemented in Python and empirically analyzed based on their execution time. While the overall trend of the results may align with other experimental findings, the specific efficiency relative to input size will vary depending on the memory capacity of the device used.

The error margin for the measurements is set at 2.5 seconds, based on experimental observations.

Recursive Method

The recursive method for computing the Fibonacci sequence adopts a natural approach by defining the problem in terms of itself. For any given n, the method computes the n-th term by first calculating the two preceding terms and then summing them to arrive at the final value. It does this by calling itself with arguments $n-1$ and $n-2$ until it reaches the base cases, typically when $n=0$ or $n=1$, where the Fibonacci number is already known.

This self-referential structure highlights the elegance of recursion, wherein a complex problem is broken down into simpler, identical subproblems. Despite its intuitive formulation, the method results in redundant computations, as the same operations are performed multiple times. This inefficiency is an important consideration when analyzing its behavior and performance.

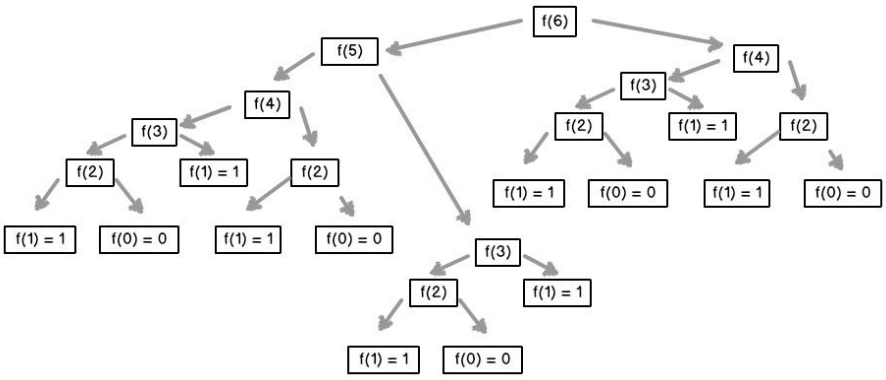

Figure 1 - Fibonacci Recursion Example

Algorithm Description

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

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

Listing 1 - Pseudocode Recursive Fibonacci

Implementation

```python
# Using the Recursive method
def recursive(n):
    if n <= 1:
        return n
    else:
        return recursive(n-1) + recursive(n-2)
```

Listing 2 - Recursive Fibonacci Implementation in Python

Results

After running the function for each n Fibonacci term proposed in the list from the first Input Format, also known as *Low N Terms list* in my implementation, and saving the time for each n, we obtained the following results:

n	5	7	10	12	15	17	20	22	25	27	30	32	35	37	40	42	45
Recursive	0.000002	0.000001	0.000006	0.000017	0.000071	0.000182	0.000736	0.001974	0.008500	0.022002	0.092869	0.247753	1.021370	2.690954	11.769791	30.287505	129.146745
Dynamic	0.00005	0.00001	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00001	0.00000	0.00000	0.00001	0.00000	0.00000	0.00001	0.00000	0.00000
Matrix	0.00003	0.00000	0.00001	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
Binet	0.00302	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
Memo.	0.00002	0.00000	0.00004	0.00000	0.00000	0.00000	0.00000	0.00003	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00001	0.00000
Iter.	0.00002	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
Doubling	0.00002	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

Figure 2 - Fibonacci Recursion Table Output

In *Figure 2*, the table of results for the first set of inputs is presented. It is evident that the execution time for computing Fibonacci numbers increases exponentially with larger values of n. When $n = 35$, the computation takes approximately one second, and by $n = 45$, it exceeds two minutes, whereas all other methods maintain extremely low execution times, close to zero.

Furthermore, in the graph shown in *Figure 3*, the rapid increase in execution time as n grows follows an exponential trend, approximately $T(2^n)$. This inefficiency arises because, for each new n, the recursive method calls itself twice, once for $n-1$ and once for $n-2$, resulting in a branching tree of recursive calls that grows exponentially. As a result, the number of redundant calculations increases drastically, making this approach highly inefficient compared to other methods.

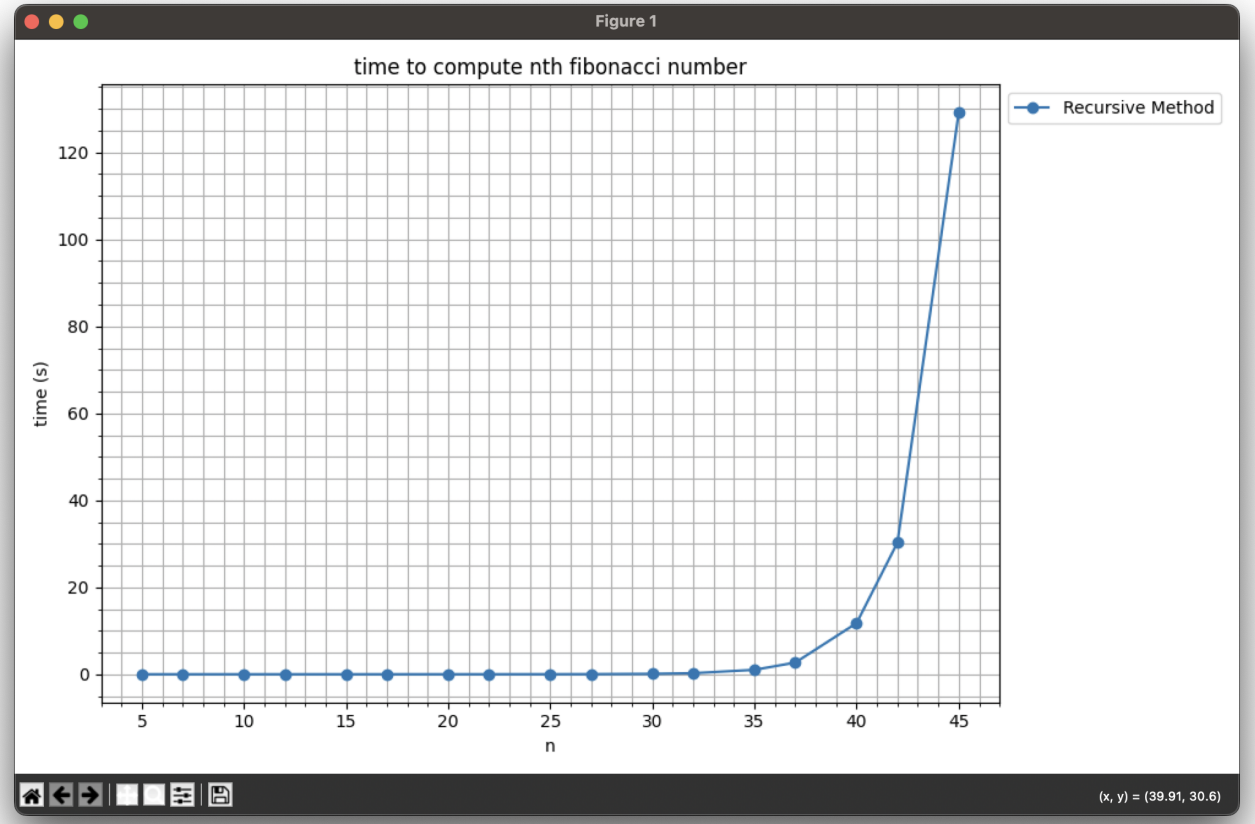

Figure 3 - Recursion Method Execution Time

Dynamic Programming Method

The Dynamic Programming method, similar to the recursive method, takes a straightforward approach to calculating the *n*-th term. However, instead of recursively calling itself in a top-down manner, it utilizes an array data structure to store previously computed terms, eliminating the need for redundant calculations.

Algorithm Description

The Dynamic Programming approach computes the Fibonacci sequence iteratively by storing previously computed values in an array, as shown in the next 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]
```

Listing 3 - Pseudocode Dynamic Programming

Implementation

```python
# uses a bottom-up approach to compute the n-th Fibonacci number.
def dynamic_programming(n):
    l = [0, 1]
    for i in range(2, n + 1):
        l.append(l[i - 1] + l[i - 2])
    return l[n]
```

Listing 4 - Dynamic Programming Fibonacci Implementation in Python

Results

This time, we are running the function for each n Fibonacci term proposed in the list from the first Second Format, also known as *High N Terms list* in my implementation, and saving the time for each n, we obtained the following results:

n	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
Dynamic	0.00004	0.00004	0.00005	0.00006	0.00008	0.00010	0.00014	0.00020	0.00027	0.00038	0.00054	0.00088	0.00139	0.00201	0.00314	0.00511
Matrix	0.00002	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00003	0.00003	0.00004	0.00005	0.00008	0.00009	0.00015	0.00020	0.00027
Binet	0.00001	0.00000	0.00000	0.00000	0.00000	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00003	0.00005	0.00008	0.00013	0.00020
Memo.	0.00011	0.00002	0.00003	0.00004	0.00005	0.00008	0.00009	0.00012	0.00016	0.00021	0.00030	0.00048	0.00065	0.00098	0.00170	0.00204
Iter.	0.00002	0.00002	0.00003	0.00003	0.00005	0.00006	0.00008	0.00011	0.00016	0.00024	0.00036	0.00060	0.00097	0.00154	0.00252	0.00380
Doubling	0.00001	0.00000	0.00000	0.00000	0.00000	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00004	0.00003	0.00008	0.00008	0.00011

Figure 4 - Fibonacci Dynamic Programming Table Output

In *Figure 4*, we observe that this method does not exhibit an exponential increase in execution time with each increment of n. Instead, the growth is gradual. This is because all previously computed values are stored in an array, allowing for efficient access and reuse.

The graph shown in *Figure 5* appears to be linear, indicating that the time complexity is $O(n)$.

One drawback of this method is its space complexity, which is $O(n)$ when using an array to store all computed values. However, an optimization exists that reduces space usage to $O(1)$ by keeping only the last two computed terms in memory. We will discuss this optimization later.

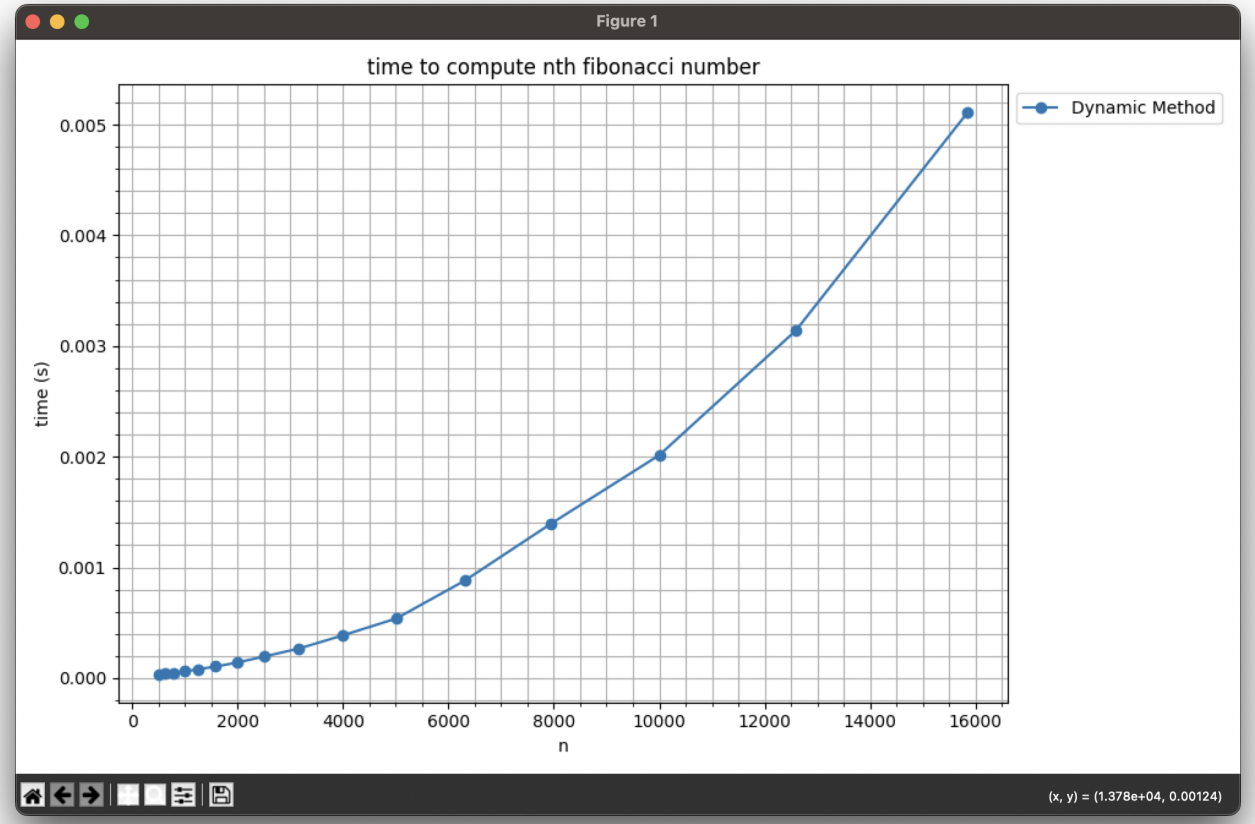

Figure 5 - Dynamic Programming Method Execution Time

Faster Matrix Power Method

The Matrix Exponentiation method computes the Fibonacci sequence efficiently by leveraging the property that Fibonacci numbers can be expressed using matrix multiplication:

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

By applying fast exponentiation, the matrix power can be computed in faster time, making this approach significantly faster than naive recursion and dynamic programming for large values of n.

Algorithm Description

The Matrix Exponentiation method computes Fibonacci numbers using the property that the sequence can be represented as a recurrence relation in matrix form. Specifically, the Fibonacci sequence follows the transformation:

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

By applying this relation repeatedly, we derive the general formula:

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

where $F_0 = 0$ and $F_1 = 1$. Instead of computing the matrix power through naive multiplication, which would take $O(n)$ time, we use **fast exponentiation** (also known as exponentiation by squaring). This reduces the number of required matrix multiplications to $O(\log n)$, making it significantly more efficient for large n.

The key idea behind fast exponentiation is that instead of multiplying the matrix $n-1$ times, we recursively square it when n is even, and multiply by one extra matrix when n is odd:

$$M^n = \begin{cases} (M^{n/2} \times M^{n/2}), & \text{if } n \text{ is even} \\ (M^{(n-1)/2} \times M^{(n-1)/2} \times M), & \text{if } n \text{ is odd} \end{cases}$$

The implementation is shown in the next few pseudocodes:

```
Fibonacci(n):
    if n <= 1 then
        return n
    F <- []
    vec <- [[0], [1]]
    Matrix <- [[0, 1], [1, 1]]
    F <- power(Matrix, n - 1)
    F <- F * vec
    return F[0][0]
```

Listing 5 - Pseudocode Matrix Power Fibonacci

```
power(Matrix, n):
    if n == 0 or n == 1 then
        return
    Mat2 <- [[1, 1], [1, 0]]
    power(Matrix, n // 2)
    multiply(Matrix, Matrix)
    if n mod 2 != 0 then
```

```
        multiply(Matrix, Mat2)
```

Listing 6 - Pseudocode Raise Matrix to n Power

```
multiply(mat1, mat2):
    x <- mat1[0][0] * mat2[0][0] + mat1[0][1] * mat2[1][0]
    y <- mat1[0][0] * mat2[0][1] + mat1[0][1] * mat2[1][1]
    z <- mat1[1][0] * mat2[0][0] + mat1[1][1] * mat2[1][0]
    w <- mat1[1][0] * mat2[0][1] + mat1[1][1] * mat2[1][1]
    mat1[0][0] <- x
    mat1[0][1] <- y
    mat1[1][0] <- z
    mat1[1][1] <- w
```

Listing 7 - Pseudocode Multiplying 2 Matrices

Implementation

```python
# multiplies two 2x2 matrices.
def multiply(mat1, mat2):
    x = mat1[0][0] * mat2[0][0] + mat1[0][1] * mat2[1][0]
    y = mat1[0][0] * mat2[0][1] + mat1[0][1] * mat2[1][1]
    z = mat1[1][0] * mat2[0][0] + mat1[1][1] * mat2[1][0]
    w = mat1[1][0] * mat2[0][1] + mat1[1][1] * mat2[1][1]

    mat1[0][0], mat1[0][1] = x, y
    mat1[1][0], mat1[1][1] = z, w
```

Listing 8 - Multiplying 2 Matrices Implementation in Python

```python
# raises a matrix to the n-th power.
def power(mat1, n):
    if n == 0 or n == 1:
        return
    mat2 = [[1, 1], [1, 0]]
    power(mat1, n // 2)
    multiply(mat1, mat1)
    if n % 2 != 0:
        multiply(mat1, mat2)
```

Listing 9 - Raise Matrix to n Power Implementation in Python

```python
# computes the n-th fibonacci number using matrix exponentiation.
def matrix_power(n):
    if n <= 1:
        return n
    mat1 = [[1, 1], [1, 0]]
```

```
    power(mat1, n - 1)
    return mat1[0][0]
```

Listing 10 - Matrix Power Fibonacci Implementation in Python

Results

n	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
Dynamic	0.00004	0.00004	0.00005	0.00006	0.00008	0.00010	0.00014	0.00020	0.00027	0.00038	0.00054	0.00088	0.00139	0.00201	0.00314	0.00511
Matrix	0.00002	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00003	0.00003	0.00004	0.00005	0.00008	0.00009	0.00015	0.00020	0.00027
Binet	0.00001	0.00000	0.00000	0.00000	0.00000	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00003	0.00005	0.00008	0.00013	0.00020
Memo.	0.00011	0.00002	0.00003	0.00004	0.00005	0.00008	0.00009	0.00012	0.00016	0.00021	0.00030	0.00048	0.00065	0.00098	0.00170	0.00204
Iter.	0.00002	0.00002	0.00003	0.00003	0.00005	0.00006	0.00008	0.00011	0.00016	0.00024	0.00036	0.00060	0.00097	0.00154	0.00252	0.00380
Doubling	0.00001	0.00000	0.00000	0.00000	0.00000	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00004	0.00003	0.00008	0.00008	0.00011

Figure 6 - Fibonacci Matrix Power Table Output

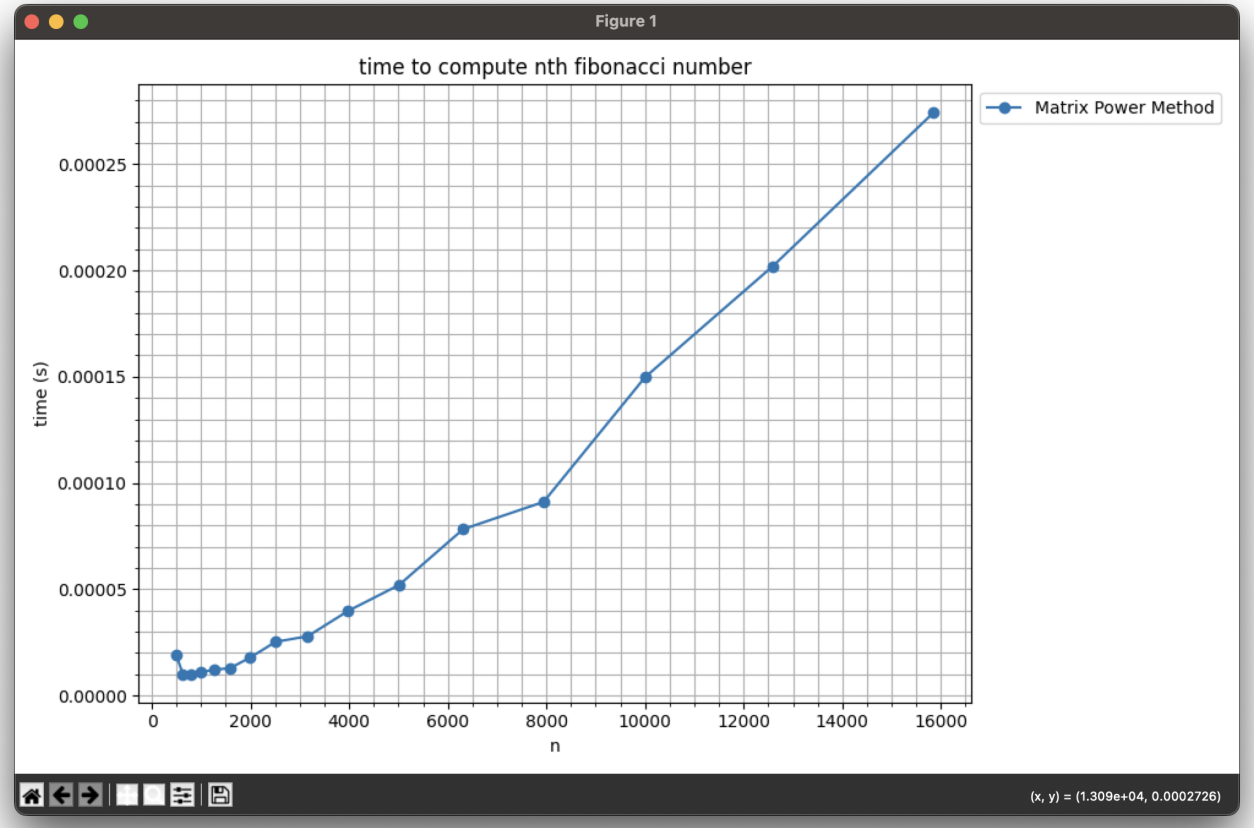

Figure 7 - Matrix Power Method Execution Time

In *Figure 6*, we observe that the execution time of this method increases very slowly as n grows. When compared to the dynamic programming approach, we see that matrix exponentiation is significantly faster. This efficiency arises from the use of matrix exponentiation, which reduces the time complexity from

$O(n)$, as seen in the dynamic programming approach, to $O(\log n)$.

Taking into account the observations above, along with the data represented in *Figure 7*, which illustrates the gradual increase in execution time, we can conclude that the time complexity of this method is indeed $O(\log n)$.

Binet's Formula Method

Binet's Formula provides a closed-form expression for computing Fibonacci numbers without iteration or recursion. It is derived from the characteristic equation of the Fibonacci recurrence relation and expresses F_n in terms of the golden ratio φ and its conjugate. Although it offers an efficient theoretical approach, its reliance on floating-point arithmetic can lead to precision issues for large n.

Algorithm Description

Binet's Formula provides a mathematical approach to computing Fibonacci numbers directly, avoiding iterative or recursive methods. It is derived from the characteristic equation of the Fibonacci recurrence and is given by:

$$F_n = \frac{1}{\sqrt{5}}(\varphi^n - \psi^n)$$

where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio, and $\psi = \frac{1-\sqrt{5}}{2}$ is its conjugate. Since $|\psi| < 1$, the term ψ^n approaches zero as n increases, making F_n approximately equal to $\frac{\varphi^n}{\sqrt{5}}$.

While this formula is theoretically efficient, it relies on floating-point arithmetic, which introduces rounding errors. These errors become significant for large n, where precision loss can cause incorrect results, especially when rounding to the nearest integer. As a result, Binet's Formula is most reliable for moderate values of n, beyond which exact computation using integer-based methods is preferred. The implementation is shown in the next pseudocode:

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

Listing 11 - Pseudocode Binet's Formula Fibonacci

Implementation

```python
# uses binet's formula to compute the n-th fibonacci number.
def binet_formula(n):
    n = Decimal(n)
    return int((phi ** n - phi_hat ** n) / sqrt_5)
```

Listing 12 - Binet's Formula Fibonacci Implementation in Python

Results

n	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
Dynamic	0.00004	0.00004	0.00005	0.00006	0.00008	0.00010	0.00014	0.00020	0.00027	0.00038	0.00054	0.00088	0.00139	0.00201	0.00314	0.00511
Matrix	0.00002	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00003	0.00003	0.00004	0.00005	0.00008	0.00009	0.00015	0.00020	0.00027
Binet	0.00001	0.00000	0.00000	0.00000	0.00000	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00003	0.00005	0.00008	0.00013	0.00020
Memo.	0.00011	0.00002	0.00003	0.00004	0.00005	0.00008	0.00009	0.00012	0.00016	0.00021	0.00030	0.00048	0.00065	0.00098	0.00170	0.00204
Iter.	0.00002	0.00002	0.00003	0.00003	0.00005	0.00006	0.00008	0.00011	0.00016	0.00024	0.00036	0.00060	0.00097	0.00154	0.00252	0.00380
Doubling	0.00001	0.00000	0.00000	0.00000	0.00000	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00004	0.00003	0.00008	0.00008	0.00011

Figure 8 - Fibonacci Binet's Formula Table Output

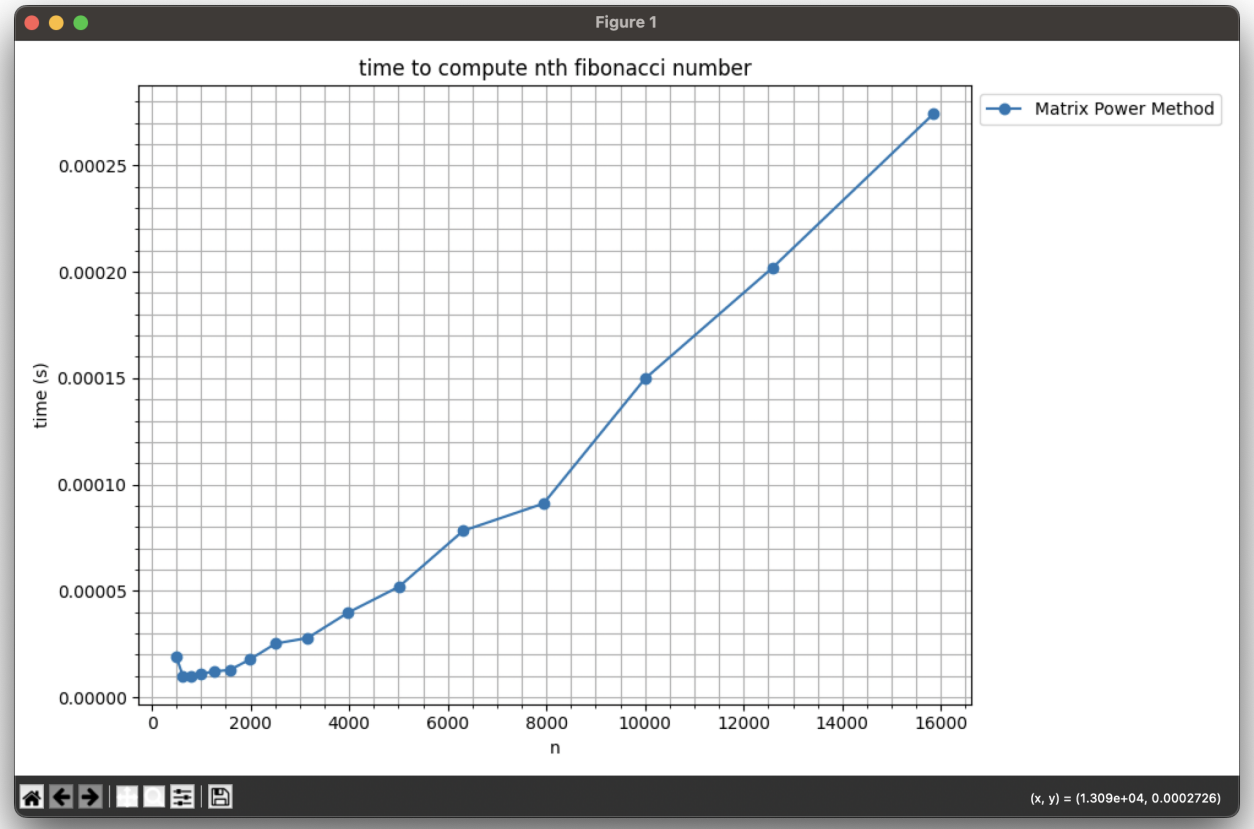

Figure 9 - Binet's Formula Method Execution Time

As expected, we can see in *Figure 9* that the execution time is very fast; however, the results become inaccurate for $n > 71$, making this method reliable only for small n but impractical for larger values.

Additionally, while we might expect this method to have $O(1)$ complexity since it consists of a few arithmetic operations, the graph in *Figure 8* suggests otherwise. This is due to the exponentiation step, which is typically performed using methods like exponentiation by squaring, leading to a time complexity

of $O(\log n)$ rather than true constant time.

Iterative Method

The iterative method follows the same approach as the dynamic programming method but eliminates the need for an array to store all Fibonacci numbers.

Algorithm Description

Instead of maintaining an entire array, we store only the two most recent Fibonacci numbers at each step. At each iteration, we update these two values to compute the next Fibonacci number efficiently, as shown in the following pseudocode.

```
Fibonacci(n):
    if n = 0 or n = 1 then
        return n
    previous <- 0
    fib_number <- 1
    for i <- 2 to n do
        temp <- fib_number
        fib_number <- previous + fib_number
        previous <- temp
    return fib_number
```

Listing 13 - Pseudocode Iterative Fibonacci

Implementation

```python
# iterative approach to calculate the n-th fibonacci number.
def iterative(n):
    if n in {0, 1}:
        return n
    previous, fib_number = 0, 1
    for _ in range(2, n + 1):
        previous, fib_number = fib_number, previous + fib_number
    return fib_number
```

Listing 14 - Iterative Fibonacci Implementation in Python

Results

n	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
Dynamic	0.00004	0.00004	0.00005	0.00006	0.00008	0.00010	0.00014	0.00020	0.00027	0.00038	0.00054	0.00088	0.00139	0.00201	0.00314	0.00511
Matrix	0.00002	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00003	0.00003	0.00004	0.00005	0.00008	0.00009	0.00015	0.00020	0.00027
Binet	0.00001	0.00000	0.00000	0.00000	0.00000	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00003	0.00005	0.00008	0.00013	0.00020
Memo.	0.00011	0.00002	0.00003	0.00004	0.00005	0.00008	0.00009	0.00012	0.00016	0.00021	0.00030	0.00048	0.00065	0.00098	0.00170	0.00204
Iter.	0.00002	0.00002	0.00003	0.00003	0.00005	0.00006	0.00008	0.00011	0.00016	0.00024	0.00036	0.00060	0.00097	0.00154	0.00252	0.00380
Doubling	0.00001	0.00000	0.00000	0.00000	0.00000	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00004	0.00003	0.00008	0.00008	0.00011

Figure 10 - Fibonacci Iterative Table Output

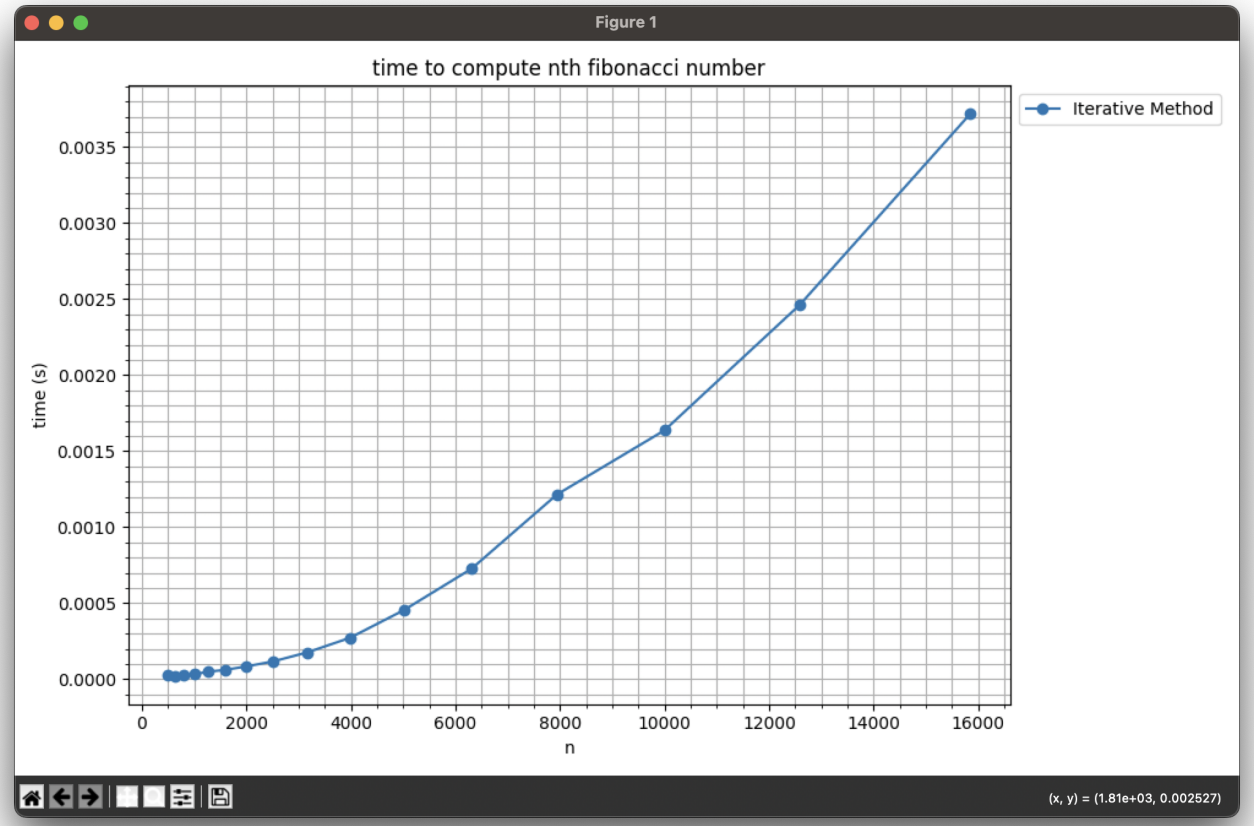

Figure 11 - Iterative Method Execution Time

Even though this method is not significantly faster than the dynamic programming method, as shown in *Figure 10*, it is preferable due to its space complexity of $O(1)$, making it more memory-efficient.

Given its comparable performance to the dynamic method, as observed in the previously mentioned table, and the linear trend in the graph *Figure 11*, we can conclude that its time complexity is $T(n)$.

Memoization Recursive Method

The memoization method is a top-down approach that improves the efficiency of the naive recursive method by storing previously computed Fibonacci numbers in a cache. This prevents redundant calculations and significantly reduces the number of recursive calls.

Algorithm Description

The memoization method enhances the recursive approach by storing previously computed Fibonacci numbers in a lookup table. When computing $F(n)$, we first check if the result is already stored; if so, we return it immediately. Otherwise, we recursively compute $F(n-1)$ and $F(n-2)$, store their sum in the table, and return the result. The next pseudocode shows it's implementation:

```
cache <- {0: 0, 1: 1}

Fibonacci(n):
    if n in cache then
        return cache[n]
    cache[n] <- Fibonacci(n - 1) + Fibonacci(n - 2)
    return cache[n]
```

Listing 15 - Pseudocode Memoization Recursive Fibonacci

Implementation

```python
cache = {0: 0, 1: 1}

# uses memoization to store results and calculate fibonacci.
def memoization(n):
    if n in cache:
        return cache[n]
    cache[n] = memoization(n - 1) + memoization(n - 2)
    return cache[n]
```

Listing 16 - Memoization Recursive Fibonacci Implementation in Python

Results

n	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
Dynamic	0.00004	0.00004	0.00005	0.00006	0.00008	0.00010	0.00014	0.00020	0.00027	0.00038	0.00054	0.00088	0.00139	0.00201	0.00314	0.00511
Matrix	0.00002	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00003	0.00003	0.00004	0.00005	0.00008	0.00009	0.00015	0.00020	0.00027
Binet	0.00001	0.00000	0.00000	0.00000	0.00000	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00003	0.00005	0.00008	0.00013	0.00020
Memo.	0.00011	0.00002	0.00003	0.00004	0.00005	0.00008	0.00009	0.00012	0.00016	0.00021	0.00030	0.00048	0.00065	0.00098	0.00170	0.00204
Iter.	0.00002	0.00002	0.00003	0.00003	0.00005	0.00006	0.00008	0.00011	0.00016	0.00024	0.00036	0.00060	0.00097	0.00154	0.00252	0.00380
Doubling	0.00001	0.00000	0.00000	0.00000	0.00000	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00004	0.00003	0.00008	0.00008	0.00011

Figure 12 - Fibonacci Memoization Recursion Table Output

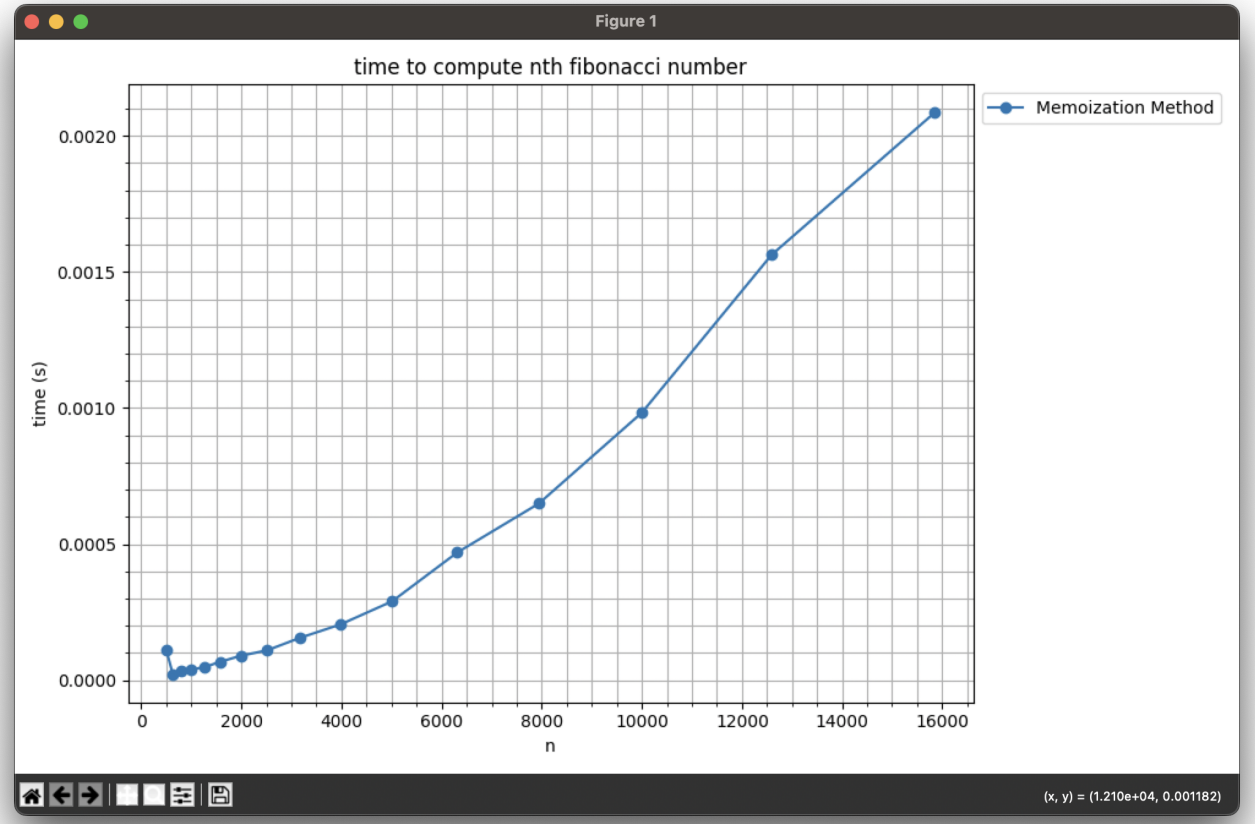

Figure 13 - Memoization Recursion Method Execution Time

In *Figure 12*, we observe that the memoization method performs faster than the previous $T(n)$ algorithms. This improvement is due to the properties of Python's dictionary, which allows for efficient lookups and storage of previously computed values. While this method may offer better speed compared to the iterative approach, it comes at the cost of higher space complexity, $O(n)$, since it stores all computed Fibonacci numbers.

From the stepwise increase in execution time as n grows and the graph shown in *Figure 13*, which exhibits a linear trend, we conclude that this method has a time complexity of $T(n)$.

Fast Doubling Method

The Fast Doubling method is an optimized approach for computing Fibonacci numbers using the properties of matrix exponentiation. It relies on the identities derived from the Fibonacci sequence to compute $F(n)$ and $F(n+1)$ simultaneously.

Algorithm Description

The Fast Doubling method efficiently computes Fibonacci numbers using the recurrence relations:

$$F(2k) = F(k) \cdot (2F(k+1) - F(k))$$

$$F(2k+1) = F(k+1)^2 + F(k)^2$$

Instead of computing Fibonacci numbers sequentially, this approach recursively applies the above formulas, reducing the number of required operations. The function computes $F(n)$ and $F(n+1)$ simultaneously, utilizing integer division to break n into smaller subproblems. Since the method avoids redundant calculations and only requires constant space, it is both time and space efficient. The implementation can be seen in the following pseudocode:

```
Fibonacci(n):
    return _fib(n)[0]

_fib(n):
    if n == 0 then
        return (0, 1)
    (a, b) <- _fib(n // 2)
    c <- a * (2 * b - a)
    d <- a * a + b * b
    if n mod 2 == 0 then
        return (c, d)
    else
        return (d, c + d)
```

Listing 17 - Pseudocode Fast Doubling Fibonacci Implementation

Implementation

```python
# uses the fast doubling algorithm to calculate fibonacci.
def fast_doubling(n):
    return _fib(n)[0]


# helper function 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)
```

Listing 18 - Fast Doubling Fibonacci Implementation in Python

Results

n	501	631	794	1000	1259	1585	1995	2512	3162	3981	5012	6310	7943	10000	12589	15849
Dynamic	0.00004	0.00004	0.00005	0.00006	0.00008	0.00010	0.00014	0.00020	0.00027	0.00038	0.00054	0.00088	0.00139	0.00201	0.00314	0.00511
Matrix	0.00002	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00003	0.00003	0.00004	0.00005	0.00008	0.00009	0.00015	0.00020	0.00027
Binet	0.00001	0.00000	0.00000	0.00000	0.00000	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00003	0.00005	0.00008	0.00013	0.00020
Memo.	0.00011	0.00002	0.00003	0.00004	0.00005	0.00008	0.00009	0.00012	0.00016	0.00021	0.00030	0.00048	0.00065	0.00098	0.00170	0.00204
Iter.	0.00002	0.00002	0.00003	0.00003	0.00005	0.00006	0.00008	0.00011	0.00016	0.00024	0.00036	0.00060	0.00097	0.00154	0.00252	0.00380
Doubling	0.00001	0.00000	0.00000	0.00000	0.00000	0.00001	0.00001	0.00001	0.00001	0.00001	0.00002	0.00004	0.00003	0.00008	0.00008	0.00011

Figure 14 - Fibonacci Fast Doubling Table Output

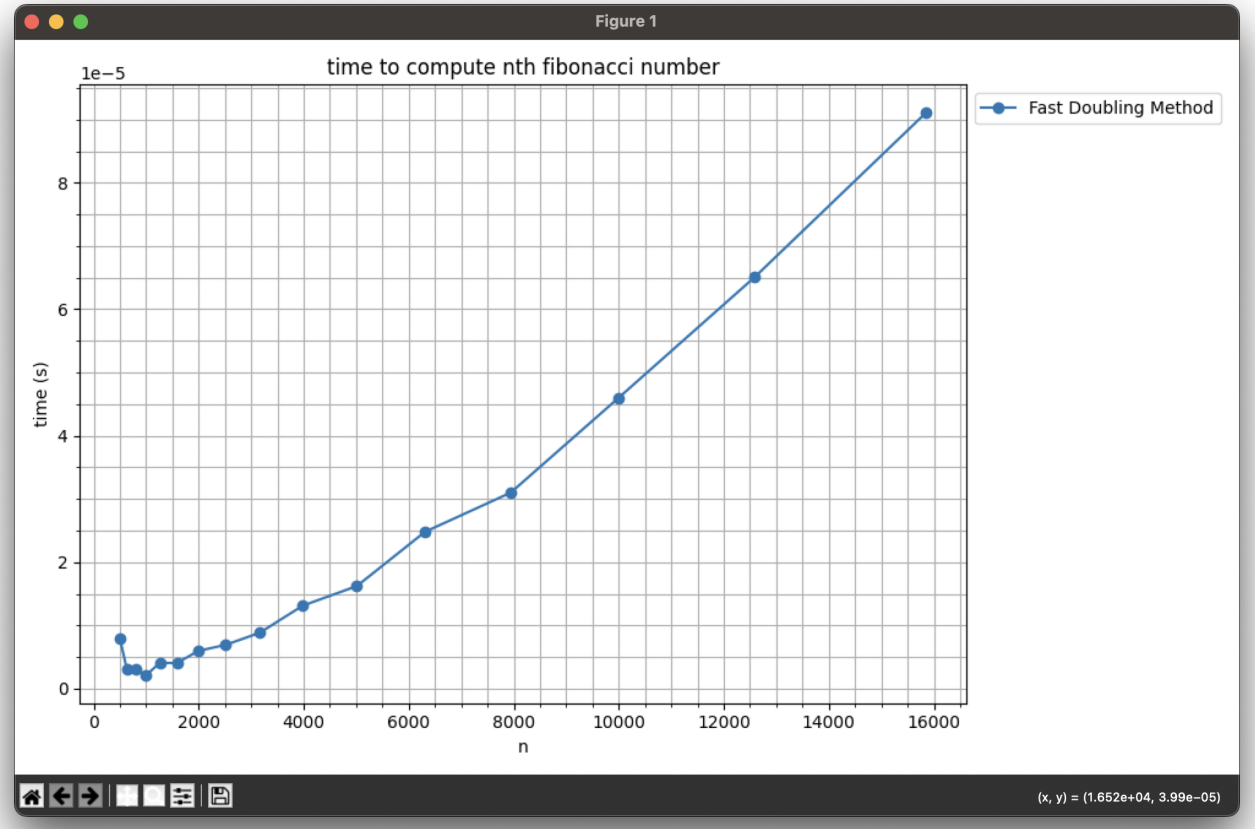

Figure 15 - Fast Doubling Method Execution Time

Based on *Figure 14*, we can see that the computation of such a large n was performed very efficiently using this method, making it the fastest among all benchmarks. It surpasses the previous $T(\log n)$ method (Matrix Exponentiation) because it relies on scalar multiplications and additions rather than matrix multiplications, which are computationally more expensive.

The graph *Figure 15*, along with the fact that this method outperforms the matrix power method, confirms its $T(\log n)$ time complexity.

CONCLUSION

Conclusion

To better compare the performance of all implemented algorithms, I decided to plot their execution times together, excluding the recursive method. Since recursion has a time complexity of $T(2^n)$, it is significantly slower and inefficient compared to the rest, making its inclusion unnecessary.

From *Figure 16*, we can observe that the fastest method is the Fast Doubling method, followed by Binet's Formula, the Matrix Exponentiation method, Memoization, the Iterative method, and finally the Dynamic Programming approach. This ranking remains consistent for moderate values of n.

However, when increasing n to 500,000, as shown in *Figure 17*, the Iterative method surpasses Memoization in speed. This is likely due to the overhead of dictionary lookups in Python, making the Iterative approach more efficient for very large values of n.

Overall, the Fast Doubling method proves to be the best approach for computing Fibonacci numbers, offering both high efficiency and low computational overhead. While Binet's Formula is fast, its precision limitations make it unreliable for large values of n. The Matrix Exponentiation method provides a good balance between speed and accuracy, while Memoization and Iterative methods remain strong choices for simpler implementations.

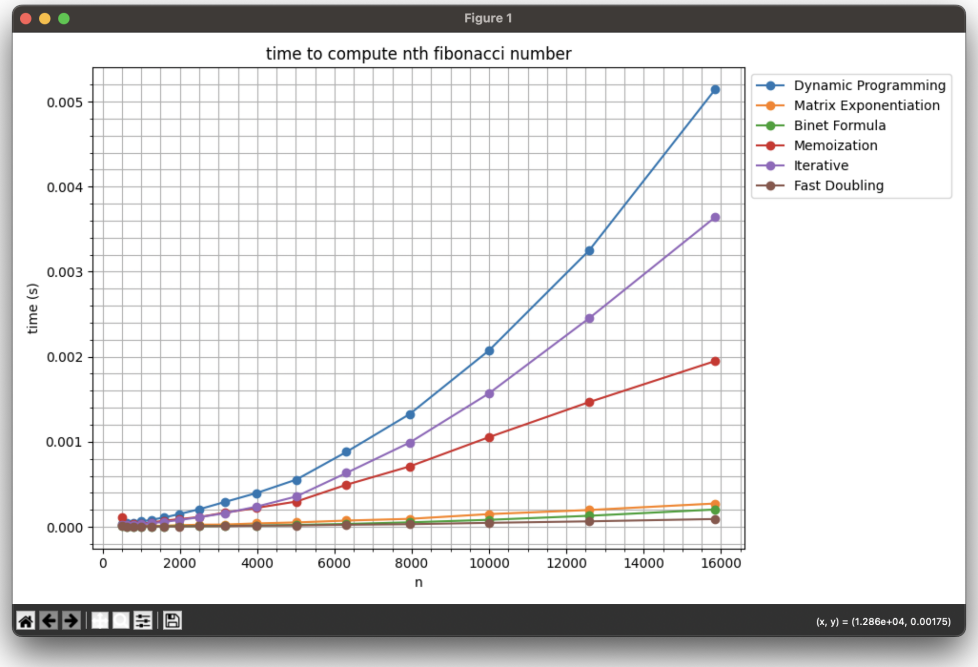

Figure 16 - Fast Doubling Method Execution Time

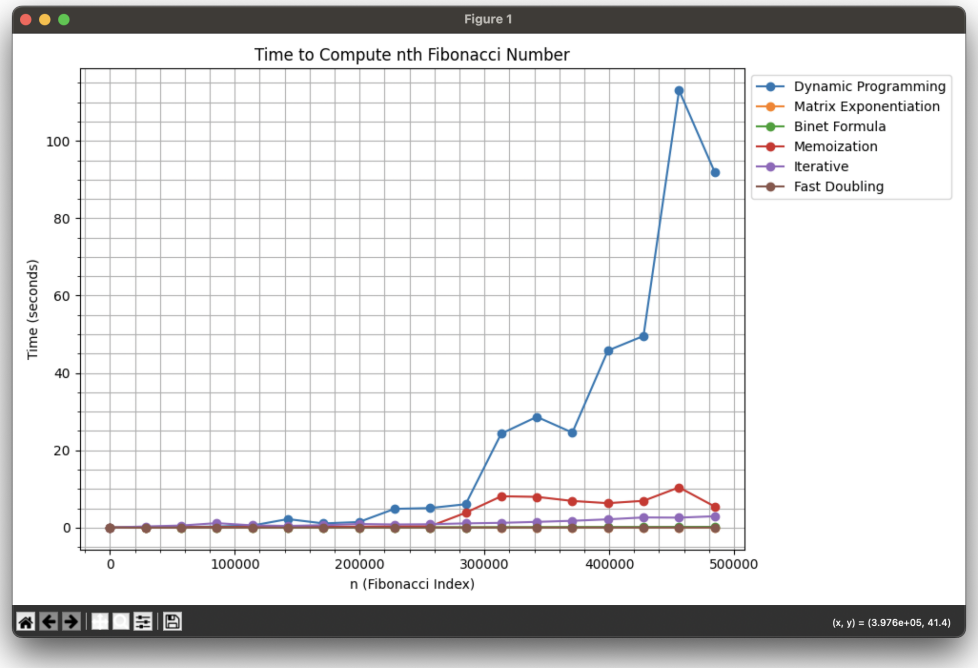

Figure 17 - Fast Doubling Method Execution Time

Bibliography

[1] Github. ”lab1”. https://github.com/mcittkmims/aa_labs.