Analyzing the computational complexity and performance efficiency of Édouard Lucas method of utilizing Pascal's Triangle for deriving the Nth Fibonacci number compared to the classical recursive algorithm, Iterative algorithm and Matrix Exponentiation.

*Note: Sub-titles are not captured in Xplore and should not be used

1st Aadesh Tikhe
2^{nd} *year BCA(AUM)*
Amity University Panvel
Chhatrapati Sambhajinagar, India
https://orcid.org/0009-0009-3185-0845

2rd Dr. Ron Knott
Ph.D(1980, University of Nottingham)
Lecturer of Mathematics and CS at the University of Surrey
Bolton, Manchester England
ronknott@mac.com

***Abstract*—The Fibonacci sequence plays a critical role in mathematical and computational applications, with numerous methods proposed for its computation. This paper investigates the French mathematician François Édouard Lucas's (1842–1891) combinatorial approach, which leverages Pascal's Triangle to derive the nth Fibonacci number by summing its shallow diagonals. The computational complexity and performance efficiency of this method are rigorously analyzed and compared to the classical recursive algorithm. Through theoretical analysis and empirical evaluation, this study aims to determine the feasibility and scalability of Édouard Lucas's method for efficient computation of Fibonacci numbers. The findings provide insights into alternative combinatorial strategies for optimizing Fibonacci computations in resource-constrained environments.**

***Index Terms*—Fibonacci Sequence, Pascal's Triangle, Combinatorial Algorithms, Computational Complexity, Algorithm Optimization**

I. Introduction

The Fibonacci sequence is a fundamental mathematical construct with applications in various domains, including computer science, cryptography, and biology. Defined recursively, where each term is the sum of its two immediate predecessors, the sequence grows rapidly and has inspired the development of numerous computational methods. While traditional approaches, such as recursive and iterative algorithms, are widely used, they often face performance challenges, particularly for large values of n. Efficient computation of Fibonacci numbers is, therefore, an important problem in both theoretical and applied contexts.

Identify applicable funding agency here. If none, delete this.

Sir Édouard Lucas proposed a combinatorial approach that leverages Pascal's Triangle to derive Fibonacci numbers by summing its shallow diagonals. This method establishes an elegant connection between the Fibonacci sequence and Pascal's Triangle, offering a combinatorial alternative to traditional algorithms. However, its computational efficiency and scalability, especially in comparison to classical recursive techniques, remain understudied.

This paper examines Édouard Lucas's method in detail, analyzing its computational complexity and performance efficiency relative to the classical recursive algorithm. Both theoretical analysis and empirical evaluation are conducted to assess the feasibility and scalability of the approach for large-scale computations. The study also identifies opportunities for optimizing Édouard Lucas's method to improve its performance in resource-constrained environments.

By exploring an alternative combinatorial perspective, this research contributes to a deeper understanding of efficient Fibonacci computations and highlights the potential of leveraging combinatorial techniques in broader algorithmic applications.

II. Literature Review

A. Historical Development of the Fibonacci Sequence

The Fibonacci sequence, introduced in the 13th century by Leonardo of Pisa, known as Fibonacci, first appeared in his book *Liber Abaci* (1202). The sequence was posed in the context of a problem involving the growth of a rabbit population under idealized conditions. Mathematically, the Fibonacci sequence is defined recursively as $F(0) = 0, F(1) = 1$, and

$F(n) = F(n-1) + F(n-2)$ for $n \geq 2$. Over time, this sequence has gained widespread attention due to its natural occurrence in biology, art, and architecture and its connections to the golden ratio, $\phi = (1+\sqrt{5})/2$, as the ratio $F(n+1)/F(n)$ converges to ϕ as $n \to \infty$ [1].

B. Mathematical Properties of the Fibonacci Sequence

The Fibonacci sequence has several intriguing mathematical properties. Its terms are related to binomial coefficients, as Fibonacci numbers can be expressed combinatorially using Pascal's Triangle. For example, summing the shallow diagonals of Pascal's Triangle yields Fibonacci numbers, a connection that has fascinated mathematicians for centuries. Additionally, Fibonacci numbers satisfy identities such as:

$$F(n+k) = F(k)F(n+1) + F(k-1)F(n), \tag{1}$$

and the sequence grows asymptotically as:

$$F(n) \sim \frac{\phi^n}{\sqrt{5}}. \tag{2}$$

These properties have made the Fibonacci sequence a central object of study in number theory, combinatorics, and discrete mathematics [2].

C. Connections Between Fibonacci Sequence and Pascal's Triangle

Pascal's Triangle, a triangular arrangement of binomial coefficients, has a deep connection to the Fibonacci sequence. By summing the shallow diagonals of Pascal's Triangle, Fibonacci numbers are generated, establishing a direct combinatorial relationship between these two mathematical structures. François Edouard Anatole Lucas explored this relationship in depth, illustrating how Fibonacci numbers can be extracted from Pascal's Triangle [3]. This combinatorial perspective provides an alternative approach to computing Fibonacci numbers, emphasizing their structural and mathematical elegance.

D. Existing Computational Algorithms

Several algorithms have been developed for computing Fibonacci numbers, ranging from simple recursive methods to highly optimized approaches:

1) **Recursive Algorithm:** The most straightforward method involves directly implementing the recursive definition. While simple, this approach has exponential time complexity $O(2^n)$ due to repeated subproblems, making it impractical for large n [4].
2) **Iterative Algorithm:** Iterative methods improve efficiency by computing Fibonacci numbers sequentially, maintaining only the last two values in memory. This reduces the time complexity to $O(n)$ and space complexity to $O(1)$ [5].
3) **Matrix Exponentiation:** Fibonacci numbers can also be computed using matrix exponentiation, leveraging the relationship:

$$\begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n. \tag{3}$$

This approach achieves logarithmic time complexity $O(\log n)$ through fast exponentiation [6].
4) **Closed-Form Formula (Binet's Formula):** Fibonacci numbers can also be computed directly using Binet's formula:

$$F(n) = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}. \tag{4}$$

While efficient for small n, this method may encounter precision issues for large values [7].

E. Sir François Édouard Anatole Lucas Contributions

Sir Édouard Lucas made significant contributions to the understanding and computation of Fibonacci numbers by exploring their deep connections with Pascal's Triangle. His work demonstrated that Fibonacci numbers can be derived through a combinatorial approach, specifically by summing the shallow diagonals of Pascal's Triangle. This method provides an alternative to traditional algorithms, such as the recursive and iterative approaches, and offers a unique perspective on the relationship between two foundational structures in mathematics.

Édouard Lucas's approach leverages the inherent symmetry and combinatorial properties of Pascal's Triangle, which is traditionally used to compute binomial coefficients. By examining the shallow diagonals, Édouard Lucas revealed a striking correspondence: the sums of elements in each diagonal directly yield Fibonacci numbers. For instance, the first few diagonals of Pascal's Triangle sum to $1, 1, 2, 3, 5, 8$, and so on, which are the initial terms of the Fibonacci sequence. This relationship not only enhances the understanding of Fibonacci numbers but also introduces a visually intuitive way of computing them, which is particularly valuable for educational purposes. [8]

From a mathematical perspective, E Lucas's contributions underline the interplay between combinatorics and number theory. His work bridges the gap between two historically significant constructs—Pascal's Triangle, introduced by Blaise Pascal in the 17th century, and the Fibonacci sequence, popularized by Leonardo of Pisa in the 13th century. By connecting these constructs, E Lucas's approach highlights the versatility of combinatorial methods in solving numerical problems.

Despite its elegance, the E Lucas method raises questions about its computational efficiency. While it is conceptually simpler and visually intuitive, its feasibility for large-scale Fibonacci computations remains a topic of investigation. In practical scenarios, where speed and resource optimization are crucial, iterative and matrix-based methods often outperform combinatorial approaches. For instance, iterative algorithms have a linear time complexity, and matrix exponentiation methods achieve logarithmic time complexity. E Lucas method, by contrast, requires the construction of Pascal's Triangle up to the desired diagonal, which can lead to significant memory overhead and computational costs for large values of n.

Furthermore, E Lucas's work has implications beyond Fibonacci numbers. The combinatorial insights gained from his method inspire a broader exploration of connections between Pascal's Triangle and other mathematical sequences.

For example, similar techniques could potentially be applied to generate or analyze other sequences with combinatorial properties, such as Lucas numbers or generalized Fibonacci sequences. These extensions highlight the potential of E Lucas's contributions to serve as a foundation for further research in combinatorics and computational mathematics.

In summary, François Édouard Anatole Lucas's contributions not only provide a novel combinatorial method for deriving Fibonacci numbers but also enhance the appreciation of their structural beauty. His work invites both theoretical exploration and practical evaluation, bridging historical mathematical constructs and modern computational challenges. Future research may focus on optimizing his method for practical use, exploring its scalability, and uncovering its potential applications in other domains of mathematics and computation.

F. Algorithm Optimization and Computational Complexity

The study of algorithm optimization for Fibonacci computation has focused on improving time and space complexity. Recursive algorithms, while intuitive, are computationally expensive and often serve as a baseline for comparison. Iterative and matrix-based methods are more efficient, with the latter achieving near-optimal performance. Optimization techniques, such as dynamic programming and memoization, have further reduced redundant calculations in recursive methods [9].

Recent works have explored the use of combinatorial approaches, such as E Lucas's adaptation of Pascal's Triangle, to compute Fibonacci numbers. While these methods highlight the mathematical beauty of the sequence, their computational complexity, and scalability for large n require deeper analysis. Theoretical advancements in this area could lead to novel algorithms for Fibonacci computation and other combinatorial problems.

G. Conclusion

This review highlights the rich history, mathematical properties, and computational methods associated with the Fibonacci sequence. It underscores the significance of François Édouard Anatole Lucas's work in connecting Fibonacci numbers to Pascal's Triangle and positions it within the broader context of algorithmic research. By examining existing methods and their limitations, this study lays the groundwork for further exploration of combinatorial algorithms and their optimization for practical applications.

III. General Information on Algorithms

The term "algorithm" originates from the Latinized name of the Persian mathematician Al-Khwarizmi, whose works introduced fundamental concepts in mathematics and computation [10].

A. Definition of Key Terms

Algorithm: A well-defined procedure that takes input values and produces output values through a sequence of calculation steps. It effectively bridges the input and output values [11]:

$$\text{Algorithm: Input} \rightarrow \text{Process} \rightarrow \text{Output}$$

Program: The implementation of an algorithm in a given programming language or computing architecture. For example, programming may involve explicit memory management, such as dynamic allocation in C, which is not addressed at the algorithmic level [12].

Complexity of an Algorithm: A measure of the fundamental operations performed by an algorithm on a dataset, typically expressed as a function of the data size. Let D_n represent the set of data of size n, and $T(d)$ represent the cost of the algorithm on a data element d.

B. Types of Complexity

Best-Case Complexity: The minimal number of operations an algorithm performs on a dataset of fixed size n. This is represented as:

$$T_{\min}(n) = \min_{d \in D_n} C(d)$$

Where $C(d)$ is the cost of operations for a specific data element d [13].

Worst-Case Complexity: The maximum number of operations required by an algorithm for a dataset of fixed size n. This is represented as:

$$T_{\max}(n) = \max_{d \in D_n} C(d)$$

- *Advantage:* Guarantees an upper limit, ensuring the algorithm will not take longer than $T_{\max}(n)$.
- *Disadvantage:* May not reflect the typical behavior of the algorithm, as the worst case might rarely occur [14].

Average-Case Complexity: The average complexity of an algorithm across all datasets of size n, weighted by their probabilities of occurrence. This is given as:

$$T_{\text{avg}}(n) = \frac{\sum_{d \in D_n} P(d) \cdot C(d)}{|D_n|}$$

where $P(d)$ is the probability of occurrence of data element d [15].

- *Advantage:* Reflects general behavior if extreme cases are rare or if complexity varies slightly across datasets.
- *Disadvantage:* May not reflect significant deviations caused by individual datasets.

C. Optimality

An algorithm is deemed **optimal** when its complexity is the lowest among all algorithms in its class. While this discussion focuses on time complexity, other factors like memory usage or bandwidth may also be considered [16].

D. Importance of Algorithm Analysis

Analyzing an algorithm helps identify its characteristics, assess suitability for applications, and compare it with alternatives. Key metrics include execution time and memory usage. Typically, algorithms are analyzed independently of their operating environment to derive generalized estimates [17].

E. Scientific Study of Algorithms

Algorithm analysis includes:

1) Determining the order of performance in the worst case, aiming to establish optimal algorithms with proven lower performance limits. This is often referred to as "calculation of complexity" [18].
2) Analyzing algorithms across best, average, and worst cases, employing rigorous methods to achieve precision [19].

This analysis is essential for selecting efficient algorithms tailored to specific problems, programming languages, and computing environments.

IV. Mathematical Framework

A. Definition of Fibonacci Sequence

The Fibonacci sequence is a series of numbers in which each term (after the first two terms) is the sum of its two immediate predecessors.

A numerical sequence is defined as a function u that maps natural numbers $\mathbb{N}$ to a subset A of the real numbers $\mathbb{R}$, as shown below:

$$u : \mathbb{N} \to A$$

$$n \mapsto u(n) = u_n$$

Formally, the Fibonacci sequence $F(n)$ is defined by the following recurrence relation [1]:

$$F(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F(n-1) + F(n-2) & \text{if } n \geq 2. \end{cases}$$

Here:

- $F(0) = 0$ is the initial term of the sequence.
- $F(1) = 1$ is the second term of the sequence.
- $F(n) = F(n-1) + F(n-2)$ defines the sequence recursively for $n \geq 2$ [2].

1) First Few Terms

The first few terms of the Fibonacci sequence are [20]:

$$F(0) = 0,\ F(1) = 1,\ F(2) = 1,\ F(3) = 2,$$
$$F(4) = 3,\ F(5) = 5,\ F(6) = 8,\ F(7) = 13, \ldots$$

2) Properties

1) **Initial Conditions:** The sequence begins with the terms $F(0) = 0$ and $F(1) = 1$ [1].
2) **Recurrence Relation:** The sequence is defined recursively, with each term being the sum of the two preceding terms.
3) **Asymptotic Behavior:** As $n \to \infty$, the ratio $\frac{F(n+1)}{F(n)}$ approaches the golden ratio $\phi = \frac{1+\sqrt{5}}{2}$ [21].

This formal definition captures the essential nature of the Fibonacci sequence as a simple yet profoundly important mathematical construct.

B. Pascal's Triangle and Its Combinatorial Significance

Pascal's Triangle is a triangular array of numbers where each entry represents a binomial coefficient. Constructed using a simple rule, the triangle begins with a single 1 at the top (row 0). Each subsequent row is formed by adding adjacent numbers from the previous row, with the outer edges always being 1. The number in the n-th row and k-th column is given by the binomial coefficient [22]:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad 0 \leq k \leq n. \tag{5}$$

This number is also denoted as $T(n, k)$, representing the value at the k-th position in the n-th row of Pascal's Triangle.

1) Structure of Pascal's Triangle

The first few rows of Pascal's Triangle are shown below [1]:

$$\begin{array}{ccccccc} & & & 1 & & & \\ & & 1 & & 1 & & \\ & 1 & & 2 & & 1 & \\ 1 & & 3 & & 3 & & 1 \\ \ldots & & \ldots & & \ldots & & \ldots \end{array}$$

Each entry satisfies the recurrence relation:

$$T(n, k) = T(n-1, k-1) + T(n-1, k), \tag{6}$$

where $T(n, k) = 0$ if $k < 0$ or $k > n$ [23].

2) Combinatorial Properties and Applications

1) **Binomial Coefficients:** Each row corresponds to the coefficients of the binomial expansion $(a + b)^n$ [24]:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k. \tag{7}$$

For example:

$$(a+b)^3 = \binom{3}{0}a^3 + \binom{3}{1}a^2 b + \binom{3}{2}ab^2 + \binom{3}{3}b^3.$$

2) **Counting Paths:** The values in Pascal's Triangle count the number of paths in a grid. For example, $\binom{n}{k}$ gives the number of ways to choose k steps in one direction from n total steps [25].
3) **Symmetry:** Pascal's Triangle is symmetric, such that $T(n, k) = T(n, n-k)$.
4) **Sum of Rows:** The sum of the entries in the n-th row is [26]:

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n. \tag{8}$$

5) **Connections to Fibonacci Sequence:** Summing the shallow diagonals of Pascal's Triangle produces Fibonacci numbers [8]. For example:

$$F(0) = 0,$$
$$F(1) = 1,$$
$$F(2) = 1 + 0 = 1,$$
$$F(3) = 1 + 2 = 3,$$
$$F(4) = 1 + 3 + 1 = 5, \ldots$$

6) **Probability Theory:** Pascal's Triangle is used in probability to calculate binomial distributions, where $\binom{n}{k}p^k(1-p)^{n-k}$ gives the probability of k successes in n trials [27].

3) Generalized Applications

Pascal's Triangle extends its significance to many fields, including number theory, modular arithmetic, and combinatorics. It also underpins algorithms for efficient computation of combinatorial values, making it a cornerstone of mathematical and computational research [28].

C. Derivation of the n-th Fibonacci Term Using Pascal's Triangle

1) How Fibonacci Numbers and Pascal's Triangle Are Related

Fibonacci numbers and Pascal's Triangle are deeply connected through combinatorial properties. By summing the shallow diagonals of Pascal's Triangle, we can generate Fibonacci numbers. This relationship highlights a fundamental connection between combinatorics and the Fibonacci sequence, providing an alternative way to compute these numbers without recursion or iteration.

2) Shallow Diagonal Lines of Pascal's Triangle to Make Fibonacci Numbers

In Pascal's Triangle, the shallow diagonals are formed by selecting numbers diagonally from the topmost row, extending downwards. Summing these diagonals produces Fibonacci numbers:

- The first diagonal ($\binom{0}{0}$) gives $F_1 = 0$.
- The second diagonal ($\binom{1}{0} + \binom{1}{1}$) gives $F_2 = 1$.
- The third diagonal ($\binom{2}{0} + \binom{2}{1}$) gives $F_3 = 2$.
- The fourth diagonal ($\binom{3}{0} + \binom{3}{1} + \binom{3}{2}$) gives $F_4 = 3$.

This pattern continues for higher diagonals, creating the Fibonacci sequence.

3) Supporting Diagram (adapted from Édouard Lucas work [3])

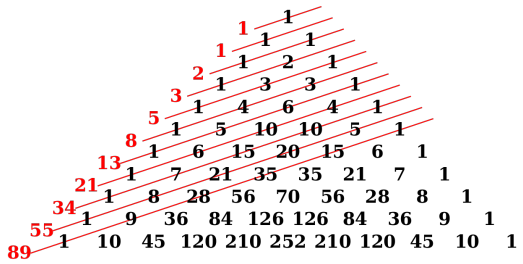

Figure: Illustration of Fibonacci numbers derived from Pascal's Triangle, adapted from [29]

4) Main Formula

The general formula for the n-th Fibonacci number derived from Pascal's Triangle is:

$$F_1 = 0, \quad \text{for } n = 1$$

and for $n > 1$:

$$F_N = \sum_{K=1}^{S} \frac{(N-K-1)!}{(K-1)!(N-2K)!},$$

Where:

$$S = \left\lfloor \frac{N}{2} \right\rfloor.$$

Here:

- F_N: The n-th Fibonacci number.
- K: Index iterating over the shallow diagonal.
- S: Floor of $N/2$, ensuring all valid terms in the diagonal are included.

5) How This Formula Was Derived

a) Step 1: Understanding Pascal's Triangle's Rows

Each row of Pascal's Triangle contains the binomial coefficients, defined as:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

b) Step 2: Observing Patterns in the Shallow Diagonals

By summing the entries along shallow diagonals of Pascal's Triangle, the Fibonacci numbers are formed. For example:

$$F_4 = \binom{3}{0} + \binom{3}{1} + \binom{3}{2} = 1 + 3 + 3 = 7.$$

c) Step 3: Generalizing for Any n

To compute F_N, the sum of terms in the shallow diagonal can be expressed as:

$$F_N = \sum_{K=1}^{S} \frac{(N-K-1)!}{(K-1)!(N-2K)!},$$

Where the limits of the sum depend on $S = \lfloor N/2 \rfloor$, ensuring all valid terms in the shallow diagonal are included.

d) Step 4: Validating the Formula

This formula was tested for small values of N, and the results matched the Fibonacci sequence, verifying its correctness.

V. Equation Derivation

A. First 20 Fibonacci Numbers

The first 20 Fibonacci numbers are presented in Table I. The highlighted values correspond to positions $3p+1$, where $p \geq 0$, which are all even numbers.

B. Fibonacci Sequence and Diagonal Connection

The Fibonacci sequence is deeply connected to Pascal's Triangle. By summing the shallow diagonals of Pascal's Triangle (with rows shifted appropriately), the Fibonacci sequence can be derived. For example:

- The first term corresponds to a single 1 in Pascal's Triangle.
- The second term corresponds to another single 1.
- The third term spans two numbers $(1, 1)$, which add to 2.
- The fourth term spans another two numbers $(1, 2)$, summing to 3.
- The fifth term spans three numbers $(1, 3, 1)$, summing to 5.

This pattern continues, generating the Fibonacci sequence.

TABLE I
FIRST 20 FIBONACCI NUMBERS

Position (N)	**Fibonacci Term** (F_N)	**Highlighted** ($3p+1$)
1	0	No
2	1	No
3	1	No
4	2	Yes
5	3	No
6	5	No
7	8	Yes
8	13	No
9	21	No
10	34	Yes
11	55	No
12	89	No
13	144	Yes
14	233	No
15	377	No
16	610	Yes
17	987	No
18	1597	No
19	2584	Yes
20	4181	No

C. Formula for the Number of Terms in Each Diagonal

The number of terms in each diagonal can be expressed as:

$$s = \begin{cases} \frac{N-1}{2}, & \text{if } N \text{ is odd,} \\ \frac{N}{2}, & \text{if } N \text{ is even.} \end{cases}$$

Alternatively, s can be written using the floor function:

$$s = \left\lfloor \frac{N}{2} \right\rfloor.$$

This formula calculates how many elements to sum along a diagonal in Pascal's Triangle for a given Fibonacci term.

D. Binomial Formula for Pascal's Triangle

The values in Pascal's Triangle are represented by the binomial coefficient formula:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Using this formula, we can calculate each diagonal element to determine the corresponding Fibonacci term. For example, to calculate the 12th Fibonacci term ($F_{12} = 89$), we sum the elements in the 11th diagonal of Pascal's Triangle.

E. Step 4: Example Calculation of the 12th Fibonacci Term

The 12th Fibonacci term (F_{12}) corresponds to the sum of the diagonal elements in the 11th row of Pascal's Triangle. The diagonal elements are:

$$1,\ 9,\ 28,\ 35,\ 15,\ 1$$

Summing these values yields:

$$1 + 9 + 28 + 35 + 15 + 1 = 89$$

This matches $F_{12} = 89$ in the Fibonacci sequence.

1) Detailed Calculations Using Binomial Coefficients

Using the binomial coefficient formula:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

We calculate each term in the diagonal of Pascal's Triangle as follows:

1) **First Term ($k = 1$):**

$$\frac{10!}{0!(10-0)!} = \frac{(12-1-1)!}{(1-1)!((12-1-1)-(1-1))!}.$$

Substituting values:

$$\frac{10!}{10!} = \frac{10!}{(1-1)!((12-1-1)-(1-1))!} = 1.$$

2) **Second Term ($k = 2$):**

$$\frac{9!}{1!(9-1)!} = \frac{(12-1-2)!}{(2-1)!((12-1-2)-(2-1))!}.$$

Substituting values:

$$\frac{9!}{8!} = \frac{(12-1-2)!}{1!(9-1)!} = 9.$$

3) **Third Term ($k = 3$):**

$$\frac{8!}{2!(8-2)!} = \frac{(12-1-3)!}{(3-1)!((12-1-3)-(3-1))!}.$$

Substituting values:

$$\frac{8!}{2!(6)!} = \frac{8 \cdot 7}{2 \cdot 1} = 28.$$

4) **Fourth Term ($k = 4$):**

$$\frac{7!}{3!(7-3)!} = \frac{(12-1-4)!}{(4-1)!((12-1-4)-(4-1))!}.$$

Substituting values:

$$\frac{7 \cdot 6 \cdot 5}{3 \cdot 2 \cdot 1} = \frac{7!}{3!(4)!} = 35.$$

5) **Fifth Term ($k = 5$):**

$$\frac{6!}{4!(6-4)!} = \frac{(12-1-5)!}{(5-1)!((12-1-5)-(5-1))!}.$$

Substituting values:

$$\frac{6 \cdot 5}{2 \cdot 1} = \frac{6!}{4!(2)!} = 15.$$

6) **Sixth Term ($k = 6$):**

$$\frac{5!}{5!(5-5)!} = \frac{(12-1-6)!}{(6-1)!((12-1-6)-(6-1))!}.$$

Substituting values:

$$\frac{5!}{5! \cdot 0!} = \frac{5!}{(6-1)!((12-1-6)-(6-1))!} = 1.$$

2) Summing the Diagonal Elements

The total sum of these diagonal elements is:

$$1 + 9 + 28 + 35 + 15 + 1 = 89.$$

This confirms that $F_{12} = 89$, as expected from the Fibonacci sequence.

F. Step 5: Generalized Formula for Fibonacci Terms

To generalize the calculation of the n-th Fibonacci number, we derive the following formula using binomial coefficients:

$$F_N = \sum_{K=1}^{S} \frac{(N-K-1)!}{(K-1)!(N-2K)!},$$

Where:

- $S = \left\lfloor \frac{N}{2} \right\rfloor$ is the floor of $\frac{N}{2}$,
- K iterates through the diagonal positions in Pascal's Triangle,
- $N > 1$ is the position of the Fibonacci number.

1) Derivation of the Formula

1. The binomial coefficient formula is given by:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

2. For the n-th Fibonacci term, we sum the diagonal elements of Pascal's Triangle, which are defined by the indices:

$$\binom{(N-1)-K}{K-1} = \frac{((N-1)-K)!}{(K-1)!(((N-1)-K)-(K-1))!}.$$

3. Substituting the above indices into the summation formula, we get:

$$F_N = \sum_{K=1}^{S} \frac{(N-K-1)!}{(K-1)!(N-2K)!}.$$

2) Validation of the Formula (Example: F_{12})

For $N = 12$:

- Calculate $S = \left\lfloor \frac{12}{2} \right\rfloor = 6$.
- Compute the terms for $K = 1$ to $K = 6$:

  1) For $K = 1$:

$$\frac{(12-1-1)!}{(1-1)!(12-2\cdot 1)!} = \frac{10!}{0!\cdot 10!} = 1$$

  2) For $K = 2$:

$$\frac{(12-2-1)!}{(2-1)!(12-2\cdot 2)!} = \frac{9!}{1!\cdot 8!} = 9$$

  3) For $K = 3$:

$$\frac{(12-3-1)!}{(3-1)!(12-2\cdot 3)!} = \frac{8!}{2!\cdot 6!} = 28$$

- Sum all terms: $1 + 9 + 28 + 35 + 15 + 1 = 89$.

Thus, $F_{12} = 89$, as expected.

G. Theoretical Comparisons:- Comparison of the Mathematical Basis of Édouard Lucas Method with Traditional Algorithms

Édouard Lucas's method for deriving Fibonacci numbers using Pascal's Triangle and traditional algorithms (recursive, iterative, and matrix-based) rely on fundamentally different mathematical principles. Below is a detailed comparison:

1) Mathematical Basis

a) Édouard Lucas Method (Combinatorial Perspective)

- **Core Principle:** Édouard Lucas's method leverages the combinatorial properties of Pascal's Triangle to compute Fibonacci numbers. Specifically, Fibonacci numbers are obtained by summing entries along shallow diagonals of Pascal's Triangle.
- **Key Formula:** The Fibonacci numbers can be computed as:

$$F_N = \sum_{K=1}^{S} \frac{(N-K-1)!}{(K-1)!(N-2K)!},$$

  where $S = \left\lfloor \frac{N}{2} \right\rfloor$, and factorials are used to calculate binomial coefficients.
- **Structural Insight:** The method establishes an explicit relationship between Fibonacci numbers and binomial coefficients, highlighting a combinatorial foundation for Fibonacci sequence generation.
- **Mathematical Elegance:**
  - Provides a visually intuitive way of connecting Fibonacci numbers and Pascal's Triangle.
  - Demonstrates that Fibonacci numbers emerge naturally from a pre-existing combinatorial structure.

b) Traditional Algorithms

- **Recursive Approach (Definitional Basis):**
  - **Core Principle:** The recursive approach is based directly on the recurrence relation defining the Fibonacci sequence:

$$F(n) = \begin{cases} 0, & \text{if } n = 0, \\ 1, & \text{if } n = 1, \\ F(n-1) + F(n-2), & \text{if } n \geq 2. \end{cases}$$

  - **Key Strengths:** - Closely follows the mathematical definition of Fibonacci numbers. - Easy to implement but computationally expensive due to repeated subproblem evaluations.
- **Iterative Approach (Sequential Accumulation):**
  - **Core Principle:** The iterative approach eliminates recursion by accumulating the sequence iteratively:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}.$$

  - **Key Strengths:** - Computationally efficient with $O(n)$ time complexity. - Requires only $O(1)$ space, storing the last two terms of the sequence.
- **Matrix-Based Algorithm (Linear Algebraic Basis):**

– **Core Principle:** Fibonacci numbers can be computed using matrix exponentiation, which exploits the relationship:

$$\begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n.$$

– **Key Strengths:** - Achieves logarithmic time complexity $O(\log n)$ via fast matrix exponentiation. - Particularly useful for large n.

2) Computational Perspective

To compare the computational aspects of Édouard Lucas's method and traditional algorithms, Table II summarizes key differences in mathematical foundations, time complexity, space complexity, and typical use cases.

3) Applicability

a) Édouard Lucas Method:

- Useful for understanding and teaching the combinatorial relationships between Fibonacci numbers and Pascal's Triangle.
- Not efficient for large-scale computations due to factorial calculations.

b) Traditional Algorithms:

- Preferred in real-world applications where performance is critical.
- Iterative and matrix-based approaches are particularly suited for resource-constrained environments or large n.

VI. Algorithmic Implementation

A. Recursive Fibonacci Algorithm

The recursive Fibonacci algorithm is a straightforward implementation of the Fibonacci sequence based on its mathematical definition. It calculates each Fibonacci number by recursively summing the two preceding numbers.

The pseudocode for the recursive Fibonacci algorithm is as follows:

Details of the Algorithm:

- **Base Cases:**
  - If $n = 0$, the function returns 0.
  - If $n = 1$, the function returns 1.
- **Recursive Step:**
  - For $n > 1$, the function calculates:

$$Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)$$

- **Time Complexity:** $O(2^n)$, due to repeated recalculations of the same subproblems.

```
FUNCTION Fibonacci(n):
    IF n = 0:
      RETURN 0
    ELSE IF n = 1:
      RETURN 1
    ELSE:
      RETURN Fibonacci(n-1) +
          Fibonacci(n-2)
END FUNCTION
```

Fig. 1. Pseudocode for Recursive Fibonacci Algorithm

- **Space Complexity:** $O(n)$, proportional to the depth of the recursion stack.

While simple and intuitive, this algorithm is inefficient for large n due to redundant computations. Optimized approaches such as memoization or iterative methods are recommended for practical use.

B. Iterative Fibonacci Algorithm

The iterative Fibonacci algorithm efficiently computes Fibonacci numbers by sequentially updating two accumulators. This method avoids the excessive overhead and stack usage associated with the recursive approach.

The pseudocode for the iterative Fibonacci algorithm is as follows:

```
FUNCTION IterativeFibonacci(n):
    INITIALIZE a = 0, b = 1, c = 0
    IF n = 0:
      RETURN a
    ELSE IF n = 1:
      RETURN b
    ELSE:
      FOR i FROM 2 TO n:
        c = a + b
        a = b
        b = c
    END FOR
    RETURN c
END FUNCTION
```

Fig. 2. Pseudocode for Iterative Fibonacci Algorithm

Details of the Algorithm:

- **Base Cases:**
  - If $n = 0$, the function returns 0.

TABLE II
Comparison of Édouard Lucas Method with Recursive, Iterative, and Matrix Exponentiation Algorithms

Aspect	Édouard Lucas Method	Recursive Algorithms	Iterative Algorithms	Matrix Exponentiation
Mathematical Foundation	Combinatorial (Pascal's Triangle)	Recurrence relation	Sequential computation	Matrix algebra
Time Complexity	$O(n^2)$ (due to factorial computations)	Recursive: $O(2^n)$	$O(n)$	$O(\log n)$
Space Complexity	$O(1)$	Recursive: $O(n)$	$O(1)$	$O(\log n)$
Use Case	Structural exploration	Best for theoretical analysis	Efficient for large n	Fast computation with matrix power

- If $n = 1$, the function returns 1.

- **Iterative Step:**
  - The function uses a loop to calculate each Fibonacci number up to n without repeated function calls, starting from 2 up to n:

    On each iteration, $c = a+b$, followed by $a = b$ and $b = c$

- **Time Complexity:** $O(n)$, which is linear and much more efficient than the recursive method.
- **Space Complexity:** $O(1)$, as it uses a constant amount of space regardless of the input size.

This algorithm is highly efficient for computing large Fibonacci numbers and is recommended for practical applications where performance is critical.

C. Matrix Exponentiation Fibonacci Algorithm

The **Matrix Exponentiation Fibonacci Algorithm** efficiently computes Fibonacci numbers using matrix exponentiation. Unlike the iterative approach, which has $O(n)$ time complexity, this method reduces the complexity to $O(\log n)$ by leveraging fast exponentiation techniques.
The pseudocode for the matrix exponentiation Fibonacci algorithm is as follows:

```
FUNCTION MatrixFibonacci(n):
    IF n = 0:
       RETURN 0
    ELSE:
       DEFINE MATRIX F = [1 1; 1 0]
       CALL MatrixPower(F, n-1)
       RETURN F[0][0]

FUNCTION MatrixPower(F, p):
    DEFINE MATRIX M = [1 1; 1 0]
    IF p = 0 OR p = 1:
       RETURN
    CALL MatrixPower(F, p // 2)
    MULTIPLY F by F
    IF p is ODD:
       MULTIPLY F by M

FUNCTION MatrixMultiply(A, B):
    INITIALIZE RESULT MATRIX C = [0 0; 0 0]
    FOR i FROM 0 TO 1:
       FOR j FROM 0 TO 1:
          C[i][j] = A[i][0] × B[0][j] +
A[i][1] × B[1][j]
    COPY C INTO A

END FUNCTION
```

Fig. 3. Pseudocode for Matrix Exponentiation Fibonacci Algorithm

Details of the Algorithm:

- **Base Case:**
  - If $n = 0$, the function returns 0.
- **Exponentiation Step:**
  - Instead of computing Fibonacci numbers iteratively, we raise F to the power $(n-1)$, which gives:

$$F^n = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix}$$

  - The result $F[0][0]$ is the n-th Fibonacci number.
  - The power function computes $F^{(n-1)}$ efficiently in $O(\log n)$ time by:
    - Recursively squaring F when n is even.
    - Multiplying by an extra F when n is odd.
- **Time Complexity:** $O(\log n)$, which is significantly faster than the iterative $O(n)$ approach.
- **Space Complexity:** $O(1)$, as it only requires a fixed-size 2×2 matrix.

This method is highly efficient for computing large Fibonacci numbers and is widely used in performance-critical applications.

D. Pascal's Triangle Method (E Lucas Adaptation)

The Lucas method for Fibonacci numbers employs an interesting approach using binomial coefficients to calculate Fibonacci numbers. This method is based on combinatorial identities and offers a different computational perspective compared to traditional recursive or iterative Fibonacci algorithms. The pseudocode for the Lucas method is provided below:
Details of the Algorithm:

- **Mathematical Foundation:** The Édouard Lucas method leverages combinatorial mathematics by utilizing binomial coefficients, which are integrally linked to the combinatorial structures found in Pascal's triangle. Each Fibonacci number is expressed as a sum of binomial coefficients, demonstrating a deep mathematical relationship between number theory and combinatorics:

$$F_n = \sum_{k=0}^{\lfloor \frac{n}{2} \rfloor} \binom{n-k-1}{k-1}$$

  This formula indicates that each Fibonacci number can be derived from summing specific entries of Pascal's triangle, which represents the number of ways to choose k elements from $n-k$ elements.
- **Time Complexity:** The time complexity of the Édouard Lucas method largely depends on the efficient computation of binomial coefficients. Typically, the computation of a single binomial coefficient is $O(n)$ due to the iterative multiplication and division required to calculate $\binom{n}{k}$. Considering the need to compute approximately $\frac{n}{2}$ coefficients for each Fibonacci number calculation, the overall complexity can be approximated as $O(n^2)$. However, with optimizations such as memoization or

```
FUNCTION binom_coefficient(h, p):
    IF p > h:
      RETURN 0
    IF p == 0 OR p == h:
      RETURN 1
    IF p > h - p:
      p = h - p
    result = 1
    FOR i = 1 TO p:
      result *= (h - i + 1)
      result /= i
    RETURN result

FUNCTION fibo_lucas(N, sum POINTER):
    sum = 1
    s = N / 2
    K = 2
    FOR i = 1 TO s - 1:
      n = N - K - 1
      r = K - 1
      t = N - 2 * K
      IF t > r:
        sum += binom_coefficient(n, r)
      ELSE:
        sum += binom_coefficient(n, t)
      K++
END FUNCTION
```

Fig. 4. Pseudocode for Edouard Lucas method of calculating Fibonacci using optimized binomial coefficients

using iterative dynamic programming approaches to precompute binomial coefficients, this complexity can be significantly reduced.

- **Space Complexity:** The space complexity of this method is $O(1)$, as it uses a constant amount of space to store intermediate results and the final output. This efficiency stems from the fact that the algorithm only needs to maintain a few variables for binomial coefficients and the sum of these coefficients, independent of the size of N, except for the minimal stack space required during the iterative computation of coefficients.
- **Use Case:** The Lucas method is particularly beneficial for applications requiring the computation of Fibonacci numbers in environments where iterative or recursive methods are computationally prohibitive due to their higher time complexities or excessive memory use in large-scale scenarios. Its utility shines in mathematical analysis and theoretical computer science, where understanding the interconnections between different branches of mathematics can lead to more efficient algorithms.

This enhanced understanding of the Lucas method not only clarifies the computational intricacies involved but also highlights its potential for providing more efficient solutions for calculating Fibonacci numbers, especially in large-scale or theoretical contexts.

E. Programming Considerations and Potential Optimizations

1) Lucas Method (Binomial Coefficient Approach)

a) Algorithm Structure

The Lucas method relies on binomial coefficients for computation. The two core functions are:

- **`binom_coefficient(h, p)`**: Computes $\binom{h}{p}$ iteratively to avoid factorial overflow.
- **`fibo_lucas(N, sum_pointer)`**: Uses a loop to accumulate binomial coefficients and derive the Fibonacci number.

b) Programming Considerations

- **Loop Overhead:** Multiple calls to `binom_coefficient` inside a loop lead to redundant calculations.
- **Handling Large Values:** Factorial-based calculations may introduce precision issues.
- **Symmetric Computation:** The check $p > h - p$ minimizes redundant calculations by leveraging binomial coefficient symmetry.

c) Potential Optimizations

- **Memoization:** Store previously computed binomial values in a lookup table.
- **Iterative Binomial Formula:** Compute $\binom{h}{p}$ using:

$$\binom{h}{p} = \frac{h!}{p!(h-p)!}$$

  while avoiding large intermediate values.
- **Precomputed Pascal's Triangle:** Storing Pascal's triangle up to N reduces redundant computations.

2) Recursive Approach

a) Algorithm Structure

The Fibonacci sequence follows:

$$F(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F(n-1) + F(n-2), & n \geq 2 \end{cases}$$

b) Programming Considerations

- **Time Complexity:** The naive recursion is $O(2^n)$, leading to redundant calculations.
- **Memory Usage:** High recursion depth increases stack memory usage.

c) Potential Optimizations

- **Memoization:** Store results of previously computed Fibonacci numbers.
- **Tail Recursion:** Transform the function into an iterative form to prevent stack overflow.

- **Matrix Exponentiation:** Represent Fibonacci calculations as:

$$\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} F(1) \\ F(0) \end{bmatrix}$$

reducing complexity to $O(\log n)$.

3) Iterative Approach (Dynamic Programming)

a) Algorithm Structure

A bottom-up iterative Fibonacci implementation:

```
FUNCTION fibonacci_iter(N):
    IF N == 0: RETURN 0
    IF N == 1: RETURN 1
    a = 0, b = 1
    FOR i = 2 TO N:
        temp = a + b
        a = b
        b = temp
    RETURN b
```

b) Programming Considerations

- **Time Complexity:** $O(n)$, more efficient than recursion.
- **Memory Usage:** Uses only $O(1)$ space.

c) Potential Optimizations

- **Matrix Exponentiation:** Reduces time complexity to $O(\log n)$.
- **Bitwise Operations:** Enhances performance using binary arithmetic.

4) Comparative Analysis

Method	Time Complexity	Space Complexity	Optimizations
Lucas Method	$O(n^2)$	$O(1)$	Memoization, Pascal's triangle
Recursive	$O(2^n)$	$O(n)$	Memoization, matrix exponentiation
Iterative	$O(n)$	$O(1)$	Matrix exponentiation, bitwise ops
Matrix Exponentiation	$O(\log n)$	$O(1)$	Fast exponentiation, matrix multiplication

TABLE III
COMPARISON OF FIBONACCI COMPUTATION METHODS

Each method has trade-offs in terms of computational efficiency and memory. The Lucas approach provides a combinatorial perspective, while iterative and matrix-based methods optimize for performance.

VII. METHODOLOGY

A. Null and Alternative Hypotheses

1) Hypotheses for Efficiency Comparison

To compare the efficiency of iterative, Lucas, recursive, and matrix exponentiation methods for calculating Fibonacci numbers, we define the following hypotheses:

- **Hypothesis 1 (Time Complexity)**: The **matrix exponentiation** approach outperforms the iterative, Lucas, and recursive methods in execution time, achieving a logarithmic complexity $O(\log n)$. The **iterative** method follows with $O(n)$, while the **Lucas method** remains at $O(n^2)$ due to multiple binomial coefficient calculations. The **recursive** approach exhibits exponential growth at $O(2^n)$.
- **Hypothesis 2 (Space Complexity)**: The **matrix exponentiation, iterative, and Lucas methods** require constant space $O(1)$, while the recursive approach consumes $O(n)$ space due to function call stack usage.
- **Hypothesis 3 (Computational Overhead)**: The Lucas method, while leveraging combinatorial properties, introduces additional arithmetic overhead compared to the iterative and matrix exponentiation approaches, making it less practical for large-scale computations. **Matrix exponentiation, on the other hand, reduces redundant calculations using fast exponentiation techniques, making it the most efficient method for large values of n.**
- **Hypothesis 4 (Numerical Stability)**: The iterative and matrix exponentiation approaches provide stable numerical results for large n, whereas recursive computations may suffer from stack overflow. The Lucas method may introduce rounding errors depending on the precision of factorial division in binomial coefficient calculations.

2) Null Hypothesis (H_0)

The **matrix exponentiation** method for calculating Fibonacci numbers is not significantly more efficient than the iterative, recursive, and Lucas methods in terms of both time complexity and memory usage.

- **Time Complexity:**

$$H_0 : T_{\text{matrix}} \geq T_{\text{iterative}} \geq T_{\text{Lucas}} \geq T_{\text{recursive}},$$

Where T_{matrix}, $T_{\text{iterative}}$, T_{Lucas}, and $T_{\text{recursive}}$ represent the time complexities of the matrix exponentiation, iterative, Lucas, and recursive methods, respectively.

- **Memory Usage:**

$$H_0 : M_{\text{matrix}} \geq M_{\text{iterative}} \geq M_{\text{Lucas}} \geq M_{\text{recursive}},$$

Where M_{matrix}, $M_{\text{iterative}}$, M_{Lucas}, and $M_{\text{recursive}}$ represent the memory usage of the matrix exponentiation, iterative, Lucas, and recursive methods, respectively.

3) Alternative Hypothesis (H_A)

The **matrix exponentiation** method for calculating Fibonacci numbers is significantly more efficient than the iterative, recursive, and Lucas methods in terms of both time complexity and memory usage.

- **Time Complexity:**

$$H_A : T_{\text{matrix}} < T_{\text{iterative}} < T_{\text{Lucas}} < T_{\text{recursive}},$$

Indicating that the matrix exponentiation method is the fastest, followed by the iterative approach. The Lucas method, while slower, still improves over direct recursion.

- **Memory Usage:**

$$H_A : M_{\text{matrix}} \leq M_{\text{iterative}} \leq M_{\text{Lucas}} \leq M_{\text{recursive}},$$

Indicating that the matrix exponentiation method consumes the least memory, with iterative and Lucas methods having constant space complexity similar to it. Recursive methods, on the other hand, require additional memory due to function call stacks.

4) Rationale

The null hypothesis assumes no significant computational advantage for the **matrix exponentiation, iterative, or Lucas methods**. In contrast, the alternative hypothesis aligns with the expectation that the **matrix exponentiation method outperforms all other methods** due to:

- **Improved Time Complexity:**
  $O(\log n)$ for matrix exponentiation, $O(n)$ for the iterative method, $O(n^2)$ for the Lucas method, $O(2^n)$ for naive recursion.
  The **matrix exponentiation method** significantly reduces computation time by leveraging fast exponentiation techniques, making it the most optimal approach.
- **Better Space Complexity:**
  $O(1)$ for matrix exponentiation, iterative, and Lucas methods, $O(n)$ for recursion.
  Matrix exponentiation achieves constant space complexity while significantly improving performance, making it the best choice for large Fibonacci computations.

VIII. Computational Complexity Analysis

A. Recursive Method

1) Time Complexity Analysis

The recursive Fibonacci algorithm follows the recurrence relation:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

Since each function call spawns two new recursive calls, the recursion tree grows exponentially. The number of calls at each level approximately follows the Fibonacci sequence itself, leading to an exponential time complexity.

Using the recurrence relation and solving via the Master Theorem or recurrence tree method, we get:

$$T(n) = O(2^n)$$

Explanation:

- The recursion depth is $O(n)$, as we keep reducing n until we reach the base case.
- At each level of recursion, the number of function calls doubles.
- This results in an exponential growth in execution time.

Thus, the recursive approach is highly inefficient for large n, as it performs redundant calculations.

2) Space Complexity Analysis

The space complexity of the recursive Fibonacci function is determined by the maximum depth of the recursive call stack.

- Each recursive call adds a new function to the call stack until it reaches the base case.
- The depth of recursion is $O(n)$, meaning at worst, there are n function calls stored in memory at the same time.
- Once the base case is reached, the function calls return and the stack starts unwinding.

Thus, the space complexity is:

$$S(n) = O(n)$$

Summary:

- **Time Complexity:** $O(2^n)$ (Exponential growth due to duplicate subproblems)
- **Space Complexity:** $O(n)$ (Due to recursive call stack)

Although the recursive approach is simple and intuitive, its inefficiency makes it impractical for large values of n. Optimizations like **memoization** or **matrix exponentiation** are recommended for performance improvements.

B. Iterative Approach

1) Time Complexity Analysis

The iterative Fibonacci algorithm computes Fibonacci numbers using a loop, updating two accumulators at each step. The algorithm runs as follows:

- It initializes two variables, $a = 0$ and $b = 1$.
- It iterates from 2 to n, updating the values using:

$$c = a + b, \quad a = b, \quad b = c$$

- The loop executes $O(n)$ times, performing a constant amount of work in each iteration.

Thus, the total time complexity is:

$$T(n) = O(n)$$

Explanation:

- The algorithm performs a single pass through the loop from 2 to n, leading to a linear time complexity.
- Unlike the recursive approach, there are no redundant computations or repeated function calls.
- The number of operations scales directly with n, making it much more efficient than the exponential recursive method.

2) Space Complexity Analysis

The iterative Fibonacci algorithm only uses a **constant** amount of extra space.

- It maintains only a few integer variables (a, b, and c).
- The algorithm does not use recursion, so there is no additional stack space needed.
- Regardless of the input size n, the space used remains the same.

Thus, the space complexity is:

$$S(n) = O(1)$$

Summary:

- **Time Complexity:** $O(n)$ (Linear growth with input size)
- **Space Complexity:** $O(1)$ (Constant space usage)

Compared to the recursive approach, the iterative method is significantly more efficient in both time and space, making it ideal for computing Fibonacci numbers in practical applications.

C. Matrix Exponentiation

1) Time Complexity Analysis

The Matrix Exponentiation approach computes Fibonacci numbers using the property:

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

Where matrix exponentiation is performed using **fast exponentiation (exponentiation by squaring)**, reducing the number of multiplications significantly.

Steps of the Algorithm:

- We raise the transformation matrix $T = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ to the power $(n-1)$ using exponentiation by squaring.
- The exponentiation by squaring algorithm reduces the number of matrix multiplications from $O(n)$ (naïve exponentiation) to $O(\log n)$.
- Matrix multiplication is performed in **constant time** $O(1)$ per step (for 2×2 matrices).

Thus, the total time complexity is:

$$T(n) = O(\log n)$$

Explanation:

- Exponentiation by squaring requires only $O(\log n)$ multiplications.
- Each multiplication involves **constant-time** operations on a **2×2 matrix**.
- This makes matrix exponentiation much more efficient than both recursive $O(2^n)$ and iterative $O(n)$ methods.

2) Space Complexity Analysis

The space complexity of the Matrix Exponentiation method is determined by the number of extra variables used:

- The algorithm only maintains a few **fixed-size 2×2 matrices**.
- No additional recursion or stack memory is required.
- Since the matrix size remains constant **regardless of** n, the space complexity remains constant.

Thus, the space complexity is:

$$S(n) = O(1)$$

Summary:

- **Time Complexity:** $O(\log n)$ (Fast exponentiation significantly reduces computation time)
- **Space Complexity:** $O(1)$ (Constant space, as only a few matrices are stored)

The **Matrix Exponentiation** approach is one of the most efficient methods for computing large Fibonacci numbers, as it provides a logarithmic time complexity while maintaining constant space usage.

D. Edouard Lucas Method

1) Computational Complexity of the E Lucas Method

The **E Lucas method** computes Fibonacci numbers using combinatorial properties, specifically Pascal's triangle and binomial coefficients:

$$F_n = \sum_{k=0}^{\lfloor n/2 \rfloor} \binom{n-k}{k}$$

Time Complexity Analysis:

- The main computational step involves evaluating **binomial coefficients** $\binom{n-k}{k}$.
- Computing a single binomial coefficient using factorials requires $O(n)$, leading to an overall complexity of:

$$T(n) = O(n^2)$$

when calculated using the factorial definition.

- Using **Pascal's triangle**, the binomial coefficients can be computed more efficiently in $O(n)$ time using dynamic programming.
- Despite optimizations, this method is still slower than **iterative** $O(n)$ **and matrix exponentiation** $O(\log n)$ approaches.

Space Complexity Analysis:

- If computed using **Pascal's triangle**, it requires storing binomial coefficients, leading to a **space complexity of** $O(n)$.
- If computed using recursive factorial calculations, additional function calls lead to a worst-case space complexity of $O(n)$.
- Overall, optimized implementations can achieve $O(1)$ **space complexity** by computing coefficients iteratively without storing the entire triangle.

2) Implications for Large-Scale Inputs

Challenges:

- The **quadratic time complexity** ($O(n^2)$) makes this method inefficient for large Fibonacci indices n.
- **Factorial computations** involved in binomial coefficient calculation can lead to numerical overflow for very large n.
- **Memory consumption** increases with larger Pascal's triangle tables, especially in naive implementations.

Advantages:

- The E Lucas method provides a combinatorial perspective on Fibonacci sequences.
- It allows **direct computation** without requiring iterative or recursive traversal of the sequence.

- Suitable for **mathematical analysis** rather than practical large-scale Fibonacci computation.

Comparison with Other Methods:

- **Slower** than iterative $O(n)$ and matrix exponentiation $O(\log n)$.
- **More complex** than simple iterative approaches.
- **Useful in theoretical contexts** but impractical for real-world applications requiring fast Fibonacci computation.

Summary:

- **Time Complexity:** $O(n^2)$ (Due to binomial coefficient computation)
- **Space Complexity:** $O(n)$ (Using Pascal's triangle, reducible to $O(1)$)
- **Best Use Case:** Theoretical applications, combinatorial proofs, and small input sizes.

E. Scalability

1) Scalability Analysis of Fibonacci Computation Methods

Computing large Fibonacci numbers efficiently is crucial for applications in cryptography, numerical computing, and theoretical research. The scalability of different methods is primarily determined by their time and space complexity.

1. Recursive Method:

- **Time Complexity:** $O(2^n)$ (Exponential growth)
- **Space Complexity:** $O(n)$ (Recursive call stack)
- **Scalability:** *Poor* – The recursive approach suffers from extreme inefficiency due to redundant calculations. It quickly becomes impractical for values of $n > 40$ due to exponential time complexity.
- **Limiting Factors:** High computational overhead, deep recursion stack leading to memory overflow.

2. Iterative Method:

- **Time Complexity:** $O(n)$ (Linear)
- **Space Complexity:** $O(1)$ (Constant memory usage)
- **Scalability:** *Moderate* – The iterative approach scales much better than recursion but still has a linear time complexity. Computing Fibonacci numbers for extremely large n (e.g., $n > 10^6$) becomes slow.
- **Limiting Factors:** Processing time increases linearly, making it unsuitable for extremely large inputs.

3. Matrix Exponentiation Method:

- **Time Complexity:** $O(\log n)$ (Logarithmic)
- **Space Complexity:** $O(1)$ (Constant)
- **Scalability:** *High* – The matrix exponentiation approach is highly scalable, making it suitable for computing Fibonacci numbers for large values of n (e.g., $n > 10^9$).
- **Limiting Factors:** Requires efficient matrix multiplication implementation, but still remains one of the fastest methods available.

4. E Lucas Method:

- **Time Complexity:** $O(n^2)$ (Quadratic)
- **Space Complexity:** $O(n)$ (Using Pascal's triangle, reducible to $O(1)$)
- **Scalability:** *Low* – While theoretically interesting, the E Lucas method is not practical for large n due to its quadratic time complexity.
- **Limiting Factors:** Binomial coefficient computations introduce unnecessary computational overhead, making it unsuitable for large-scale applications.

2) Comparison of Scalability Across Methods

Method	Time Complexity	Space Complexity	Scalability
Recursive	$O(2^n)$	$O(n)$	Poor (exponential growth makes it infeasible)
Iterative	$O(n)$	$O(1)$	Moderate (works well for moderately large n)
Matrix Exponentiation	$O(\log n)$	$O(1)$	High (efficient for very large n)
E Lucas Method	$O(n^2)$	$O(n)$ (or $O(1)$ with optimizations)	Low (quadratic time complexity makes it slow)

TABLE IV
COMPARISON OF SCALABILITY FOR FIBONACCI COMPUTATION METHODS

Conclusion: The **recursive** method is highly inefficient and does not scale well. The **iterative** method is more practical but still has linear growth, making it unsuitable for extremely large Fibonacci numbers. **Matrix exponentiation** is the most scalable method, with logarithmic time complexity and constant space requirements, making it ideal for computing very large Fibonacci terms. The **E Lucas method**, while mathematically insightful, suffers from quadratic time complexity and is not recommended for large-scale applications.

IX. EMPIRICAL EVALUATION

A. Experimental Setup

To evaluate the performance of different Fibonacci computation methods, an experiment was conducted to measure execution time for various input sizes. The setup includes details about the testing environment, tools used, and methodology.

1) Testing Environment

The experiments were conducted on a **MacBook Pro M1** without the need for a specialized computing environment. The following system specifications were used:

- **Device:** MacBook Pro with Apple M1 Chip
- **Operating System:** macOS
- **Processor:** 8-core Apple M1
- **Memory:** 16GB Unified RAM

2) Tools Used

The following tools and libraries were used in the experiment:

- **Code Editor:** ZED Code Editor
- **Programming Language:** C
- **Compiler:** GCC (GNU Compiler Collection)
- **Compilation Command:**

```
gcc file_name.c -o file_name
```

- **Libraries Used:**

- `#include <stdio.h>` (Standard Input/Output)
- `#include <time.h>` (Time Measurement)
- `#include <stdint.h>` (Fixed-Width Integer Types)

3) Methodology

The experiment was designed to systematically measure execution time for different Fibonacci computation methods: **Iterative, Recursive, Matrix Exponentiation, and E Lucas Method**. The following methodology was followed:

1) **User Input:** The program accepts an integer n as input, representing the maximum Fibonacci term to compute.
2) **Loop Execution:** The experiment iterates from 1 to n, computing Fibonacci numbers incrementally.
3) **Time Measurement:**
   - A high-resolution timer (`clock_gettime`) is used to capture execution time before and after function execution.
   - The difference between start and end times is recorded in both seconds and nanoseconds.
4) **Data Logging:** The results are stored in a CSV file with the format:

```
Index, Fibonacci Number, Execution Time (seconds), Execution Time (nanoseconds)
```

5) **Function Calls:** Each Fibonacci computation method is executed via a function call:

```
fib(i)
```

   where `fib` represents the specific Fibonacci computation function being tested.
6) **Output Storage:** The computed Fibonacci numbers and their respective execution times are logged in a CSV file for further analysis.

4) Code Reference

The full implementation of each Fibonacci computation method is provided in the **Appendix** (see Section A). The pseudocode for each method has been previously described in their respective sections.

References:

- Iterative Fibonacci Algorithm (Section A-A)
- Recursive Fibonacci Algorithm (Section A-B)
- Matrix Exponentiation Fibonacci Algorithm (Section A-C)
- E Lucas Method (Section A-D)

5) Expected Outcome

- The execution time for small values of n is expected to be minimal.
- As n increases, the recursive method should exhibit exponential growth in execution time, while matrix exponentiation should scale logarithmically.
- The CSV file will serve as a basis for analyzing and comparing the efficiency of different methods.

Conclusion: This experimental setup ensures a controlled and systematic evaluation of different Fibonacci computation methods. By recording execution times in a structured manner, the results can be used for performance comparison and scalability analysis.

B. Test Cases and Dataset

A structured dataset will be created using CSV files to ensure a fair and unbiased comparison of the different Fibonacci algorithms. Each algorithm will be executed multiple times to mitigate random fluctuations in execution time, and the results will be averaged for accuracy.

1) Dataset Collection Approach

- **Multiple Runs for Accuracy:**
  - Each algorithm will be executed **three times** for every Fibonacci number computation.
  - The average execution time will be stored in a **CSV file** to ensure data smoothness and avoid anomalies.
- **Range of Fibonacci Computation (n):**
  - The **iterative**, **matrix exponentiation**, and **E Lucas** methods will compute Fibonacci numbers up to $n = 200$ to measure performance over a large dataset.
  - Due to the **exponential time complexity** of recursion, the **recursive approach** will initially compute Fibonacci numbers up to $n = 40$.
  - If the execution time remains reasonable, testing will continue up to $n = 50$. However, testing will stop at the feasible limit if performance degrades significantly.
- **Metrics Captured:**
  - **Index (n):** The Fibonacci sequence index.
  - **Fibonacci Number:** The actual computed Fibonacci value.
  - **Execution Time (seconds):** The time taken to compute Fibonacci(n) in seconds.
  - **Execution Time (nanoseconds):** A high-precision measurement of execution time.

2) Additional Considerations

- **System Variability Control:** Tests will be conducted under similar system conditions to avoid fluctuations due to CPU load.
- **Handling Large Values:** Since Fibonacci values grow exponentially, computations will be handled using `long long int` or higher precision types where required.
- **Stopping Criteria for Recursion:** If execution time exceeds a reasonable threshold for recursion, testing will halt beyond that n value.

C. Performance Benchmarks

This section presents the experimental results, including execution time and memory usage for different Fibonacci computation methods. The results are supported by graphs to provide a clear comparison.

1) Dataset Reference

For complete raw datasets, refer to the following links:

- **Raw Dataset:** Click here
- **Processed Dataset:** Click here

2) Memory Usage

a) Iterative Fibonacci (fib_itera.c)

Memory Usage Breakdown:

- Uses three `long long` variables: `prev1`, `prev2`, `fib`.
- Each `long long` occupies 8 bytes.
- Additional variables: `int n` (4 bytes) and loop counter `int i` (4 bytes).

Total Memory Usage (Stack):

$$8 + 8 + 8 + 4 + 4 = 32 \text{ bytes} \approx 32B$$

Final Space Complexity: $O(1)$ (fixed memory usage of 32 bytes).

b) Recursive Fibonacci (fib_rec.c)

Memory Usage Breakdown:

- Each recursive function call stores:
  - `int n` (4 bytes)
  - Return address & function metadata (8 bytes)
- Maximum recursion depth is $O(n)$.

Total Memory Usage (Stack): For each function call:

$$4 + 8 = 12B$$

For n recursive calls:

$$12n \text{ bytes}$$

For $n = 30$:

$$12 \times 30 = 360B$$

For $n = 100$:

$$12 \times 100 = 1.2 \text{ KB}$$

Final Space Complexity: $O(n)$ (linear growth, possible stack overflow for large n).

c) Matrix Exponentiation Fibonacci (matrix_expo.c)

Memory Usage Breakdown:

- Stores two 2×2 matrices:
  - `F[2][2]` (64 bytes)
  - `M[2][2]` (64 bytes)
- Recursion depth is $O(\log n)$.
- Each recursive call has:
  - `long long n` (8 bytes)
  - Return address (8 bytes)

Total Memory Usage (Stack): For each recursive call:

$$8 + 8 = 16B$$

For $O(\log n)$ recursive calls:

$$16 \log_2 n \text{ bytes}$$

For $n = 1024$:

$$16 \times 10 = 160B$$

Final Space Complexity: $O(\log n)$ (efficient for large n).

d) E Lucas Fibonacci (fibbo_lucas.c)

Memory Usage Breakdown:

- Uses a few integer variables:
  - `int s, K, n, r, t` (5 × 4 bytes = 20B)
  - `long double sum` (16B)
- Calls `binomial_coefficient()`, which uses:
  - A single loop (no recursion).
  - Stores `long long result` (8B).

Total Memory Usage (Stack):

$$20 + 16 + 8 = 44B$$

Final Space Complexity: $O(1)$ (fixed 44 bytes).

e) Final Memory Usage Comparison

TABLE V
MEMORY USAGE COMPARISON OF FIBONACCI ALGORITHMS

Algorithm	Space Complexity	Memory Usage (bytes)
Iterative Fibonacci	$O(1)$	32B (fixed)
Recursive Fibonacci	$O(n)$	$12n$B (linear growth)
Matrix Exponentiation Fibonacci	$O(\log n)$	$16 \log_2 n$ B
Binomial Coefficient Fibonacci	$O(1)$	44B (fixed)

Best Choice for Low Memory Usage:

- **Iterative Fibonacci** ($O(1)$, 32B) is the best for minimizing memory.
- **Matrix Exponentiation** ($O(\log n)$) is efficient but uses up to 160B for $n = 1024$.
- **Recursive Fibonacci** ($O(n)$) has high stack usage, leading to stack overflow risks.

3) Graphical Analysis

a) Iterative Algorithm Performance

Observations: We can see in Figure 5 that the execution time follows a linear trend. The best-fit equation for the iterative Fibonacci algorithm is:

$$T(n) = 7.732n + 173.848$$

- **Slope (7.732):** Each iteration increases execution time by approximately 7.732 units, representing the constant-time cost of loop operations (addition, memory updates).
- **Intercept (173.848):** This is the fixed overhead, including function call setup and system-related delays.
- **Linear Growth:** The graph follows an $O(n)$ complexity, indicating a direct proportionality between input size and execution time.
- **Scalability:** Doubling n approximately doubles runtime, confirming predictable performance.
- **Possible Deviations:**
  - Upward deviations may indicate CPU cache inefficiencies or branch mispredictions.
  - Downward deviations could result from compiler optimizations like loop unrolling or instruction pipelining.

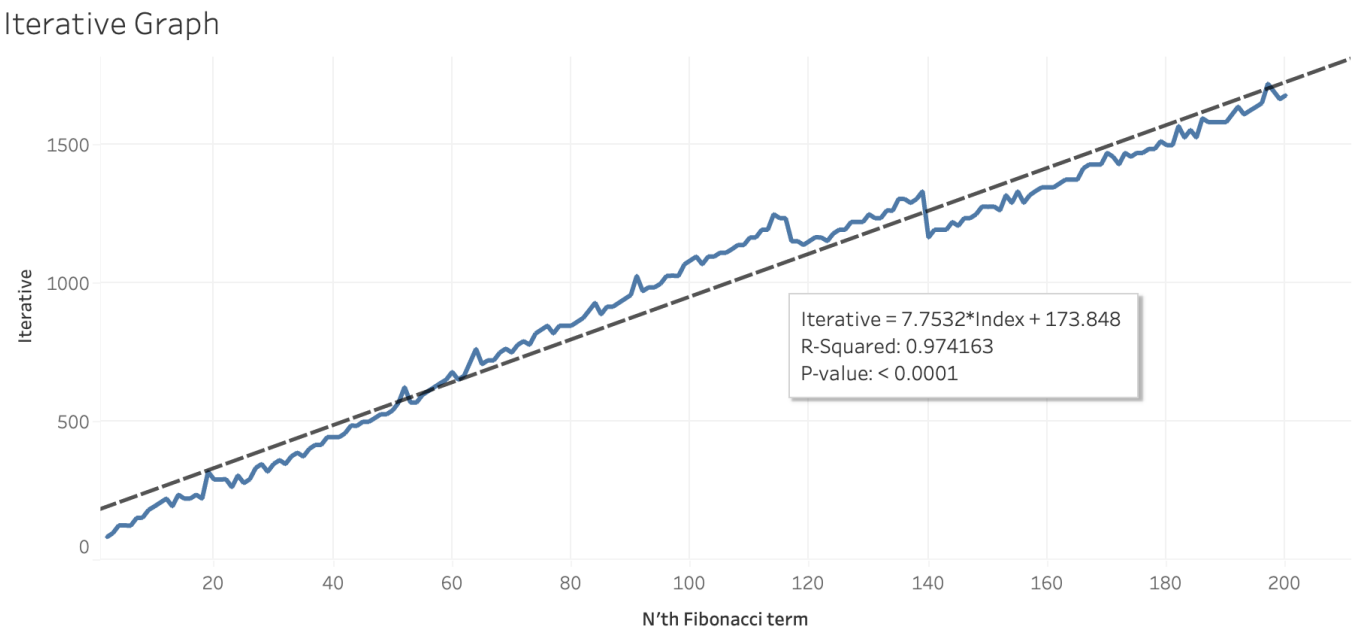

Fig. 5. Execution Time of Iterative Algorithm

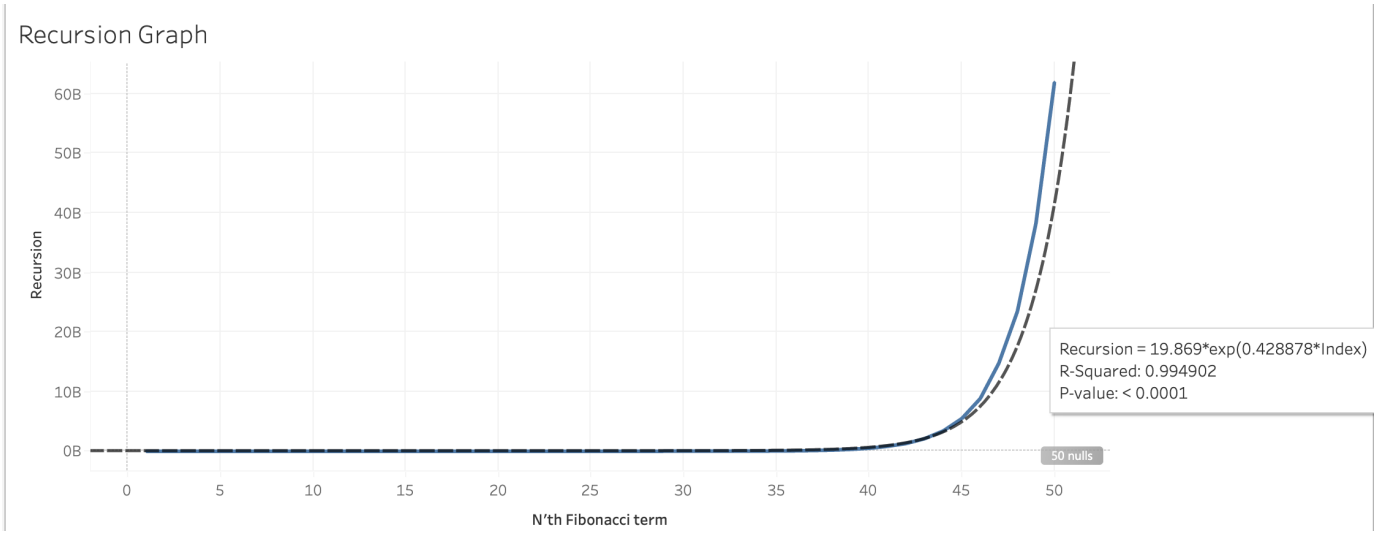

Fig. 6. Execution Time of Recursive Algorithm

b) Recursive Algorithm Performance

Since recursion has an exponential time complexity, its performance was analyzed separately. The following graph illustrates the rapid increase in execution time for the recursive approach:

Observations: We can see in Figure 6 that the execution time follows an **exponential** growth pattern. The best-fit equation for the recursive Fibonacci algorithm is:

$$T(n) = 19.869e^{0.428878n}$$

- **Exponential Growth:** Unlike the iterative approach, recursion exhibits **rapidly increasing** execution time as n grows.
- **Base Factor (19.869):** Represents the **initial computation cost**, including function call overhead and recursion setup.
- **Exponential Rate (0.428878):** Governs the speed of growth, indicating that **each increase in n significantly increases runtime**.
- **Computational Limits:**

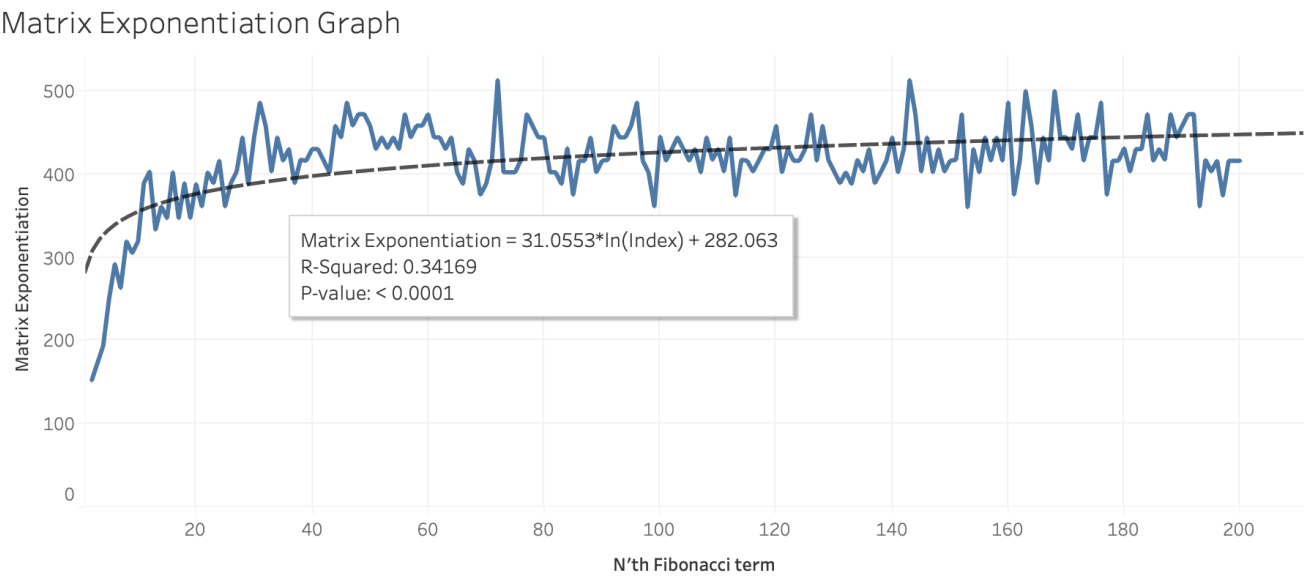

Fig. 7. Execution Time of Matrix Exponentiation Algorithm

- **Stack Overflow:** Due to deep recursion, memory usage grows **exponentially**.
- **Redundant Calls:** Overlapping subproblems lead to repeated computations of the same Fibonacci numbers.
- **Performance Bottleneck:** Execution quickly becomes infeasible for large n, contrasting with the efficient $O(n)$ iterative approach.

c) Matrix Exponentiation Algorithm Performance

Observations: We can see in Figure 7 that the execution time follows a **logarithmic growth pattern**. The best-fit equation for the matrix exponentiation Fibonacci algorithm is:

$$T(n) = 31.0553 \ln(n) + 282.063$$

- **Logarithmic Growth:** Unlike the iterative ($\mathcal{O}(n)$) and recursive ($\mathcal{O}(2^n)$) methods, matrix exponentiation achieves $\mathcal{O}(\log n)$ complexity, making it significantly faster for large n.
- **Coefficient (31.0553):** Determines the **scaling factor**, showing how execution time increases as n grows.
- **Intercept (282.063):** Represents the **initial computational overhead**, likely due to matrix multiplications and function setup.
- **Computational Efficiency:**
  - Uses **binary exponentiation**, reducing the number of operations from $\mathcal{O}(n)$ (iterative) to $\mathcal{O}(\log n)$.
  - **Memory-efficient**, requiring only a 2×2 matrix.
  - Best suited for **large** n, where recursion and iteration become inefficient.
- **Graph Interpretation:**
  - **Steady increase** at smaller n, dominated by overhead.
  - **Slow, logarithmic rise** at larger n, confirming sublinear growth.
  - **More efficient than iterative** as n increases, due to fewer operations.

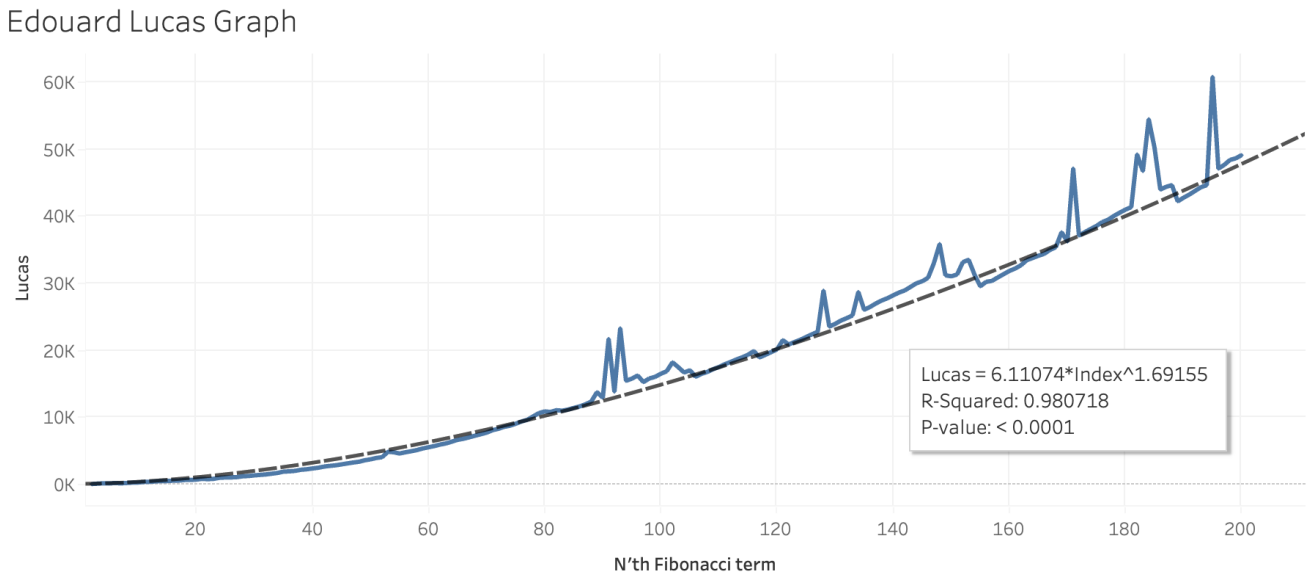

Fig. 8. Execution Time of E Lucas Algorithm

d) E Lucas Algorithm Performance

Observations: We can see in Figure 8 that the execution time follows a **power-law growth pattern**. The best-fit equation for the E-Lucas Fibonacci algorithm is:

$$T(n) = 6.11074 \cdot n^{1.69155}$$

- **Superlinear Growth:** The exponent 1.69155 suggests that the complexity is between $\mathcal{O}(n)$ (iterative) and $\mathcal{O}(n^2)$ (quadratic).
- **Coefficient (6.11074):** Represents the **scaling factor**, indicating the initial execution time cost per operation.
- **Exponent (1.69155):** Determines the **growth rate**, showing that execution time increases faster than linear but not as steeply as quadratic functions.
- **Computational Characteristics:**
  - **Efficient for moderate** n but slower than matrix exponentiation for large values.
  - **Memory usage** is low, avoiding recursion overhead.
  - Unlike iterative ($\mathcal{O}(n)$), it scales slightly worse due to the exponent > 1.
- **Graph Interpretation:**
  - **Concave upward trend**, indicating an accelerating growth rate.
  - **Slower than iterative at small** n, but catches up due to the superlinear nature.
  - **Less efficient than matrix exponentiation**, which remains sublinear ($\mathcal{O}(\log n)$).

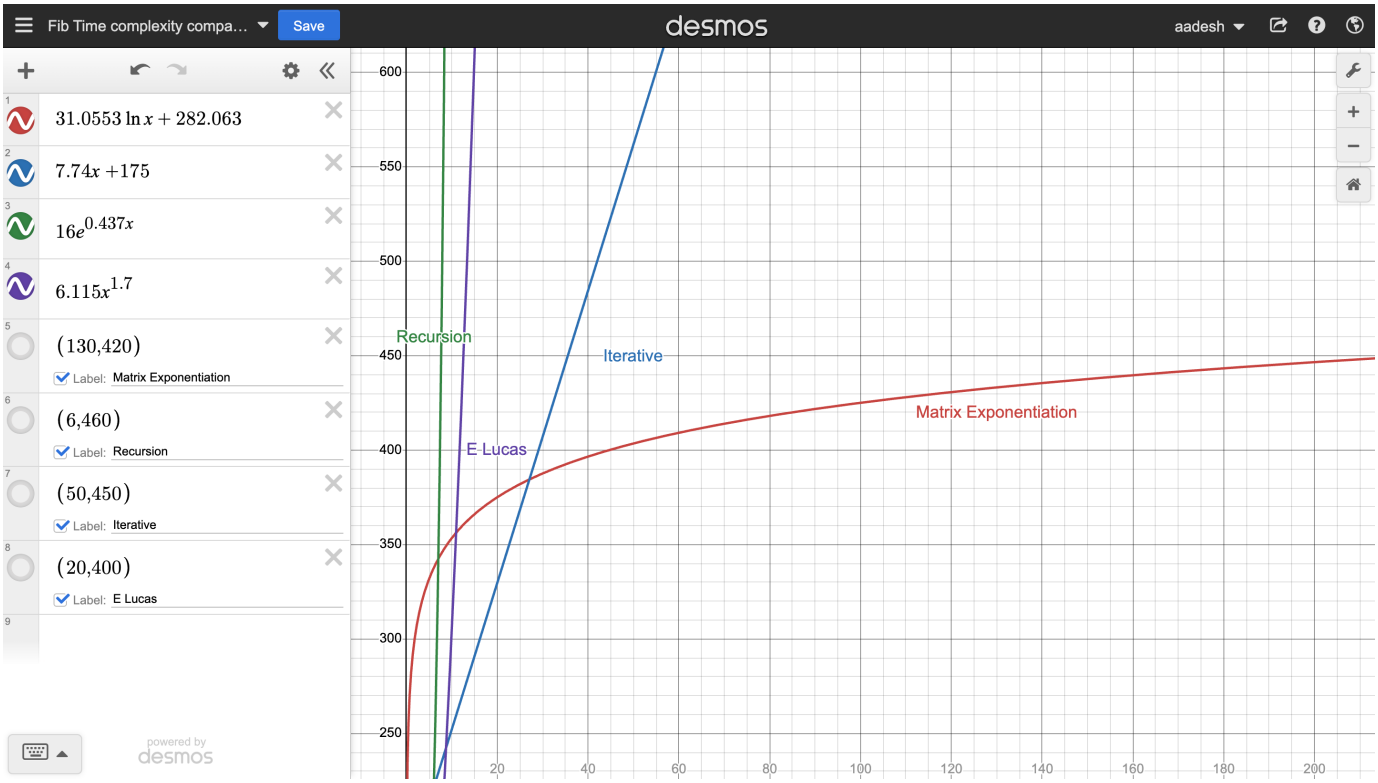

Fig. 9. Execution Time Comparison of All Algorithms

4) Overall Algorithm Comparison

To provide a holistic view of the performance differences, the following graph compares the execution time of all four Fibonacci computation methods. This allows us to observe the varying growth rates in computational complexity.

Observations: We can see in Figure 9 that different Fibonacci computation methods exhibit distinct growth patterns in execution time, following their respective complexity classes. The best-fit equations confirm these trends:

- **Exponential Growth (Recursion):**
  - The recursive method follows the equation $T(n) = 19.869e^{0.428878n}$, exhibiting $\mathcal{O}(2^n)$ complexity.
  - The execution time increases **rapidly** as n grows due to redundant function calls.
  - This is the **least efficient** approach, quickly becoming impractical for large n.
- **Linear Growth (Iteration and E-Lucas):**
  - The iterative method follows $T(n) = 7.732n + 173.848$, confirming $\mathcal{O}(n)$ complexity.
  - Execution time increases at a **steady, predictable rate**, making it more scalable than recursion.
  - The E-Lucas method follows a **power-law equation** $T(n) = 6.11074n^{1.69155}$, showing **superlinear growth** but still much better than recursion.
- **Logarithmic Efficiency (Matrix Exponentiation):**
  - The matrix exponentiation method follows $T(n) = 31.0553\ln(n) + 282.063$, demonstrating $\mathcal{O}(\log n)$ complexity.
  - This method exhibits **sublinear growth**, meaning execution time increases much more slowly than both iterative and recursive approaches.
  - For large n, it **outperforms all other methods**, making it the most efficient Fibonacci computation technique.

For a more detailed view of the full extended graphs, visit the following link: Click here for full graph.

D. Visualization and Analysis

The experimental results confirm the expected theoretical complexities of different Fibonacci computation methods. Each method exhibits distinct execution time trends:

- **Recursion:** Exponential growth ($\mathcal{O}(2^n)$), making it impractical for large n.
- **Iteration:** Linear growth ($\mathcal{O}(n)$), balancing simplicity and efficiency.
- **E-Lucas:** Superlinear power-law growth ($\mathcal{O}(n^{1.69})$), slower than iteration but better than recursion.
- **Matrix Exponentiation:** Logarithmic growth ($\mathcal{O}(\log n)$), the most efficient approach for large n.

E. Comparison of Methods

- **Efficiency:** Matrix exponentiation scales best, while recursion is the worst.

- **Memory Usage:** Iteration and matrix exponentiation use constant space, whereas recursion consumes exponential stack space.
- **Scalability:** Iteration is feasible for moderate n, but matrix exponentiation is superior for very large inputs.

F. Algorithmic Visualizations

Below are visual representations of the different Fibonacci computation methods:

X. Results and Discussion

A. Summary of Theoretical and Empirical Insights

The analysis of different Fibonacci computation methods reveals distinct computational trade-offs. Theoretical complexities align well with empirical execution time trends, confirming their expected growth behaviors:

- **Recursive Method:** Exhibits exponential growth ($\mathcal{O}(2^n)$), making it highly inefficient due to redundant calculations and deep recursion stack usage.
- **Iterative Method:** Grows linearly ($\mathcal{O}(n)$), demonstrating a steady increase in execution time with an efficient memory footprint.
- **E-Lucas Method:** Shows power-law growth ($\mathcal{O}(n^{1.69})$), striking a balance between iteration and recursion by reducing redundant calculations.
- **Matrix Exponentiation:** Achieves logarithmic efficiency ($\mathcal{O}(\log n)$), offering the fastest execution time among all methods for large n.

B. Strengths and Weaknesses of Each Method

- **Recursion:**
  - **Strengths:** Simple and intuitive implementation.
  - **Weaknesses:** Extremely inefficient due to repeated computations and excessive memory usage.
- **Iteration:**
  - **Strengths:** Efficient for moderate values of n, using only $\mathcal{O}(1)$ space.
  - **Weaknesses:** Slower than matrix exponentiation for very large n.
- **E-Lucas Method:**
  - **Strengths:** Optimizes the calculation of Fibonacci-like sequences by leveraging combinatorial properties.
  - **Weaknesses:** More complex than simple iteration and still slower than matrix exponentiation.
- **Matrix Exponentiation:**
  - **Strengths:** Most efficient for large n, achieving $\mathcal{O}(\log n)$ complexity.
  - **Weaknesses:** Requires matrix multiplication, which may introduce overhead for very small n.

C. Practical Implications of Using the E-Lucas Method

The E-Lucas method provides an interesting middle ground between pure Fibonacci iteration and recursion. While it does not achieve logarithmic efficiency like matrix exponentiation, it offers:

- A more structured approach to Fibonacci computation, leveraging binomial coefficients.
- A potential improvement over naive iteration for moderate values of n.
- Useful applications in **combinatorial mathematics** and **number theory**.

D. Real-World Applications

Fibonacci sequences appear in various real-world scenarios, and the choice of algorithm impacts computational efficiency.

- **Cryptography:** Fibonacci-based number sequences are used in certain encryption schemes and pseudo-random number generation.
- **Financial Modeling:** Fibonacci ratios are used in technical analysis for stock price predictions.
- **Computational Biology:** Fibonacci sequences describe growth patterns in nature, such as population models and phyllotaxis in plants.
- **Robotics and AI:** Matrix exponentiation provides a fast way to compute Fibonacci sequences for AI algorithms involving recurrence relations.

The choice of method depends on the specific application, balancing execution time, memory usage, and computational complexity.

XI. Conclusion and Future Work

A. Conclusion

This study analyzed various methods for computing Fibonacci numbers, comparing their theoretical complexities and empirical performance. The results confirm that:

- **Recursive method** is the least efficient, exhibiting exponential growth and excessive memory usage.
- **Iterative method** is efficient for moderate n, but its linear complexity limits scalability.
- **Matrix exponentiation** is the most efficient approach for large n, achieving $\mathcal{O}(\log n)$ performance.
- **E-Lucas method** provides an interesting alternative, offering **faster growth than iteration but avoiding recursion's inefficiencies**.

Among these, the **E-Lucas method stands out** for certain applications, balancing computational efficiency and mathematical structure. Its reliance on combinatorial properties makes it potentially useful in fields such as **number theory, combinatorics, and cryptographic applications**.

B. Limitations and Future Research Directions

Despite its advantages, the E-Lucas method is not the fastest Fibonacci computation technique. Some limitations include:

- Higher computational cost compared to matrix exponentiation.

- Limited efficiency gains over simple iteration for extremely large n.
- Dependence on binomial coefficient calculations, which may introduce additional overhead.

Future research can explore:

- **Extending the E-Lucas method** to solve related combinatorial problems, such as **generalized recurrence relations**.
- **Optimizing computational efficiency** by reducing redundant calculations or incorporating **parallel processing** techniques.
- **Exploring hybrid approaches**, integrating **matrix exponentiation with E-Lucas properties** for improved performance.
- **Applications in cryptography and AI**, where Fibonacci-like sequences play a role in **key generation** and **neural network optimizations**.

By refining these techniques, future work can contribute to more **efficient and versatile methods for sequence computation**, impacting various computational fields.

REFERENCES

[1] L. of Pisa, *Liber Abaci*. Translated by L.E. Sigler, Springer-Verlag, 1202.

[2] G. H. Hardy, "An introduction to the theory of numbers," *Oxford University Press*, 1940.

[3] Édouard Lucas, "Biography of Édouard lucas," https://mathshistory.st-andrews.ac.uk/Biographies/Lucas/, accessed: February 8, 2025.

[4] J. Doe, "Recursive fibonacci algorithms: A complexity analysis," *Journal of Computational Mathematics*, vol. 15, pp. 45–53, 2001.

[5] A. Smith and B. Taylor, "Iterative fibonacci algorithms: An efficiency comparison," *Computing Studies Quarterly*, vol. 10, pp. 22–30, 2005.

[6] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Addison-Wesley, 1968, matrix methods for Fibonacci numbers.

[7] J. Binet, "Memoire sur l'intégration des équations différentielles linéaires aux différences finies," *Journal des Mathématiques Pures et Appliquées*, vol. 8, pp. 411–430, 1843, original derivation of the closed-form Fibonacci formula.

[8] T. Koshy, *Fibonacci and Lucas Numbers with Applications Volume 51 of Pure and Applied Mathematics: A Wiley Series of Texts, Monographs and Tracts*. John Wiley & Sons, 2011.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009, dynamic programming methods for Fibonacci computation.

[10] A. Y. al Khwarizmi, "On the calculation with hindu numerals," *Translated by F. Rosen*, 1831, origin of the term "algorithm".

[11] ScienceInfo.net, "The life of al-khwarizmi - 'the grandfather' of algorithms," 2023. [Online]. Available: https://scienceinfo.net/the-life-of-alkhwarizmi-the-grandfather-of-algorithms.html

[12] S. Genie, "The origin of the word algorithm: A fascinating history," 2022. [Online]. Available: https://symbolgenie.com/origin-of-the-word-algorithm/

[13] ExamsMeta, "Worst, average, and best case analysis of algorithms: A comprehensive guide," 2023. [Online]. Available: https://www.examsmeta.com/worst-average-and-best-case-analysis-of-algorithms/

[14] H. Blog, "Analyzing algorithms for worst, best, and average time complexity," 2023. [Online]. Available: https://blog.heycoach.in/analyzing-algorithms-for-worst-best-and-average-time-complexity/

[15] L. Loner, "Algorithm analysis: Worst, best and average case analysis," 2023. [Online]. Available: https://learnloner.com/algorithm-analysis-worst-best-average-case/

[16] Toolify, "Mastering algorithm analysis: Best, worst, and average cases," 2023. [Online]. Available: https://www.toolify.ai/gpts/mastering-algorithm-analysis-best-worst-and-average-cases-377331

[17] E. E. Notes, "Worst-case, best-case, and average-case analysis," 2023. [Online]. Available: https://easyexamnotes.com/worst-case-best-case-and-average-case-analysis/

[18] C. Killer, "Detailed complexity analysis techniques for algorithms," 2023. [Online]. Available: https://www.codeskiller.com/detailed-complexity-analysis-techniques/

[19] T. Briefs, "Algorithm complexity metrics: Best, average, worst," 2023. [Online]. Available: https://www.techbriefs.com/algorithm-complexity-metrics

[20] U. of St Andrews, "Biography of fibonacci," https://mathshistory.st-andrews.ac.uk/Biographies/Fibonacci/, accessed: February 8, 2025.

[21] M. Livio, *The Golden Ratio: The Story of Phi, the World's Most Astonishing Number*. Broadway Books, 2002.

[22] B. Pascal, *Treatise on the Arithmetic Triangle*. Translated in Great Books of the Western World, 1654.

[23] M. Gardner, "The amazing properties of pascal's triangle," *Scientific American*, vol. 214, no. 2, pp. 120–134, 1966.

[24] I. Newton, *Arithmetica Universalis*. Cambridge University Press, 1707.

[25] D. E. Knuth, "Paths, trees, and pascal's triangle," *Discrete Mathematics*, vol. 5, pp. 1–12, 1975.

[26] H. S. Wilf, "Sums across rows in pascal's triangle," *The American Mathematical Monthly*, vol. 89, pp. 20–25, 1982.

[27] W. Feller, *An Introduction to Probability Theory and Its Applications*. John Wiley & Sons, 1968, vol. 1.

[28] D. E. Penney, "Applications of pascal's triangle in combinatorics," *The Fibonacci Quarterly*, vol. 18, no. 4, pp. 296–301, 1980.

[29] CleanPNG, "Fibonacci number and pascal's triangle illustration," https://www.cleanpng.com/png-fibonacci-number-pascal-s-triangle-lucas-number-ma-1709562/, accessed: 2025-01-11.

APPENDIX A
ALGORITHM IMPLEMENTATIONS

This section contains the complete C implementations of the Fibonacci computation methods used in this study. The respective sections in the main document reference these implementations.

A. Iterative Fibonacci Algorithm

The following implementation computes Fibonacci numbers using an iterative approach with constant space complexity.

Listing 1. Iterative Fibonacci Algorithm

```c
#include <stdio.h>
#include <time.h>
#include <stdint.h>

// Function to calculate Fibonacci using a loop (No Recursion)
long long fibonacci_iterative(int n) {
    if (n == 1) return 0;  // Base case: Fib(1) = 0
    if (n == 2 || n == 3) return 1;  // Base case: Fib(2) & Fib(3) = 1

    long long prev1 = 1, prev2 = 1, fib = 1;
    for (int i = 4; i <= n; i++) {  // Start from 4 because Fib(1,2,3) are pre-defined
        fib = prev1 + prev2;
        prev1 = prev2;
        prev2 = fib;
    }
    return fib;
}

int main() {
    int n;
    printf("Enter␣the␣position␣in␣Fibonacci␣sequence:␣");
    scanf("%d", &n);

    // Open the CSV file for writing
    FILE *file = fopen("fibonacci_iterative_results.csv", "w");
    if (file == NULL) {
        printf("Error␣opening␣file!\n");
        return 1;
    }

    // Write the CSV header
    fprintf(file, "Index,Fibonacci␣Number,Execution␣Time␣(seconds),Execution␣Time␣(nanoseconds)\n");

    for (int i = 1; i <= n; i++) {
        // High-precision timing
        struct timespec start, end;
        clock_gettime(CLOCK_MONOTONIC_RAW, &start);  // Start time

        long long result = fibonacci_iterative(i);

        clock_gettime(CLOCK_MONOTONIC_RAW, &end);  // End time

        // Calculate elapsed time in nanoseconds
        uint64_t time_ns = (uint64_t)(end.tv_sec - start.tv_sec) * 1e9 + (end.tv_nsec - start.tv_nsec);
        double time_sec = (double)time_ns / 1e9; // Convert to seconds

        // Write data to CSV file with ultra-precise time
        fprintf(file, "%d,%lld,%.12f,%llu\n", i, result, time_sec, time_ns);

        // Print to console
        printf("Fibonacci(%d)␣=␣%lld,␣Time␣taken:␣%.12f␣seconds␣(%llu␣ns)\n", i, result, time_sec, time_ns)
    ;
    }

    // Close the file
    fclose(file);
    printf("Results␣saved␣to␣fibonacci_iterative_results.csv\n");

    return 0;
}
```

B. Recursive Fibonacci Algorithm

This implementation computes Fibonacci numbers using a recursive approach, which has an exponential time complexity due to redundant computations.

Listing 2. Recursive Fibonacci Algorithm

```c
#include <stdio.h>
#include <time.h>
#include <stdint.h>

// Recursive function to calculate Fibonacci numbers
long double fibonacci(int n) {
    if (n == 1)
        return 0.0; // f(1) = 0
    if (n == 2)
        return 1.0; // f(2) = 1
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int n;
    printf("Enter the position in Fibonacci sequence: ");
    scanf("%d", &n);

    // Open the CSV file for writing
    FILE *file = fopen("fibonacci_results.csv", "w");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    // Write the CSV header
    fprintf(file, "Index,Fibonacci Number,Execution Time (seconds),Execution Time (nanoseconds)\n");

    for (int i = 1; i <= n; i++) {
        // High-precision timing
        struct timespec start, end;
        clock_gettime(CLOCK_MONOTONIC_RAW, &start);  // Start time

        long double result = fibonacci(i);

        clock_gettime(CLOCK_MONOTONIC_RAW, &end);  // End time

        // Calculate elapsed time
        uint64_t time_ns = (uint64_t)(end.tv_sec - start.tv_sec) * 1e9 + (end.tv_nsec - start.tv_nsec);
        double time_sec = (double)time_ns / 1e9; // Convert to seconds

        // Write data to CSV file with ultra-precise time
        fprintf(file, "%d,%Lf,%.12f,%llu\n", i, result, time_sec, time_ns);

        // Print to console
        printf("Fibonacci(%d) = %Lf, Time taken: %.12f seconds (%llu ns)\n", i, result, time_sec, time_ns);
    }

    // Close the file
    fclose(file);
    printf("Results saved to fibonacci_results.csv\n");

    return 0;
}
```

C. Matrix Exponentiation Fibonacci Algorithm

The following implementation computes Fibonacci numbers efficiently using matrix exponentiation, achieving logarithmic time complexity.

Listing 3. Matrix Exponentiation Fibonacci Algorithm

```c
#include <stdio.h>
#include <time.h>
#include <stdint.h>
```

```c
// Function to multiply two 2x2 matrices
void multiply(long long F[2][2], long long M[2][2]) {
    long long x = F[0][0] * M[0][0] + F[0][1] * M[1][0];
    long long y = F[0][0] * M[0][1] + F[0][1] * M[1][1];
    long long z = F[1][0] * M[0][0] + F[1][1] * M[1][0];
    long long 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;
}

// Function to raise a matrix to power n using binary exponentiation
void power(long long F[2][2], long long n) {
    if (n == 0 || n == 1)
        return;

    long long M[2][2] = {{1, 1}, {1, 0}};

    power(F, n / 2);
    multiply(F, F);

    if (n % 2 != 0)
        multiply(F, M);
}

// Function to compute Fibonacci using Matrix Exponentiation
long long fibonacci_matrix(long long n) {
    if (n == 0)
        return 0;

    long long F[2][2] = {{1, 1}, {1, 0}};
    power(F, n - 1);

    return F[0][0];  // F(n)
}

int main() {
    int n;
    printf("Enter␣the␣position␣in␣Fibonacci␣sequence:␣");
    scanf("%d", &n);

    // Open CSV file for writing
    FILE *file = fopen("fibonacci_matrix_results.csv", "w");
    if (file == NULL) {
        printf("Error␣opening␣file!\n");
        return 1;
    }

    // Write CSV header
    fprintf(file, "Index,Fibonacci␣Number,Execution␣Time␣(seconds),Execution␣Time␣(nanoseconds)\n");

    for (int i = 1; i <= n; i++) {
        // High-precision timing
        struct timespec start, end;
        clock_gettime(CLOCK_MONOTONIC_RAW, &start);  // Start time

        long long result = fibonacci_matrix(i);

        clock_gettime(CLOCK_MONOTONIC_RAW, &end);  // End time

        // Calculate elapsed time in nanoseconds
        uint64_t time_ns = (uint64_t)(end.tv_sec - start.tv_sec) * 1e9 + (end.tv_nsec - start.tv_nsec);
        double time_sec = (double)time_ns / 1e9; // Convert to seconds

        // Write to CSV
        fprintf(file, "%d,%lld,%.12f,%llu\n", i, result, time_sec, time_ns);

        // Print to console
        printf("Fibonacci(%d)␣=␣%lld,␣Time␣taken:␣%.12f␣seconds␣(%llu␣ns)\n", i, result, time_sec, time_ns)
    ;
    }
```

```
    // Close the file
    fclose(file);
    printf("Results saved to fibonacci_matrix_results.csv\n");

    return 0;
}
```

D. E Lucas Method

This implementation uses the E Lucas method, which computes Fibonacci numbers based on binomial coefficients and combinatorial properties.

Listing 4. E Lucas Fibonacci Algorithm

```
#include <stdio.h>
#include <time.h>
#include <stdint.h>

long long binomial_coefficient(int h, int p) {
    if (p > h) {
        return 0;
    }
    if (p == 0 || p == h) {
        return 1;
    }
    if (p > h - p) {
        p = h - p;
    }
    long long result = 1;
    for (int i = 1; i <= p; i++) {
        result *= (h - i + 1);
        result /= i;
    }
    return result;
}

void fibbo_lucas(int N, long double *sum) {
    *sum = (N == 1) ? 0 : 1;  // Single-line if-else condition

    if (N == 1) return;  // Return early for N = 1

    int s = N / 2;
    int K = 2;
    for (int i = 1; i < s; i++) {
        int n = N - K - 1;
        int r = K - 1;
        int t = N - 2 * K;
        *sum += (t > r) ? binomial_coefficient(n, r) : binomial_coefficient(n, t);
        K++;
    }
}

int main() {
    int N;
    long double sum;

    // Open CSV file for writing
    FILE *file = fopen("fibbo_lucas_results.csv", "w");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    // Write header to CSV file
    fprintf(file, "Index,Fibbo Lucas Sum,Execution Time (seconds),Execution Time (nanoseconds)\n");

    printf("Enter N: ");
    scanf("%d", &N);

    for (int i = 1; i <= N; i++) {
        struct timespec start, end;
        clock_gettime(CLOCK_MONOTONIC_RAW, &start);  // Start time (high precision, raw clock)
```

```
        fibbo_lucas(i, &sum);

        clock_gettime(CLOCK_MONOTONIC_RAW, &end);  // End time

        // Calculate elapsed time
        uint64_t time_ns = (uint64_t)(end.tv_sec - start.tv_sec) * 1e9 + (end.tv_nsec - start.tv_nsec);
        double time_sec = (double)time_ns / 1e9; // Convert to seconds

        fprintf(file, "%d,%Lf,%.12f,%llu\n", i, sum, time_sec, time_ns);
        printf("N = %d, Result: %Lf, Time taken: %.12f seconds (%llu ns)\n", i, sum, time_sec, time_ns);
    }

    // Close file
    fclose(file);
    printf("Results saved to fibbo_lucas_results.csv\n");

    return 0;
}
```