EXPLORING METHODS FOR DISCOVERING THE NEXT LARGEST UNKNOWN PRIME NUMBER

This report is submitted in a partial fulfillment of the requirements for the four year Bachelor of Science (Special) Degree in Mathematics with Statistics

W.L.T.Sankalpani

(192126)

Principal Supervisor's Name: Prof. P M N Dharmawardane

Name of the Course Module: MATS 4†28- Research Project

Department of Mathematical Sciences

Faculty of Applied Sciences

Wayamba University of Sri Lanka

Kuliyapitiya

October – 2024

DECLARATION

I declare that except where due acknowledgement has been made, the work is that of myself alone and the work has not been submitted previously, in whole or in part, to qualify for any other academic award.

Signed:

Signature

……
(W.L.T.Sankalpani)

Date:

I certify that this statement is correct.

Signed:

Signature

……
(Prof. P M N Dharmawardane)

Date:

ACKNOWLEDGEMENT

I would like to take this opportunity to express my deepest gratitude to the people who have helped and guided me to make this research project successful. Because without them, it would not be possible to success this research project.

First of all, I have been very grateful to my supervisor, Prof. P M N Dharmawardane, for his continuous guidance, invaluable feedback and unwavering support throughout the course of this research project. His expertise and encouragement helped the successful completion of this study. Without his guidance, this work would not have reached its full potential.

I would like to extend my appreciation to the Department of Mathematical Sciences, Faculty of Applied Sciences at Wayamba University of Sri Lanka for providing me with the resources and good environment to do my research project successfully.

A special thanks to the Prime Pages website for serving as a valuable resource for the collection of prime number data, which forms a crucial part of this research.

I am sincerely grateful to my colleagues and peers for their insightful discussions and moral support. Last but not least, I am thankful to my family and friends for their patience, understanding and constant encouragement throughout this journey. Their support has been my greatest motivation and I dedicate this work to them.

ABSTRACT

The discovery of prime numbers is fundamental aspect of number theory, significantly advancing fields such as in mathematics, cryptography, and computational science. While significant studies have been done in understanding the distribution of primes, the challenge of efficiently determining the next largest prime remains unresolved. This research explores both theoretical and computational approaches to address this problem, focusing on developing new methodologies to discover prime numbers more efficiently. The study begins by reviewing existing primality testing algorithms, including the Sieve of Eratosthenes, Miller-Rabin test, and Sieve of Atkin. These methods are examined in terms of their computational complexity, memory requirements, and scalability. Although the Sieve of Eratosthenes is esteemed for its simplicity and efficiency in generating prime numbers within small ranges, its memory requirements become excessive when applied to larger dataset. Next, the Miller-Rabin test, a probabilistic algorithm, is efficient for individual prime checking but becomes computationally expensive for larger inputs. Then, the Sieve of Atkin, a more complex and modern algorithm, offers improvements in time complexity for larger ranges but requires significant computational overhead for implementation. To improve upon these existing methods, the research focuses on patterns in the modular behavior of prime numbers, particularly the cubes of primes (excluding 2 and 3) under moduli such as 3, 6, 9, and 18. A key observation is that cubes of primes are one less than or one more than a multiple of 9. This modular pattern forms the basis of a new primality testing algorithm. The algorithm systematically checks whether a given number satisfies these modular conditions, significantly reducing the computational complexity of verifying primes. The proposed algorithm was implemented in Python and tested across various input sizes, ranging from small primes to the largest known prime numbers. The study demonstrates that the new method is not only accurate but also highly efficient in terms of execution time and memory usage. It outperforms traditional algorithms, particularly for large prime numbers, offering faster results with lower memory requirements. This makes it particularly well-suited for cryptographic applications where large primes are essential for secure encryption protocols. In addition to the new primality test, the research introduces a novel model for predicting the next largest prime after a given prime. By analyzing the difference between the squares of consecutive primes and Mersenne primes, the study develops a formula that accurately

predicts subsequent primes. This predictive model, when combined with the primality testing algorithm, offers a powerful new tool for prime discovery. Graphical analyses of prime distributions, gaps between consecutive primes, twin primes, and the growth rates of squared and cubed primes were conducted, offering further insights into prime number behavior. The findings of this research contribute significantly to both theoretical and computational number theory. The proposed primality test and predictive model for finding the next largest prime offer advancements in the field of large prime number generation. These methods are highly scalable, memory-efficient, and fast, making them applicable not only to academic research but also to practical fields such as cryptography and secure communication.

CONTENT

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

CHAPTER 1: INTRODUCTION

1.1 Background

For ages, researchers have been fascinated with prime numbers because of their distinct characteristics and stochastic distribution, which make them the fundamental units of arithmetic. Primes are defined as integers larger than one that have no divisors other than one and themselves. They are essential to many areas, including computer science, pure mathematics, and cryptography. Primes are a key concept in number theory. Despite their basic definition, prime numbers display complex behaviors, particularly as they grow larger, making their discovery both a fascinating theoretical challenge and a significant practical problem in computer disciplines. The quest to find large prime numbers continues to drive innovation, particularly in the context of cryptography, where large primes are important for the security of modern encryption systems.

Even with the availability of existing techniques, finding new, larger prime numbers is still a challenging task. The accuracy, memory consumption, and computing efficiency of existing methods must all be balanced, especially when quantities approach the millions or billions. With the aid of extremely sophisticated algorithms and vast computing power, the largest known prime numbers which are generally found in the form of Mersenne primes, or primes that may be expressed as $2^n - 1$, where n is a prime have been found. Although Mersenne primes are important in number theory and cryptography, the general challenge of determining the next greatest prime is still unresolved.

1.2 Significance of Study

The significance of prime numbers extends beyond theoretical mathematics. In cryptography, large primes are essential for creating secure encryption keys, which protect sensitive information in fields such as online banking, communications, and data storage. The security of encryption algorithms like Riveset-Shamir-Adleman (RSA) depends on the use of large prime numbers, as the difficulty of factoring large composite numbers into primes underpins the security of public-key cryptosystems. Discovering larger primes enhances cryptographic strength, making systems more secure against possible attacks.

Prime numbers are integral to understanding the structure of natural numbers, and their distribution informs various fields of mathematics, including algebra, combinatorics, and geometry. As computational tools continue to improve, finding new, efficient algorithms for prime number discovery becomes more pressing. Improved methods not only deepen our theoretical understanding of primes but also have practical implications for advancing technology in secure communications and computing.

This research contributes to both theoretical and applied mathematics by developing an efficient algorithm to find the next largest prime number. In doing so, it addresses a crucial gap in prime number discovery and provides a scalable solution that can be applied to large numbers. The proposed method has the potential to advance cryptographic security and contribute to ongoing research in number theory.

1.3 Objectives of the Study

The primary objective of this study is to develop a novel, efficient methods for primality testing discovering the next largest prime number, particularly for large numbers.

1.4 Overview of the Study

- Chapter 1 - This chapter gives a brief introduction of the background of the selected research area, the significance of the research study and the objective of the study
- Chapter 2 - Refers to the literature review for this research study with related to current methods for primality testing and prime number generating, findings of prime number relationships and theories related to area of study.
- Chapter 3 - Discuss the computational and theoretical techniques used to find the new algorithm and new models for primality testing and prime number generating
- Chapter 4 - Includes the results obtained from the analysis. Similarly, it includes interpretation of results and identification of new algorithm and new models for primality testing and prime number generating.
- Chapter 5 - This chapter discusses the conclusion of the analysis conducted in previous chapters and make suggestions for future works.

CHAPTER 2: LITERATURE REVIEW AND THEORETICAL BACKGROUND

Prime numbers have held a special place in mathematics for centuries, both for their theoretical significance and their practical applications. Over time, mathematicians have developed various methods and algorithms to generate and verify primes. The growing demand for large primes, particularly in cryptography, has spurred the development of more sophisticated algorithms capable of efficiently handling the computational burden associated with testing extremely large numbers. This section reviews the key algorithms and concepts from number theory that have influenced the finding of primes, verifying primes and provides the theoretical background that underpins this research.

2.1 Literature Related to Area of Study

One of the earliest and most fundamental methods for identifying prime numbers is the Sieve of Eratosthenes, a simple and efficient algorithm introduced by the ancient Greek mathematician Eratosthenes around 240 B.C. The Sieve of Eratosthenes is an algorithm used to find all prime numbers up to a specified integer value. It is one of the most efficient ways to find small prime numbers (Burton, 2011). For a given upper limit n the algorithm works by iteratively marking the multiples of primes as composite, starting from 2. Once all multiples of 2 have been marked composite, the multiples of next prime, 3 are marked as composite. This process continues until $p < \sqrt{n}$, where p is a prime number (Geeksforgeeks.org, 2012). While the Sieve of Eratosthenes is highly effective for small numbers, its efficiency diminishes for larger primes due to the extensive memory requirements needed to store the list of integers being marked (Lazer, 2024).

The Miller-Rabin primality test is a probabilistic algorithm used to determine whether a given number is prime or composite. The algorithm works by repeatedly testing a number with randomly chosen bases to check whether it behaves like a prime it provides a high level of confidence in its result by repeating iterations (Ginni, 2022). It is one of the simplest and fastest tests known. Miller-Rabin test requires less memory than deterministic methods. Gary L. Miller discovered the test in 1976. Miller's version of the test is deterministic, but its correctness relies on the unproven extended Riemann hypothesis. Michael O. Rabin modified

it to obtain an unconditional probabilistic algorithm in 1980. The algorithm can sometimes falsely identify composite numbers as primes, known as "false positives" (Kleinberg, 2010).

The Sieve of Atkin represents a contemporary algorithm developed for identifying all prime numbers up to a specified integer. Introduced in 2003 by mathematicians A.O.L. Atkin and D.J. Bernstein, this algorithm leverages the properties of binary quadratic forms to efficiently calculate prime numbers (Wrentz, 2021). The Sieve of Atkin is based on quadratic forms and uses modular arithmetic to identify prime numbers more efficiently than its ancient predecessor, particularly for large datasets. The algorithm works by marking potential primes based on their behavior under modulo 60 and selectively flipping flags for numbers that satisfy certain quadratic equations $4x^2 + y^2 = n, 3x^2 + y^2 = n$ and $3x^2 - y^2 = n$ when $x > y$ While the Sieve of Atkin is faster than the Sieve of Eratosthenes for large ranges of numbers, it is also more complex to implement. Its advantages are most apparent when generating large quantities of primes, but it still requires significant computational resources for extremely large numbers (Geeksforgeeks.org, 2016).

Porras (2022) worked on the foundational aspects of prime number theory, specifically exploring the structure of primes greater than or equal to 5. He demonstrated that all primes greater than or equal to 5 can be expressed in the form of either $6k + 1$ or $6k - 1$, where k is an integer. Porras's findings provide a solid theoretical framework for understanding prime number distribution and are instrumental in the development of various methods for prime identification. His work is particularly relevant for algorithms that rely on modular arithmetic to identify prime numbers, as it sets the stage for more refined techniques like those based on the behavior of squared primes, thereby facilitating a more efficient approach to prime verification.

Parker (2024) built upon Porras's foundational work by focusing on an intriguing pattern related to the squaring of primes. observed that when primes (excluding 2 and 3) are squared, they consistently produce results that are either one more than or one less than a multiple of 24, depending on whether they are in the form of $6k + 1$ or $6k - 1$. Specifically, primes in the form of $6k + 1$ yield squares that are one more than a multiple of 24, while primes in the

form of $6k - 1$ produce squares that are one less than a multiple of 24. This pattern, first suggested by Porras's work, provides a powerful tool for identifying prime numbers using modular arithmetic.

Modular arithmetic has also been instrumental in exploring the properties of Mersenne primes, which are primes of the form $2^n - 1$, where n itself is a prime number. Mersenne primes have long been of interest to mathematicians, not only because of their elegant form but also because they are among the largest known prime numbers. The discovery of Mersenne primes is closely linked to the development of algorithms that can handle large exponents efficiently (Weisstein, 2024).

2.2 Theories Related to Area of Study

The Big O notation is a powerful tool used to express the time and space complexity of algorithms. It allows us to compare and contrast different algorithms, predicting how they will scale with larger inputs and identifying potential bottlenecks in their execution. Time complexity in Big O notation express the runtime of an algorithm in terms of how quickly it grows relative to the input '*n*' by defining the *N* number of operations that are done on it. It provides an estimate of the worst-case time required to execute an algorithm as a function of the input size. In other words, it gives us an upper bound on the time taken by the algorithm to complete its task. As example, under Constant Time complexity (O(1)) the algorithm's running time does not depend on the size of the input. It performs a fixed number of operations. Under logarithmic time complexity (O(log *n*)) , the algorithm's running time grows logarithmically with the size of the input. Under linear time complexity (O(*n*)), the algorithm's running time scales linearly with the size of the input. Under linearithmic time complexity(O(*n* log *n*)), The algorithm's running time grows in proportion to *n* times the logarithm of *n*. Space complexity in Big O notation measures the additional space used by the algorithm, not the total space. It represents the worst-case memory consumption as the input size increases. As example, under constant space complexity (O(1)) , the algorithm uses a fixed amount of memory that does not depend on the input size. Under linear space complexity (O(*n*)) , The algorithm's memory usage grows linearly with the input size. Under quadratic

space complexity ($O(n^2)$) the algorithm's memory usage increases proportionally to the square of the input size (Okeke, 2023).

CHAPTER 3: METHODOLOGY

This research focuses on developing a novel approach to prime number identification by utilizing both theoretical and computational methods. The methodology is divided into several stages, each aimed at examination of current primality testing and prime generating algorithms, exploring prime number properties, developing efficient primality testing algorithms, and generating subsequent primes through prime square analysis. The core elements of this approach include the evaluation of existing algorithms, graphical exploration of prime number patterns, the development of a new primality testing algorithm based on modular arithmetic, and the creation of a predictive model using prime square differences. Data were collected from the Prime Pages web site as first thousand prime numbers. All computational algorithms and graphical analysis were implemented using Python 3.10, executed on the Google Colab online platform.

3.1 Examination of Current Algorithms

The initial phase of the research involves analyzing well-established primality testing and prime generating algorithms. This phase aims to benchmark their performance, which serves as a foundation for comparison with the new methods developed in this study. Three primary algorithms were chosen for evaluation:

- Sieve of Eratosthenes
- Miller-Rabin Primality Test
- Sieve of Atkin

Each algorithm was assessed based on its time complexity, space complexity, and performance particularities. Python implementations of each algorithm were used to test various input sizes, ranging from small to large prime number.

3.1.1 Implementation and Benchmarking

- **Sieve of Eratosthenes**: Time complexity and space complexity of this algorithm were analyzed. Time complexity was O(n log log n) and space complexity was O(n) in theoretical calculations where n was input size. In order to verify this theoretical

calculations, this algorithm was implemented using an array to mark non-prime numbers under the input size ranges 1000, 10000,100000,500000 and 1000000. All prime numbers within this ranges were generated and execution time in seconds(s) and memory usage in bytes were measured.

- **Miller-Rabin Primality Test**: Time complexity and space complexity of this algorithm were observed. The time complexity was $O(k\ (\log\ n)^3)$ and the space complexity was $O(\log\ n)$ in theoretical calculations where k is number of iteration and n is input size. In order to verify this theoretical calculations, within input ranges 1000, 10000,100000,500000 and 1000000, all numbers were checked whether they were prime or not and the probabilistic nature of this algorithm was tested with 10 iterations. Then execution time in seconds(s) and memory usage in bytes were measured.
- **Sieve of Atkin**: Time complexity and space complexity of this algorithm were benchmarked. Time complexity was $O(n\ /\ \log\log\ n)$ and space complexity was $O(n)$ in theoretical calculations where n was input size. Implemented using modular arithmetic and quadratic equations, the Sieve of Atkin was tested for different input ranges 1000, 10000,100000,500000 and 1000000. All prime numbers were found within these ranges and execution time in seconds(s) and memory usage in bytes were measured.

This evaluation of existing methods highlights the performance trade-offs in prime generating and primality testing algorithms, which guided the design of the new primality testing algorithms.

3.2 Exploring Prime Number Relationships Using Graphical Plotting

Graphical analysis was used to visualize relationships and patterns in prime numbers, providing insights into prime behavior and distribution. This phase focuses on examining prime gaps, twin primes, and the behavior of squared and cubed primes.

3.2.1 Prime Distribution Analysis

First thousand primes were plotted against their indices to study the prime distribution at their lower indices and higher indices. The data helped reveal patterns in prime number distribution.

3.2.2 Prime Gap Analysis

Prime gaps are the difference between consecutive primes. So prime gaps were plotted to study their growth over large numerical ranges. It helped to get realization of prime gaps and it helped to generate new primality testing algorithms.

3.2.3 Twin Prime Analysis

Twin prime is the pair of prime numbers with a difference of 2. By plotting twin primes, it revealed how twin primes distributed at lower indices and higher indices and it had been used to get some idea to develop new algorithms.

3.2.4 Graphical Analysis of Prime Squares and Cubes

Graphical plotting was also used to analyze the behavior of squared and cubed primes. This was really important for understanding that there were some pattern among squared and there were some pattern among cubed primes. And it helped to develop new primality test algorithms.

3.3 Cubes of Primes Analysis

The third phase focuses on identifying and leveraging modular arithmetic patterns in prime numbers to develop a more efficient primality testing algorithm.

3.3.1 Identification of Modular Patterns in Prime Numbers

Under this phase, cubes of prime numbers were calculated and those cubes of primes were divided by different divisors. Then it was checked the residues and it was examined whether there were some patterns under different modular.

All primes greater than or equal to 5 can be expressed in the form of either $6k + 1$ or $6k - 1$, where k is an integer. With this form of prime some similar patterns under specific modular 3, 6, 9 and 18 were identified.

3.3.2 Modular Arithmetic Analysis with Algebraic Manipulation

Under this phase, those four patterns were proved by conducting modular arithmetic analysis with algebraic manipulation.

Since primes can only be above or below a multiple of 6, prime number *p* was considered, which could be represented as one of the following forms:

- $p = 6k + 1$,
- $p = 6k - 1$,

where *k* was an positive integer that could be either a positive even $(2m)$ integer or a positive odd$(2m + 1)$ integer. Therefore, the primes greater than or equal to 11 could be represented as one of the following forms:

- $p = 6(2m) + 1 = 12m + 1$
- $p = 6(2m + 1) + 1 = 12m + 7$,
- $p = 6(2m) - 1 = 12m - 1$,
- $p = 6(2m + 1) - 1 = 12m + 5$.

Finally, the cube of prime *p* were examined.

Using those results and using those specific arithmatic modulo 3, 6, 9 and 18, a prime *p* could be written in four forms and also four methods for primality testing, were developed.

3.3.3 Comparison the time complexity, space complexity and scalability

Under this phase, accuracy, time complexity and space complexity of each methods were measured and compared in order to find the best method to check whether the given number is prime or not.

3.3.3.1 Time Complexity and Space Complexity

Each methods' execution duration in seconds(s) and memory usage in bytes within different input ranges (0, 1000), (0.10000), (0, 100000), (0, 500000) and (0, 1000000) were measured using the time module and the tracemalloc module in Python. Then results were compared and the method which has minimum execution time and lowest memory usage were selected as the best method for primality testing.

3.3.3.2 Scalability

In order to measure scalability, it was needed to test execution time and memory usage of each method with a very large number. The largest known prime number is $2^{82589933} - 1$. Therefore execution time in seconds(s) and memory usage in megabytes(MB) were computed, by checking whether the $2^{82589933} - 1$ is prime or not, using the 'time' and 'tracemalloc' modules for measuring execution time and memory usage, respectively. Then results were compared and the method which has minimum execution time and lowest memory usage were selected as the best method for primality testing.

3.3.4 Comparison Time Complexity and Space complexity between the New Algorithm and Current Algorithms

Execution duration and memory usage of the new primality testing algorithm and current primality testing and prime generating algorithms such as Sieve of Eratosthenes, Miller-Rabin and Sieve of Atkin within different input ranges (0,1000), (0.10000),(0,100000), (0,500000) and (0,1000000) were measured using time module and tracemalloc module in Python. Execution duration and memory usage were measured in seconds and in bytes respectively. Then results were compared and it was checked whether the new primality testing algorithm was efficient and scalable or not.

3.4 Squares of Primes Analysis

Under the fourth phase, squares of primes were calculated and then the differences of those consecutive squared primes were analyzed. Therefore it could be developed a new model using this analysis. In addition to the primality testing algorithm, this study developed a predictive model for estimating the next largest prime number based on differences between the squares of consecutive primes.

3.4.1 First Thousand Primes Analysis

Under this analysis, the differences of squares of first thousand consecutive primes were examined. Therefore the model was developed by analyzing the differences between the squares of consecutive primes:

$$P_n^2 - P_{n-1}^2$$

This difference grows predictably as prime numbers increase, providing a mathematical basis for estimating the next prime number. The patterns observed were used to create a model of generating the next prime after a given prime.

3.4.2 Last Ten Known Mersenne Primes Analysis

Under this, the largest ten known consecutive mersenne primes were examined.

Table 3.1. Largest ten known consecutive mersenne primes

$2^{24036583}-1$
$2^{25964951}-1$
$2^{30402457}-1$
$2^{32582657}-1$
$2^{37156667}-1$
$2^{42643801}-1$
$2^{43112609}-1$
$2^{57885161}-1$
$2^{74207281}-1$
$2^{77232917}-1$
$2^{82589933}-1$

The differences of squares of these largest ten known consecutive mersenne primes were analyzed. Therefore a new model was developed by analyzing these differences between the squares of largest ten known consecutive mersenne primes. That model helped guide further the next largest unknown prime discovery efforts.

CHAPTER 4: RESULTS AND DISCUSSION

The results of this study were derived from the examination of current primality testing and prime generating algorithms, the graphical exploration of prime number behavior, the development and the computational implementation of the new primality testing algorithm, and the predictive model based on prime square differences. This section presents the findings from each of these components, followed by a discussion of how these results compare to existing methods and their implications for prime number theory and cryptographic applications.

4.1 Performances of Current Algorithms

In this section, performances of each current primality testing and prime generating algorithms were described clearly, providing good and bad about those algorithms, their particularities, areas where improvements can be made and etc.

4.1.1 Strengths and Weaknesses of Current Algorithms

Strengths and weaknesses of each current primality testing and prime generating algorithms were mentioned based on theoretical details of memory complexity and time complexity of each current algorithms.

Table 4.1. Memory complexity, time complexity, strengths and weaknesses of current algorithms

Algorithm	Memory Complexity	Time Complexity	Strengths	Weaknesses
Sieve of Eratosthenes	$O(n)$	$O(n \log\log n)$	• Simple and efficient	• High memory usage
Miller-Rabin	$O(k \log n)$	$O(k (\log n)^3)$	• Fast for individual primes • Low memory usage	• Possible false positives (probabilistic)

Sieve of Atkin	$O(n)$	$O(n / \log \log n)$	• Efficient for large ranges of primes	• Complex implementation • High memory usage

Sieve of Eratosthenes Algorithm was very efficient for finding all primes up to a large number but requires significant memory. Miller-Rabin Algorithm was good for checking individual numbers for primality with high confidence and probabilistic nature can lead to false positives. Atkin Sieve Algorithm was faster and more efficient for large ranges but complex to implement compared to the Sieve of Eratosthenes.

4.1.2 Verification of Theoretical Calculations

In this section, execution time in seconds(s) and memory usage in bytes were mentioned under different input ranges 1000, 10000,100000,500000 and 1000000 and results were described. Since this method depended on the hardware, the implementation, and the test cases, this could be used as complement for theoretical calculation, not as a substitute.

Table 4.2. Execution time(s), memory usage (bytes) and observations of current algorithms

Algorithm	**Input size**	**Execution Time(s)**	**Memory Usage (bytes)**	**Observations**
Sieve of Eratosthenes	1000	0.004815	8348	• Fast, efficient for large ranges. • High memory usage for large input ranges
	10000	0.004871	80348	
	100000	0.017757	800376	
	500000	0.090962	40000376	
	1000000	0.251724	80000376	
	1000	0.000176	308	• Fast for small numbers
	10000	8.964538	336	

Miller-Rabin	100000	8.535385	364	• Slow for medium to large range • Low memory Usage
	500000	8.201599	13796	
	1000000	8.177757	37330	
Atkin Sieve	1000	0.002064	8324	• Efficient , fast for large ranges • High memory usage for large input size
	10000	0.020155	80324	
	100000	0.220874	800436	
	500000	1.085028	4000436	
	1000000	2.152453	8000436	

(Maximum speed of the computer processor is 2.16 GHz and installed memory(RAM) is 2GB)

Sieve of Eratosthenes Algorithm demonstrated the fastest execution times across all input sizes, making it a suitable choice for applications requiring quick results for generating prime numbers. Miller-Rabin Primality Test performed well for smaller input sizes, but became slow as the input size grew. Sieve of Atkin Algorithm was theoretically more efficient than the Sieve of Eratosthenes Algorithm, showed similar execution times in practice.

Both Algorithms Sieve of Eratosthenes and Sieve of Atkin demonstrated High memory usage. Miller-Rabin Primality Test showed moderate memory usage, which increased linearly with the input size. This is expected due to its iterative nature.

Based on these analysis, the Sieve of Eratosthenes Algorithm is recommended for applications requiring efficient prime number generation. The Miller-Rabin Primality Test is suitable for scenarios where probabilistic accuracy is acceptable and moderate input sizes are involved. The Sieve of Atkin Algorithm offers a viable alternative to the Sieve of Eratosthenes Algorithm, particularly for specific use cases where its overhead can be justified.

4.2 Graphical Exploration of Prime Number Relationships

In this section, graphical analysis was used to visualize key properties of prime numbers, including prime gaps, twin primes, and the behavior of squared and cubed primes by plotting first 1000 prime numbers.

4.2.1 Prime Number Distribution

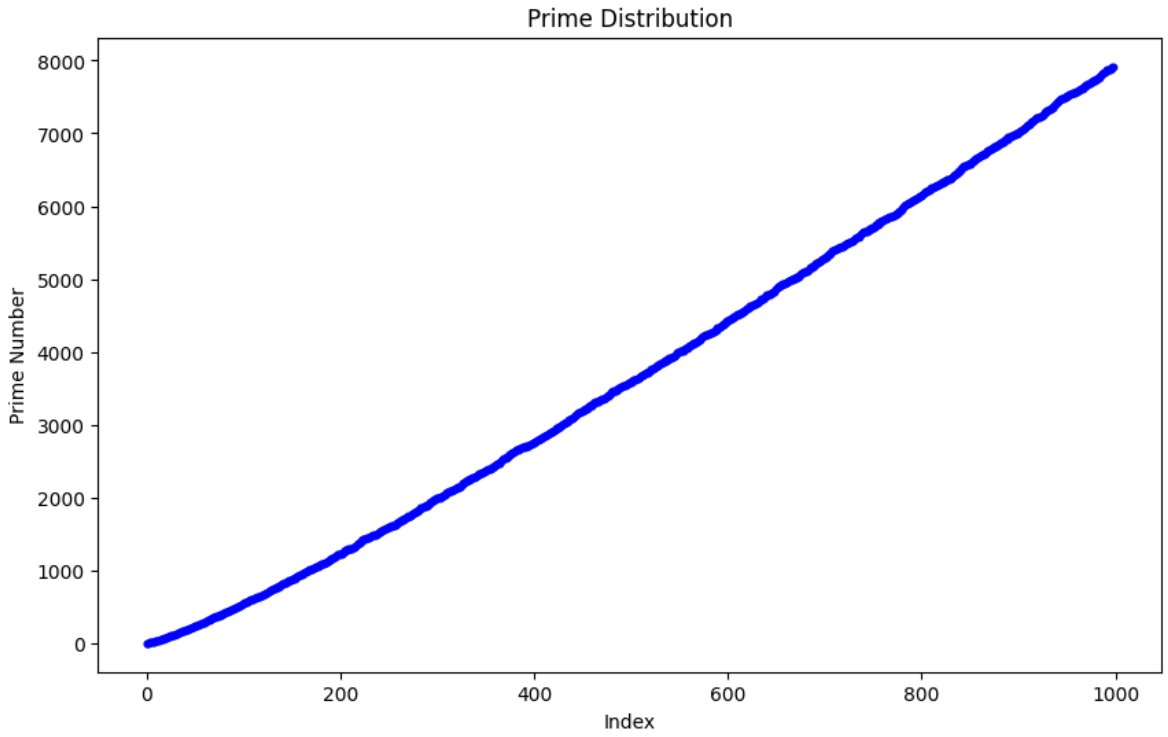

Figure 4.1. Prime number distribution

This plot showed approximately positive straight line.

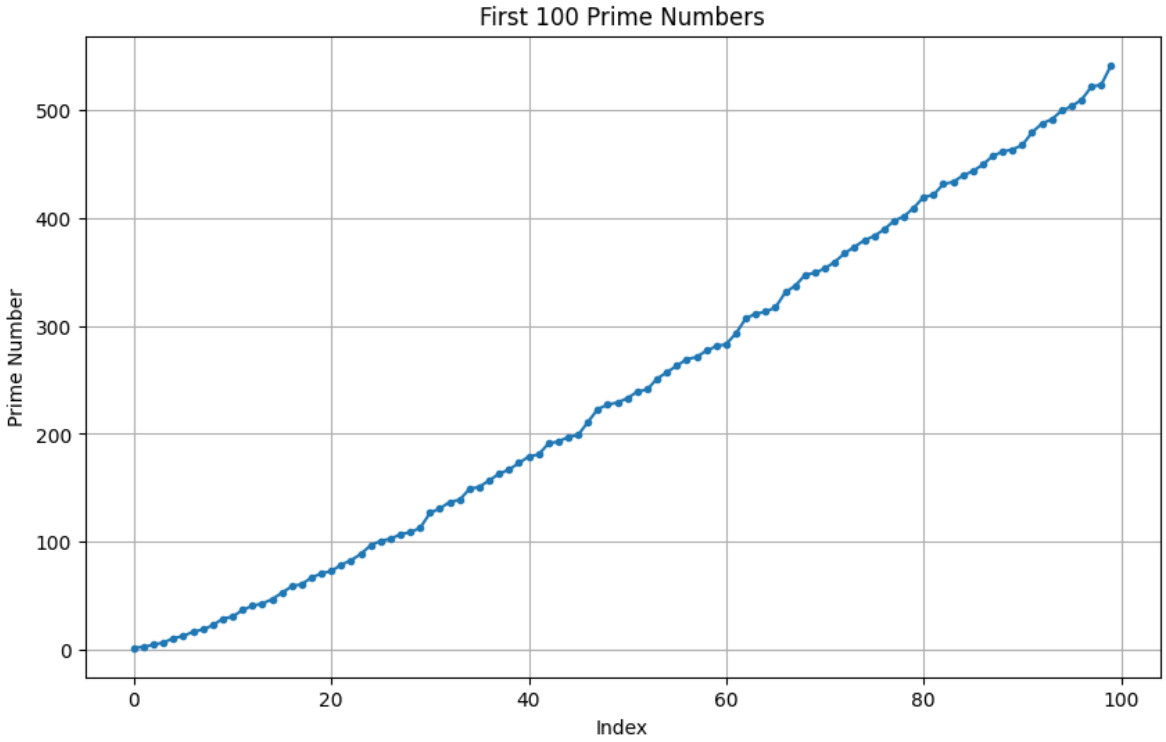

Figure 4.2. First Hundred Primes distribution

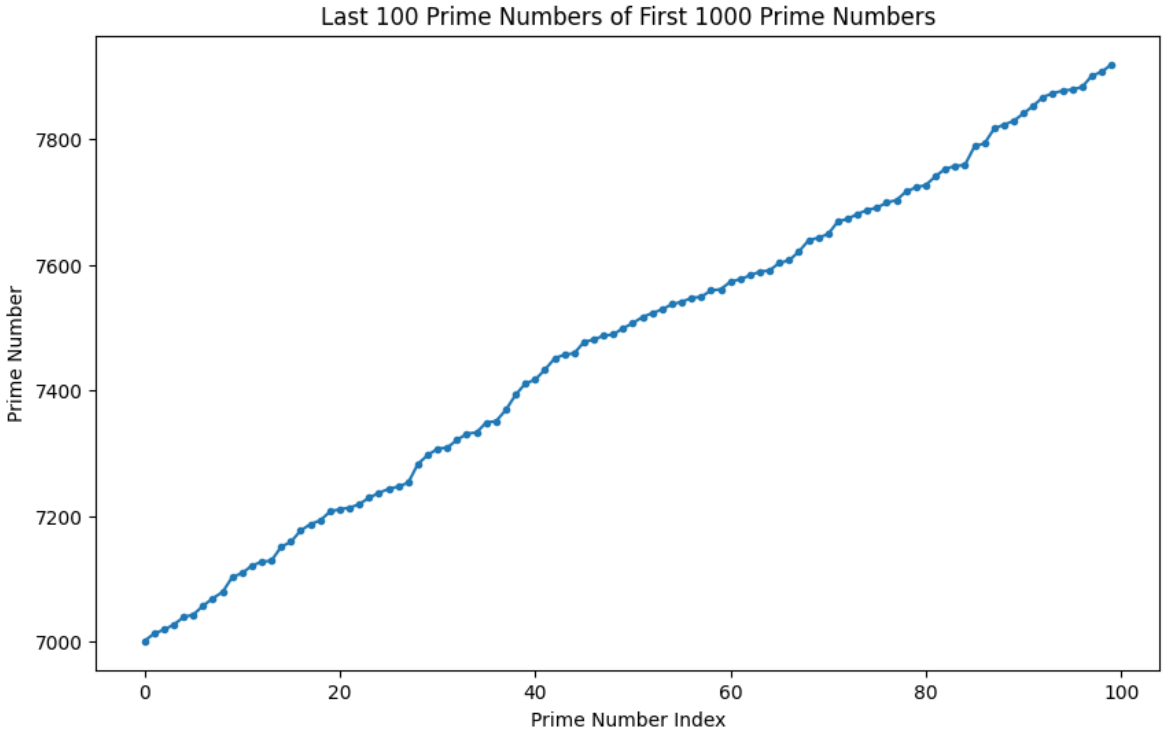

Figure 4.3. Last Hundred Primes distribution

First 100 prime numbers and last 100 prime numbers were considered. First 100 primes were in between 0 and 600. Last 100 prime numbers were in between 7000 and 8000.

Therefore this plot indicates that prime numbers are more frequent among smaller numbers and become less frequent as numbers get larger.

4.2.2 Prime Gap

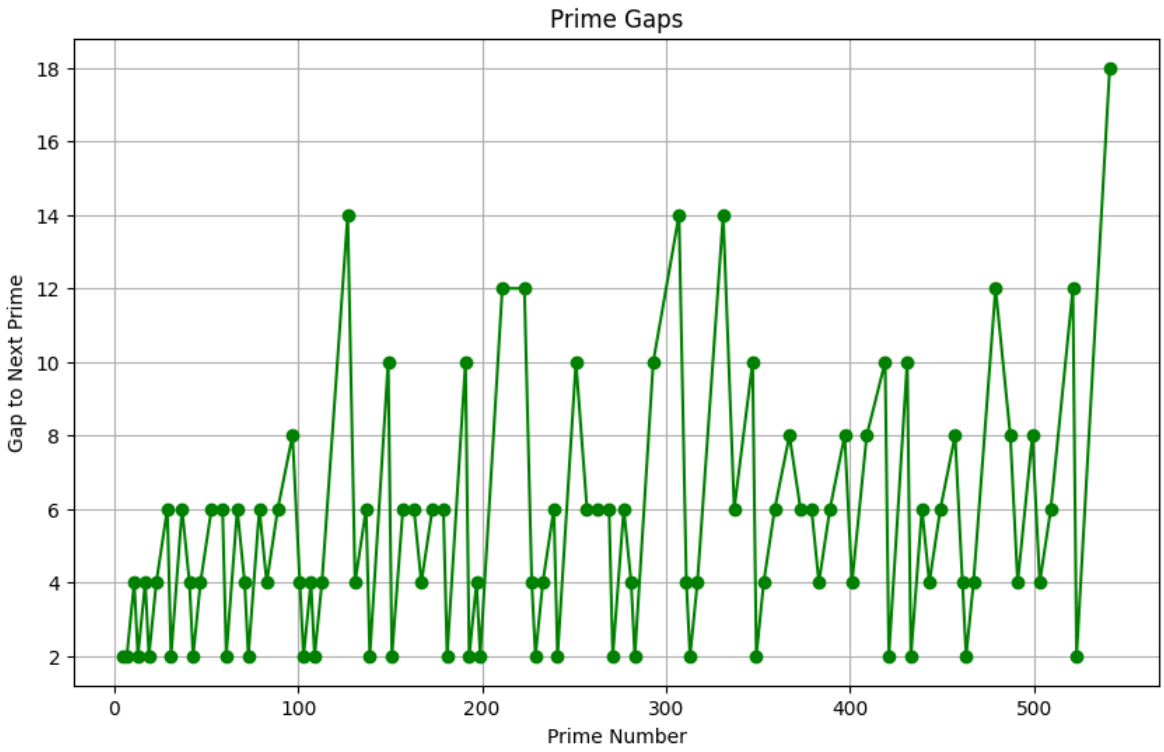

Figure 4.4. Gap to next prime number

The plot showed increasing trends. Therefore the gaps between consecutive primes increased as the primes become larger.

4.2.3 Twin Prime

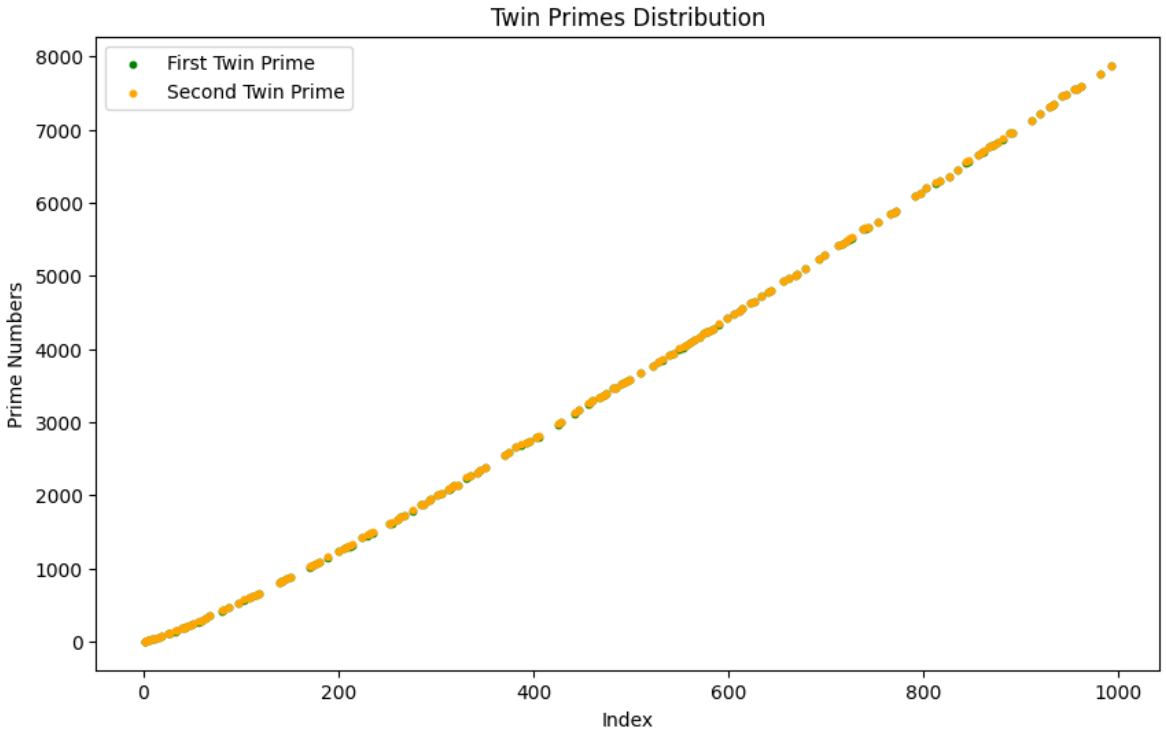

Figure 4.5. Twin prime distribution

This plot showed clusters of points at lower indices with some isolated points at higher indices. Therefore more twin primes are among smaller primes and there are less twin primes as the numbers increased.

4.2.4 Squares of Prime

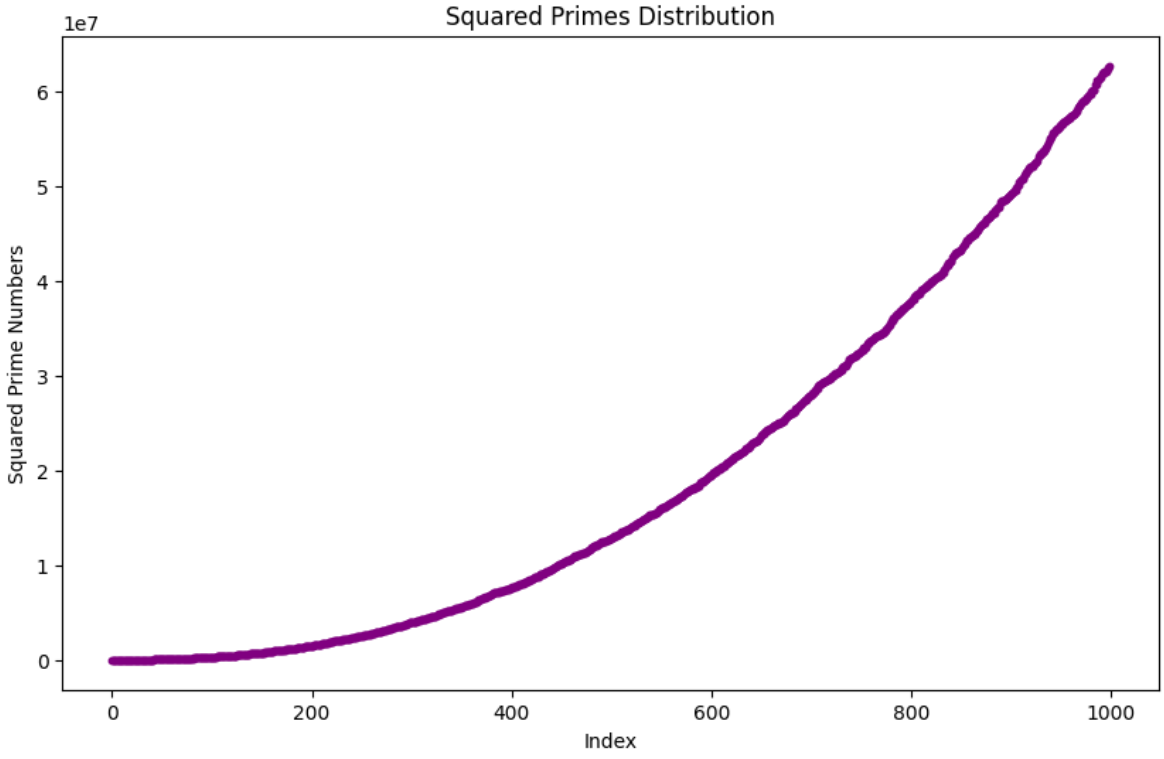

Figure 4.6. Squared primes distribution

The plot indicated a curve where the values rapidly increased. Therefore the square of prime numbers grow quickly for moderately large primes.

4.2.5 Cubes of Primes

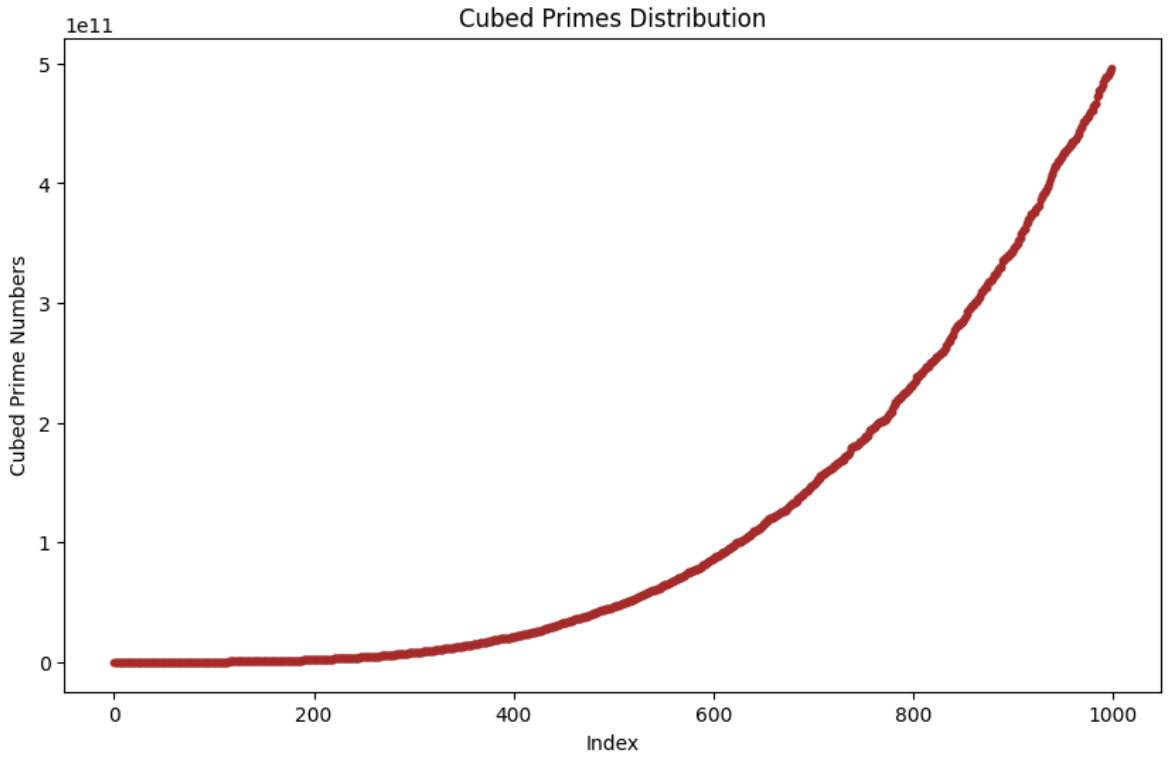

Figure 4.7. Cubes of primes distribution

The plot indicated a curve where the values slowly increased. Therefore the cubes of prime numbers grow slowly for moderately large primes and grow quickly increased for very large primes.

4.3 Cubes of Primes Analysis

4.3.1 Congruence Relations

The cubes of primes under different modules (2 to 18) are as follows:

Table 4.3. Residues of cubes of primes under different divisors (2 to 9)

Primes	Cube	Modulo Operation	Residues							
			n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9
2	8	8 mod n	0	2	0	3	2	1	0	8
3	27	27 mod n	1	0	3	2	3	6	3	0
5	125	125 mod n	1	2	1	0	5	6	5	8
7	343	343 mod n	1	1	3	3	1	0	7	1
11	1331	1331 mod n	1	2	3	1	5	1	3	8
13	2197	2197 mod n	1	1	1	2	1	6	5	1
17	4913	4913 mod n	1	2	1	3	5	6	1	8
19	6859	6859 mod n	1	1	3	4	1	6	3	1
23	12167	12167 mod n	1	2	3	2	5	1	7	8
29	24389	24389 mod n	1	2	1	4	5	1	5	8
31	29791	29791 mod n	1	1	3	1	1	6	7	1
37	50653	50653 mod n	1	1	1	3	1	1	5	1
41	68921	68921 mod n	1	2	1	1	5	6	1	8
43	79507	79507 mod n	1	1	3	2	1	1	3	1
47	103823	103823 mod n	1	2	3	3	5	6	7	8
53	148877	148877 mod n	1	2	1	2	5	1	5	8
59	205379	205379 mod n	1	2	3	4	5	6	3	8
61	226981	226981 mod n	1	1	1	1	1	6	5	1
67	300763	300763 mod n	1	1	3	3	1	1	3	1
71	357911	357911 mod n	1	2	3	1	5	1	7	8

Table 4.4. Residues of cubes of primes under different divisors (9 to 18)

Primes	Cube	Modulo Operation	Residues								
			n=10	*n*=11	*n*=12	*n*=13	*n*=14	*n*=15	*n*=16	*n*=17	*n*=18
2	8	8 mod *n*	8	8	8	8	8	8	8	8	8
3	27	27 mod *n*	7	5	3	1	13	12	11	10	9
5	125	125 mod *n*	5	4	5	8	13	5	13	6	17
7	343	343 mod *n*	3	2	7	5	7	13	7	3	1
11	1331	1331 mod *n*	1	0	11	5	1	11	3	5	17
13	2197	2197 mod *n*	7	8	1	0	13	7	5	4	1
17	4913	4913 mod *n*	3	7	5	12	13	8	1	0	17
19	6859	6859 mod *n*	9	6	7	8	13	4	11	8	1
23	12167	12167 mod *n*	7	1	11	12	1	2	7	12	17
29	24389	24389 mod *n*	9	2	5	1	1	14	5	11	17
31	29791	29791 mod *n*	1	3	7	8	13	1	15	7	1
37	50653	50653 mod *n*	3	9	1	5	1	13	13	10	1
41	68921	68921 mod *n*	1	6	5	8	13	11	9	3	17
43	79507	79507 mod *n*	7	10	7	12	1	7	3	15	1
47	103823	103823 mod *n*	3	5	11	5	13	8	15	4	17
53	148877	148877 mod *n*	7	3	5	1	1	2	13	8	17
59	205379	205379 mod *n*	9	9	11	5	13	14	3	2	17
61	226981	226981 mod *n*	1	7	1	1	13	1	5	14	1
67	300763	300763 mod *n*	3	1	7	8	1	13	11	16	1
71	357911	357911 mod *n*	1	4	11	8	1	11	7	10	17

Above table showed a similar pattern when cubes of primes were divided by 3, 6, 9, 18. Then, it was shown that all cubes of primes excluding 2 and 3 are one more than or one less than a multiple of 3, 6, 9 or 18.

Therefore it can be concluded that all cubes of primes excluding 2 and 3 are one more than or one less than a multiple of 3, 6, 9 or 18.

4.3.2 Algebraic Manipulation of Cubes of Primes

The cube of prime in the each four forms $12m + 1$, $12m + 7$, $12m - 1$, $12m + 5$ are as follows:

1. Modulo 3

$$p^3 = (12m + 1)^3$$
$$= 3(576m^3 + 144m^2 + 12m) + 1$$
$$= 3k_1 + 1$$
$$\equiv 1 \bmod 3$$

$$p^3 = (12m + 7)^3$$
$$= 3(576m^3 + 1008m^2 + 588m + 114) + 1$$
$$= 3k_2 + 1$$
$$\equiv 1 \bmod 3$$

$$p^3 = (12m - 1)^3$$
$$= 3(576m^3 - 144m^2 + 12m) - 1$$
$$= 3k_3 - 1$$
$$\equiv 2 \bmod 3$$

$$p^3 = (12m + 5)^3$$
$$= 3(576m^3 + 720m^2 + 300m + 42) - 1$$
$$= 3k_4 - 1$$
$$\equiv 2 \bmod 3$$

Here all $k_i (i = 1,2,3,4)$ are positive integers.

2. Modulo 6

$$p^3 = (12m + 1)^3$$
$$= 6(288m^3 + 72m^2 + 6m) + 1$$
$$= 6n_1 + 1$$
$$\equiv 1 \bmod 6$$

$p^3 = (12m+7)^3$
$= 6(288m^3 + 504m^2 + 294m + 57) + 1$
$= 6n_2 + 1$
$\equiv 1 \bmod 6$

$p^3 = (12m-1)^3$
$= 6(288m^3 - 72m^2 + 6m) - 1$
$= 6n_3 - 1$
$\equiv 5 \bmod 6$

$p^3 = (12m+5)^3$
$= 6(288m^3 + 360m^2 + 150m + 21) - 1$
$= 6n_4 - 1$
$\equiv 5 \bmod 6$

Here all $n_i (i = 1,2,3,4)$ are positive integers.

3. Modulo 9

$p^3 = (12m+1)^3$
$= 9(192m^3 + 48m^2 + 4m) + 1$
$= 9l_1 + 1$
$\equiv 1 \bmod 9$

$p^3 = (12m+7)^3$
$= 9(192m^3 + 336m^2 + 196m + 38) + 1$
$= 9l_2 + 1$
$\equiv 1 \bmod 9$

$p^3 = (12m-1)^3$
$= 9(192m^3 - 48m^2 + 4m) - 1$
$= 9l_3 - 1$
$\equiv 8 \bmod (9)$

$p^3 = (12m+5)^3$
$= 9(192m^3 + 240m^2 + 100m + 14) - 1$
$= 9l_4 - 1$
$\equiv 8 \bmod (9)$

Here all $l_i (i = 1,2,3,4)$ are positive integers.

4. Modulo 18

$$p^3 = (12m+1)^3$$
$$= 18(96m^3 + 24m^2 + 2m) + 1$$
$$= 18q_1 + 1$$
$$\equiv 1 \bmod 18$$

$$p^3 = (12m+7)^3$$
$$= 18(96m^3 + 168m^2 + 98m + 19) + 1$$
$$= 18q_2 + 1$$
$$\equiv 1 \bmod 18$$

$$p^3 = (12m-1)^3$$
$$= 18(96m^3 - 24m^2 + 2m) - 1$$
$$= 18q_3 - 1$$
$$\equiv 17 \bmod 18$$

$$p^3 = (12m+5)^3$$
$$= 18(96m^3 + 120m^2 + 50m + 7) - 1$$
$$= 18q_4 - 1$$
$$\equiv 17 \bmod 18$$

Here all $q_i (i = 1,2,3,4)$ are positive integers.

Therefore it could be realized that cubes of primes in the form of $6k+1$ are one more than a multiple of 3, 6, 9 or18 and cubes of primes in the form of $6k-1$ are one less than a multiple of 3, 6, 9 or 18. However, this does not work for prime numbers 2 and 3.

Then a prime number could be written in four ways according to these four divisors 3, 6,9 and 18. Those results can be summarized as follows: Let *p* be a prime number. Then it can be represented in following forms:

- $p = (3m \pm 1)^{\frac{1}{3}}$,
- $p = (6m \pm 1)^{\frac{1}{3}}$,
- $p = (9m \pm 1)^{\frac{1}{3}}$,
- $p = (18m \pm 1)^{\frac{1}{3}}$

for some positive integer m.

Therefore it could be checked whether the given number is prime or not using cubes of primes using the following four methods:

- **Method 1**: By checking whether cubes of primes are one more than or one less than a multiple of 3.
- **Method 2**: By checking whether cubes of primes are one more than or one less than a multiple of 6.
- **Method 3**: By checking whether cubes of primes are one more than or one less than a multiple of 9.
- **Method 4**: By checking whether cubes of primes are one more than or one less than a multiple of 18.

It is noteworthy to mentioned that these methods do not work for prime numbers 2 and 3.

4.3.3 Performances of Each Method

4.3.3.1 Time Complexity and Space Complexity

Execution duration in seconds(s) and memory usage in bytes of each method within (0, 1000), (0, 10000), (0, 100000), (0, 500000) and (0, 1000000) ranges were as follows:

Table 4.5. Execution time(s) and memory usage (bytes) of each methods for different input ranges

Methods	Input Ranges	Execution time(s)	Memory Usage(bytes)
Method 1	1000	0.007121	76
	10000	0.033984	112
	100000	0.492691	116
	500000	7.091680	13004
	1000000	20.775451	30753
Method 2	1000	0.001937	76
	10000	0.031688	112
	100000	0.510605	116
	500000	7.080605	27249

	1000000	18.921023	20959
Method 3	1000	0.001911	76
	10000	0,030320	112
	100000	0.470221	116
	500000	5.728774	5917
	1000000	20.586679	21445
Method 4	1000	0.003702	76
	10000	0.033136	112
	100000	0.506140	116
	500000	8.834296	15118
	1000000	19.349590	26910

According to these results of time complexity and space complexity of each methods, it could be concluded that Method 3 with minimum execution time and memory usage, is the best method to test whether the given number is prime or not.

4.3.3.1 Scalability

To interpret scalability, execution time in seconds(s) and memory usage in megabytes (MB) for checking whether the $2^{82589933} - 1$ is prime or not, were as follows:

Table 4.6. Execution time(s) and memory usage (bytes) of each methods for the largest known prime number

Method	Execution time(s)	Memory Usage(MB)
Method 1	320.969723	178.551938
Method 2	329.176947	178.552766
Method 3	320.643212	178.537901
Method 4	325.457568	178.535895

According to these results of scalability, it could be concluded that Method 3 with minimum execution time and memory usage, is the best method to test whether the given number is prime or not.

4.3.4 New Algorithm for Checking Primality

According to above results, steps of the new algorithm created based on method 3 are as follows:

1) Small primes are checked. i.e. If the number n is 2 or 3, then it is prime.
2) Next, The cube of number n is taken. If cube of n is one more than or one less than a multiple of 9, then it is a prime.
3) Next prime divisors up to $\sqrt{n}$ are checked. If any divisors are found, n is not prime.
4) If the corresponding condition was satisfied and no divisors are found, n is prime. Otherwise, it is not prime.

e.g:

- Consider $n = 11$

Then, checking the cube of n;

$11^3 = 1331 = 9\,(142) - 1$

$\therefore n^3$ is one less than multiple of 9.

Therefore 11 is a prime number.

For further checking;

$\sqrt{11} = 3.31$

There are no divisors of n up to $\sqrt{n}$.

It verify that 11 is a prime.

- Consider $n = 81$

$81^3 = 531441 = 9\,(59048) + 9$

$\therefore n^3$ is not one less than or more than multiple of 9.

Therefore 81 is not a prime number.

All primes was in the form of $6k + 1$ or $6k - 1$. But all positive integers which could be written in the form of $6k + 1$ or $6k - 1$ are not prime numbers. So A method had to be used to identify numbers which were not prime numbers, but in the form of $6k + 1$ or $6k - 1$. So basic concept of Seive of Eratosthenes was used for identifying numbers which were not prime numbers, but in the form of $6k + 1$ or $6k - 1$. That was divisors up to $\sqrt{n}$ be considered as composite numbers for a given number n.

4.3.5 Performance of the New Algorithm Vs. Current Algorithms:

Table 4.7. Execution time(s) and memory usage (bytes) of current methods and the new method under different input ranges

Algorithm	Input Ranges	Execution time (s)	Memory Usage (bytes)
Sieve of Eratosthenes	1000	0.004815	8348
	10000	0.004871	80348
	100000	0.017757	800376
	500000	0.090962	40000376
	1000000	0.251724	80000376
Miller-Rabin	1000	0.000176	308
	10000	8.964538	336
	100000	8.535385	364
	500000	8.201599	13796
	1000000	8.177757	37330
	1000	0.002064	8324
	10000	0.020155	80324

Sieve of Atkin	100000	0.220874	800436
	500000	1.085028	4000436
	1000000	2.152453	8000436
New Algorithm	1000	0.001911	76
	10000	0,030320	112
	100000	0.470221	116
	500000	5.728774	5917
	1000000	20.586679	21445

According to above results, there is low memory required for large input size and the algorithm is fast for small, medium and large input ranges, but little bit slow for very large input sizes. However the new algorithm is faster than current methods.

Therefore it could be concluded that this new algorithm is scalable and high efficient.

4.4 Squares of Primes Analysis

4.4.1 First Thousand Primes

The differences of squares of first hundred and last fifty consecutive primes among first thousand primes are as follows:

Table 4.8. Differences of squares of first hundred primes among first thousand primes

n	Prime (P_n)	Square (P_n^2)	$P_n^2 - P_{n-1}^2$	$\frac{P_n^2 - P_{n-1}^2}{24}$	n	Prime (P_n)	Square (P_n^2)	$P_n^2 - P_{n-1}^2$	$\frac{P_n^2 - P_{n-1}^2}{24}$
1	2	4			51	233	54289		
2	3	9	5		52	239	57121	1848	77
3	5	25	16		53	241	58081	2832	118
4	7	49	24	1	54	251	63001	960	40
5	11	121	72	3	55	257	66049	4920	205
6	13	169	48	2	56	263	69169	3048	127
7	17	289	120	5	57	269	72361	3120	130
8	19	361	72	3	58	271	73441	3192	133
9	23	529	168	7	59	277	76729	1080	45
10	29	841	312	13	60	281	78961	3288	137
11	31	961	120	5	61	283	80089	2232	93
12	37	1369	408	17	62	293	85849	1128	47
13	41	1681	312	13	63	307	94249	5760	240
14	43	1849	168	7	64	311	96721	8400	350
15	47	2209	360	15	65	313	97969	2472	103
16	53	2809	600	25	66	317	100489	1248	52
17	59	3481	672	28	67	331	109561	2520	105
18	61	3721	240	10	68	337	113569	9072	378
19	67	4489	768	32	69	347	120409	4008	167

20	71	5041	552	23	70	349	121801	6840	285
21	73	5329	288	12	71	353	124609	1392	58
22	79	6241	912	38	72	359	128881	2808	117
23	83	6889	648	27	73	367	134689	4272	178
24	89	7921	1032	43	74	373	139129	5808	242
25	97	9409	1488	62	75	379	143641	4440	185
26	101	10201	792	33	76	383	146689	4512	188
27	103	10609	408	17	77	389	151321	3048	127
28	107	11449	840	35	78	397	157609	4632	193
29	109	11881	432	18	79	401	160801	6288	262
30	113	12769	888	37	80	409	167281	3192	133
31	127	16129	3360	140	81	419	175561	6480	270
32	131	17161	1032	43	82	421	177241	8280	345
33	137	18769	1608	67	83	431	185761	1680	70
34	139	19321	552	23	84	433	187489	8520	355
35	149	22201	2880	120	85	439	192721	1728	72
36	151	22801	600	25	86	443	196249	5232	218
37	157	24649	1848	77	87	449	201601	3528	147
38	163	26569	1920	80	88	457	208849	5352	223
39	167	27889	1320	55	89	461	212521	7248	302
40	173	29929	2040	85	90	463	214369	3672	153
41	179	32041	2112	88	91	467	218089	1848	77

42	181	32761	720	30	92	479	229441	3720	155
43	191	36481	3720	155	93	487	237169	11352	473
44	193	37249	768	32	94	491	241081	7728	322
45	197	38809	1560	65	95	499	249001	3912	163
46	199	39601	792	33	96	503	253009	7920	330
47	211	44521	4920	205	97	509	259081	4008	167
48	223	49729	5208	217	98	521	271441	6072	253
49	227	51529	1800	75	99	523	273529	12360	515
50	229	52441	912	38	100	541	292681	2088	87

Table 4.9. Differences of squares of last fifty primes among first thousand primes

n	Primes(P_n)	Square(P_n^2)	$P_n^2 - P_{n-1}^2$	$\frac{P_n^2 - P_{n-1}^2}{24}$
951	7507	56355049	120048	5002
952	7517	56505289	150240	6260
953	7523	56595529	90240	3760
954	7529	56685841	90312	3763
955	7537	56806369	120528	5022
956	7541	56866681	60312	2513
957	7547	56957209	90528	3772
958	7549	56987401	30192	1258

959	7559	57138481	151080	6295
960	7561	57168721	30240	1260
961	7573	57350329	181608	7567
962	7577	57410929	60600	2525
963	7583	57501889	90960	3790
964	7589	57592921	91032	3793
965	7591	57623281	30360	1265
966	7603	57805609	182328	7597
967	7607	57866449	60840	2535
968	7621	58079641	213192	8883
969	7639	58354321	274680	11445
970	7643	58415449	61128	2547
971	7649	58507201	91752	3823
972	7669	58813561	306360	12765
973	7673	58874929	61368	2557
974	7681	58997761	122832	5118
975	7687	59089969	92208	3842
976	7691	59151481	61512	2563
977	7699	59274601	123120	5130
978	7703	59336209	61608	2567
979	7717	59552089	215880	8995
980	7723	59644729	92640	3860

981	7727	59706529	61800	2575
982	7741	59923081	216552	9023
983	7753	60109009	185928	7747
984	7757	60171049	62040	2585
985	7759	60202081	31032	1293
986	7789	60668521	466440	19435
987	7793	60730849	62328	2597
988	7817	61105489	374640	15610
989	7823	61199329	93840	3910
990	7829	61293241	93912	3913
991	7841	61481281	188040	7835
992	7853	61669609	188328	7847
993	7867	61889689	220080	9170
994	7873	61984129	94440	3935
995	7877	62047129	63000	2625
996	7879	62078641	31512	1313
997	7883	62141689	63048	2627
998	7901	62425801	284112	11838
999	7907	62520649	94848	3952
1000	7919	62710561	189912	7913

Therefore it could be concluded that the greatest common divisor of difference of squares of first thousand consecutive primes was 24.

4.4.2 New Model for Generating the Next Primes

According to above results, a model for finding the next prime after a given prime was developed as following:

$$P_n = \sqrt{24k + P_{n-1}^2} \ ,$$

where $k \in \mathbb{Z}^+$.

This model was not for the small primes (2, 3 and 5).

This model can be used for finding the next prime number after a given prime number by checking primality of P_n for different k values, using the new primality checking algorithm(Cubes of primes (excluding primes 2,3) are one more than or one less than a multiple of 9.).

4.4.3 Last Ten Known Mersenne Primes

The difference of squares of the largest ten known consecutive mersenne primes areas follows:

Table 4.10. Differences of squares of the largest ten known consecutive mersenne primes

n	n^{th} **Mersenne Prime** (a_n)	$a_n^2 - a_{n-1}^2$	$\frac{a_n^2 - a_{n-1}^2}{2^{937616}}$
41	$2^{24036583} - 1$		
42	$2^{25964951} - 1$	$2^{8875012}$	$2^{2919120}$
43	$2^{30402457} - 1$	$2^{8875012}$	$2^{7937396}$
44	$2^{32582657} - 1$	$2^{4360400}$	$2^{3422784}$
45	$2^{37156667} - 1$	$2^{9148020}$	$2^{8210404}$
46	$2^{42643801} - 1$	$2^{10974268}$	$2^{10036652}$
47	$2^{43112609} - 1$	2^{937616}	1
48	$2^{57885161} - 1$	$2^{29545104}$	$2^{28607488}$

49	$2^{74207281}-1$	$2^{32644240}$	$2^{31706624}$
50	$2^{77232917}-1$	$2^{6051272}$	$2^{5113656}$
51	$2^{82589933}-1$	$2^{10714032}$	$2^{9776416}$

Therefore it could be seen that the greatest common divisor of difference of squares of the largest ten known consecutive mersenne primes is 2^{937616}.

4.4.4 New Model for Discovering the Next Largest Unknown Prime

According to above results, a model for discovering the next largest unknown prime after the last known prime $2^{82589933}-1$ is developed as follows:

$$a_n = \sqrt{2^{937616}\,k + a_{n-1}^2}\,,$$

where $a_{n-1} = 2^{82589933} - 1$ and $k \in \mathbb{Z}^+$.

This model can be used for discovering the next largest unknown prime after the last known largest prime $2^{82589933} - 1$. This model can be used for finding a_n for different *k* s. And the new primality checking algorithm [Cubes of primes (excluding primes 2, 3) are one more than or one less than a multiple of 9.] can be used to check whether the a_n is a prime or not for different *k* s.

CHAPTER 5: CONCLUSION

This research project introduces innovative methods for prime number identification and generation, contributing significantly to both theoretical and computational number theory. The development of a novel primality testing algorithm, along with a model for generating the next prime number, addresses key challenges in the next largest unknown prime number discovery and demonstrates substantial improvements over traditional algorithms.

The new primality testing algorithm, based on modular arithmetic properties of cubed primes, leverages the behavior of primes in the form of $6k + 1$ and $6k - 1$under various modulars. Through extensive testing, it was found that cubes of primes (excluding 2 and 3) are one more than or one less than multiple of 9, depending on whether they are in the form of $6k + 1$ or $6k - 1$. This pattern formed the basis for an efficient primality test, offering a scalable alternative to classical algorithms such as the Sieve of Eratosthenes, Miller-Rabin, and Sieve of Atkin.

The research demonstrated that Method 3, which checks cubes of primes using mod 9, was the most efficient in terms of execution time and memory usage across various input sizes. This method outperformed other approaches, proving particularly effective for large primes. It was also scalable, as shown by its successful application to very large primes, such as Mersenne primes, where it demonstrated minimal resource usage while maintaining accuracy and speed. The new algorithm was required significantly less memory compared to traditional algorithms, making it well-suited for computational tasks involving large prime numbers, particularly in cryptography.

In addition to the primality test, this study developed a novel predictive model for generating the next prime after a given prime number. By analyzing the difference between the squares of consecutive primes, the research identified a pattern that allowed the development of a formula to predict subsequent primes. This model was further tested on the first thousand primes and the last ten known Mersenne primes, confirming its accuracy and reliability. The predictive model is particularly valuable in the field of cryptography, where large primes are essential for secure encryption systems.

The graphical analysis of prime distributions, prime gaps, twin primes, and the growth of squared and cubes of primes provided additional insights into the behavior of primes. These visualizations helped to reveal trends and patterns that further supported the development of the primality test and predictive model. The increasing gaps between consecutive primes, as well as the rapid growth of squared and cube of primes, highlight the complexity of prime number distribution, especially as numbers grow larger.

Overall, the contributions of this research are multifaceted. The new primality testing algorithm offers a more efficient and scalable solution for prime identification, particularly for large numbers. The predictive model for generating the next prime extends the theoretical understanding of prime number behavior and provides practical tools for the finding the next largest unknown prime based on the largest prime found up to now. The combination of these methodologies addresses longstanding challenges in number theory and offers advancements in computational efficiency, making them highly applicable in fields such as cryptography, secure communications, and mathematical research.

The success of this project enhances the importance of continued exploration into the properties of prime numbers. The findings provide a foundation for future research, particularly in refining the predictive model and the primality testing algorithm. As computational power increases and the need for larger primes grows, these methods will become relevant in both theoretical and practical applications.

This research makes significant contributions to prime number theory, offering innovative solutions that are both theoretically sound and computationally efficient. The proposed methodologies not only enhance prime discovery but also open new avenues for research, with implications for cryptography, computer science, and secure communication systems.

REFERENCES

1. Burton, D. M. (2011). *Elementary number theory*. Mcgraw-Hill.
2. Geeksforgeeks.org. (2012). Sieve of Eratosthenes, Retrieved from https://www.geeksforgeeks.org/sieve-of-eratosthenes/ (2024.06.14)
3. Geeksforgeeks.org (2016). Sieve of Atkin, Retrieved from https://www.geeksforgeeks.org/sieve-of-atkin/ (2024.06.14)
4. Ginni. (2022). What are the Miller-Rabin Algorithm for testing the primality of a given number? https://www.tutorialspoint.com/what-are-the-miller-rabin-algorithm-for-testing-the-primality-of-a-given-number
5. GIMPS. (n.d.). List of known Mersenne prime numbers - PrimeNet. https://www.mersenne.org/primes/
6. Kleinberg, B. (2010). Introduction to Algorithms (CS 482). https://www.cs.cornell.edu/courses/cs4820/2019sp/handouts/MillerRabin.pdf
7. Lazer. (2024). Time complexity of Sieve of Eratosthenes algorithm. Stack Overflow. https://stackoverflow.com/questions/2582732/time-complexity-of-sieve-of-eratosthenes-algorithm
8. Lynn, B. (2019). Number Theory - Primality Tests. Stanford.edu. https://crypto.stanford.edu/pbc/notes/numbertheory/millerrabin.html
9. Okeke, C. (2023, July 16). Introduction to BIG O Notation — Time and Space Complexity. Medium. https://medium.com/@DevChy/introduction-to-big-o-notation-time-and-space-complexity-f747ea5bca58
10. Parker, M. (2024). - *YouTube*. Youtu.be.

https://youtu.be/ZMkIiFs35HQ?si=DsnDjy83fdGCr6ty

11. Porras, W. (2022). Study about the Pattern of Prime Numbers. *Book Publisher International (a Part of SCIENCEDOMAIN International)*, *9*, 83–93. https://doi.org/10.9734/bpi/ramrcs/v9/15662d
12. Weisstein, E. W. (2024, October 10). Mersenne Prime. Retrieved from mathworld.wolfram.com website: https://mathworld.wolfram.com/MersennePrime.html
13. Wrentz, C. (2021). Sieve of Atkin — The Theoretical Optimization of Prime Number Generation. https://medium.com/smucs/sieve-of-atkin-the-theoretical-optimization-of-prime-number-generation-e47107d61e28
14. Zaman, B. U. (2023). Prime Discovery A Formula Generating Primes and Their Composites. *INDIGO (University of Illinois at Chicago)*. https://doi.org/10.36227/techrxiv.24416455.v1

APPENDIX

- **Dataset**

2	547	1229	1993	2753	3583	4423	5297	6151	7013
3	557	1231	1997	2767	3593	4441	5303	6163	7019
5	563	1237	1999	2777	3607	4447	5309	6173	7027
7	569	1249	2003	2789	3613	4451	5323	6197	7039
11	571	1259	2011	2791	3617	4457	5333	6199	7043
13	577	1277	2017	2797	3623	4463	5347	6203	7057
17	587	1279	2027	2801	3631	4481	5351	6211	7069
19	593	1283	2029	2803	3637	4483	5381	6217	7079
23	599	1289	2039	2819	3643	4493	5387	6221	7103
29	601	1291	2053	2833	3659	4507	5393	6229	7109
31	607	1297	2063	2837	3671	4513	5399	6247	7121
37	613	1301	2069	2843	3673	4517	5407	6257	7127
41	617	1303	2081	2851	3677	4519	5413	6263	7129
43	619	1307	2083	2857	3691	4523	5417	6269	7151
47	631	1319	2087	2861	3697	4547	5419	6271	7159
53	641	1321	2089	2879	3701	4549	5431	6277	7177
59	643	1327	2099	2887	3709	4561	5437	6287	7187
61	647	1361	2111	2897	3719	4567	5441	6299	7193
67	653	1367	2113	2903	3727	4583	5443	6301	7207
71	659	1373	2129	2909	3733	4591	5449	6311	7211
73	661	1381	2131	2917	3739	4597	5471	6317	7213
79	673	1399	2137	2927	3761	4603	5477	6323	7219
83	677	1409	2141	2939	3767	4621	5479	6329	7229
89	683	1423	2143	2953	3769	4637	5483	6337	7237
97	691	1427	2153	2957	3779	4639	5501	6343	7243
101	701	1429	2161	2963	3793	4643	5503	6353	7247
103	709	1433	2179	2969	3797	4649	5507	6359	7253
107	719	1439	2203	2971	3803	4651	5519	6361	7283
109	727	1447	2207	2999	3821	4657	5521	6367	7297
113	733	1451	2213	3001	3823	4663	5527	6373	7307
127	739	1453	2221	3011	3833	4673	5531	6379	7309
131	743	1459	2237	3019	3847	4679	5557	6389	7321
137	751	1471	2239	3023	3851	4691	5563	6397	7331
139	757	1481	2243	3037	3853	4703	5569	6421	7333
149	761	1483	2251	3041	3863	4721	5573	6427	7349
151	769	1487	2267	3049	3877	4723	5581	6449	7351
157	773	1489	2269	3061	3881	4729	5591	6451	7369
163	787	1493	2273	3067	3889	4733	5623	6469	7393
167	797	1499	2281	3079	3907	4751	5639	6473	7411

173	809-	1511	2287	3083	3911	4759	5641	6481	7417
179	811	1523	2293	3089	3917	4783	5647	6491	7433
181	821	1531	2297	3109	3919	4787	5651	6521	7451
191	823	1543	2309	3119	3923	4789	5653	6529	7457
193	827	1549	2311	3121	3929	4793	5657	6547	7459
197	829	1553	2333	3137	3931	4799	5659	6551	7477
199	839	1559	2339	3163	3943	4801	5669	6553	7481
211	853	1567	2341	3167	3947	4813	5683	6563	7487
223	857	1571	2347	3169	3967	4817	5689	6569	7489
227	859	1579	2351	3181	3989	4831	5693	6571	7499
229	863	1583	2357	3187	4001	4861	5701	6577	7507
233	877	1597	2377	3191	4003	4871	5711	6581	7517
239	881	1601	2381	3203	4007	4877	5717	6599	7523
241	883	1607	2383	3209	4013	4889	5737	6607	7529
251	887	1609	2389	3217	4019	4903	5741	6619	7537
257	907	1613	2393	3221	4021	4909	5743	6637	7541
263	911	1619	2399	3229	4027	4919	5749	6653	7547
269	919	1621	2411	3251	4049	4931	5779	6659	7549
271	929	1627	2417	3253	4051	4933	5783	6661	7559
277	937	1637	2423	3257	4057	4937	5791	6673	7561
281	941	1657	2437	3259	4073	4943	5801	6679	7573
283	947	1663	2441	3271	4079	4951	5807	6689	7577
293	953	1667	2447	3299	4091	4957	5813	6691	7583
307	967	1669	2459	3301	4093	4967	5821	6701	7589
311	971	1693	2467	3307	4099	4969	5827	6703	7591
313	977	1697	2473	3313	4111	4973	5839	6709	7603
317	983	1699	2477	3319	4127	4987	5843	6719	7607
331	991	1709	2503	3323	4129	4993	5849	6733	7621
337	997	1721	2521	3329	4133	4999	5851	6737	7639
347	1009	1723	2531	3331	4139	5003	5857	6761	7643
349	1013	1733	2539	3343	4153	5009	5861	6763	7649
353	1019	1741	2543	3347	4157	5011	5867	6779	7669
359	1021	1747	2549	3359	4159	5021	5869	6781	7673
367	1031	1753	2551	3361	4177	5023	5879	6791	7681
373	1033	1759	2557	3371	4201	5039	5881	6793	7687
379	1039	1777	2579	3373	4211	5051	5897	6803	7691
383	1049	1783	2591	3389	4217	5059	5903	6823	7699
389	1051	1787	2593	3391	4219	5077	5923	6827	7703
397	1061	1789	2609	3407	4229	5081	5927	6829	7717
401	1063	1801	2617	3413	4231	5087	5939	6833	7723
409	1069	1811	2621	3433	4241	5099	5953	6841	7727
419	1087	1823	2633	3449	4243	5101	5981	6857	7741
421	1091	1831	2647	3457	4253	5107	5987	6863	7753

431	1093	1847	2657	3461	4259	5113	6007	6869	7757
433	1097	1861	2659	3463	4261	5119	6011	6871	7759
439	1103	1867	2663	3467	4271	5147	6029	6883	7789
443	1109	1871	2671	3469	4273	5153	6037	6899	7793
449	1117	1873	2677	3491	4283	5167	6043	6907	7817
457	1123	1877	2683	3499	4289	5171	6047	6911	7823
461	1129	1879	2687	3511	4297	5179	6053	6917	7829
463	1151	1889	2689	3517	4327	5189	6067	6947	7841
467	1153	1901	2693	3527	4337	5197	6073	6949	7853
479	1163	1907	2699	3529	4339	5209	6079	6959	7867
487	1171	1913	2707	3533	4349	5227	6089	6961	7873
491	1181	1931	2711	3539	4357	5231	6091	6967	7877
499	1187	1933	2713	3541	4363	5233	6101	6971	7879
503	1193	1949	2719	3547	4373	5237	6113	6977	7883
509	1201	1951	2729	3557	4391	5261	6121	6983	7901
521	1213	1973	2731	3559	4397	5273	6131	6991	7907
523	1217	1979	2741	3571	4409	5279	6133	6997	7013
541	1223	1987	2749	3581	4421	5281	6143	7001	7019

- **Python Syntax**

Computation Memory Usage and Execution Time of Current Algorithms:

```
import time
import memory_profiler

import psutil

import random

import math

 # Sieve of Eratosthenes
def sieve_of_eratosthenes(n):

    sieve = [True] * (n + 1)

    sieve[0] = sieve[1] = False

    for i in range(2, int(n**0.5) + 1):

        if sieve[i]:

            for j in range(i*i, n + 1, i):

                sieve[j] = False

    return [x for x in range(n + 1) if sieve[x]]
```

```python
# Miller-Rabin Primality Test
def miller_rabin(n, k=5):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False

    def is_composite(a, d, n, s):
        if pow(a, d, n) == 1:
            return False
        for i in range(s):
            if pow(a, d * 2**i, n) == n - 1:
                return False
        return True

    d, s = n - 1, 0
    while d % 2 == 0:
        d //= 2
        s += 1

    for _ in range(k):
        a = random.randint(2, n - 2)
        if is_composite(a, d, n, s):
            return False
    return True
```

```
# Sieve of Atkin
def sieve_of_atkin(limit):
    if limit < 2:
        return []

    sieve = [False] * (limit + 1)
    for x in range(1, int(limit**0.5) + 1):
        for y in range(1, int(limit**0.5) + 1):
            n = 4*x**2 + y**2
            if n <= limit and (n % 12 == 1 or n % 12 == 5):
                sieve[n] = not sieve[n]
            n = 3*x**2 + y**2
            if n <= limit and n % 12 == 7:
                sieve[n] = not sieve[n]
            n = 3*x**2 - y**2
            if x > y and n <= limit and n % 12 == 11:
                sieve[n] = not sieve[n]
    for n in range(5, int(limit**0.5) + 1):
        if sieve[n]:
            for k in range(n**2, limit + 1, n**2):
                sieve[k] = False
    return [2, 3] + [x for x in range(5, limit + 1) if sieve[x]]
 @memory_profiler.profile
def measure_sieve():
    start_time = time.time()
    primes = sieve_of_eratosthenes(1000000)
    end_time = time.time()
    execution_time = end_time - start_time
    print(f"Execution Time: {execution_time} seconds")

@memory_profiler.profile
def measure_miller_rabin():
```

```
    start_time = time.time()
    result = miller_rabin(1000000)
    end_time = time.time()
    execution_time = end_time - start_time
    print(f"Execution Time: {execution_time} seconds")


@memory_profiler.profile
def measure_atkin():
    start_time = time.time()
    primes = sieve_of_atkin(1000000)
    end_time = time.time()
    execution_time = end_time - start_time
    print(f"Execution Time: {execution_time} seconds")

# Run the measurements
measure_sieve()
measure_miller_rabin()
measure_atkin()


def benchmark_algorithm(algorithm, input_sizes):
    results = []

    for size in input_sizes:
        start_time = time.time()
        mem_before = psutil.Process().memory_info().rss
        algorithm(size)
        mem_after = psutil.Process().memory_info().rss
        end_time = time.time()

        execution_time = end_time - start_time
        memory_usage = mem_after - mem_before

        results.append({
            'InputSize': size,
            'ExecutionTime': execution_time,
            'MemoryUsage': memory_usage
        })

    return results

input_sizes = [1000, 10000, 100000, 500000, 1000000]

# Benchmarking results
```

```
sieve_results = benchmark_algorithm(sieve_of_eratosthenes,
input_sizes)
miller_rabin_results = benchmark_algorithm(miller_rabin, input_sizes)
atkin_results = benchmark_algorithm(sieve_of_atkin, input_sizes)

print(sieve_results)
print(miller_rabin_results)
print(atkin_results)
```

Graphs

Prime Distribution

```
!pip install matplotlib
!pip install numpy
import matplotlib.pyplot as plt
import numpy as np
# Prime Distribution
plt.figure(figsize=(10, 6))
plt.scatter(range(len(primes)), primes, s=10, c='blue')
plt.title('Prime Distribution')
plt.xlabel('Index')
plt.ylabel('Prime Number')
plt.show()
```

```
import matplotlib.pyplot as plt

# Function to check if a number is prime
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

# Generate the first 1000 prime numbers
primes = []
num = 2  # Starting from the first prime number
while  len(primes) < 100:
    if is_prime(num):
```

```
        primes.append(num)
    num += 1

# Plotting the prime numbers
plt.figure(figsize=(10, 6))
plt.plot(primes, marker='o', linestyle='-', markersize=3)
plt.title('First 100 Prime Numbers')
plt.xlabel('Index')
plt.ylabel('Prime Number')
plt.grid()
plt.show()
```

```
import matplotlib.pyplot as plt

def is_prime(num):
  """Checks if a number is prime."""
  if num <= 1:
    return False
  if num <= 3:
    return True
  if num % 2 == 0 or num % 3 == 0:
    return False
  i = 5
  while i * i <= num:
    if num % i == 0 or num % (i + 2) == 0:
      return False
    i += 6
  return True

def find_primes(n):
  """Finds the first n prime numbers."""
  primes = []
  num = 2
  while len(primes) < n:
    if is_prime(num):
      primes.append(num)
    num += 1
  return primes

# Find the first 1000 prime numbers
primes = find_primes(1000)

# Get the last 100 prime numbers
last_100_primes = primes[-100:]
```

```
# Plot the last 100 prime numbers
plt.figure(figsize=(10, 6))
plt.plot(last_100_primes, marker='o', linestyle='-', markersize=3)
plt.xlabel('Prime Number Index')
plt.ylabel('Prime Number')
plt.title('Last 100 Prime Numbers of First 1000 Prime Numbers')
plt.show()
```

Prime Gap

```
# Prime Gap
prime_gaps = [primes[i+1] - primes[i] for i in range(len(primes)-1)]
plt.figure(figsize=(10, 6))
plt.scatter(range(len(prime_gaps)),prime_gaps, s=10, linestyle='-
',c='red')
plt.title('Prime Gap Distribution')
plt.xlabel('Index')
plt.ylabel('Prime Gap')
plt.show()
```

Twin Prime:

```
# Twin Primes
twin_primes = [(primes[i], primes[i+1]) for i in range(len(primes)-1)
if primes[i+1] - primes[i] == 2]
twin_prime_indices = [i for i in range(len(primes)-1) if primes[i+1] -
primes[i] == 2]
plt.figure(figsize=(10, 6))
plt.scatter(twin_prime_indices, [p[0] for p in twin_primes], s=10,
c='green', label='First Twin Prime')
plt.scatter(twin_prime_indices, [p[1] for p in twin_primes], s=10,
c='orange', label='Second Twin Prime')
plt.title('Twin Primes Distribution')
plt.xlabel('Index')
plt.ylabel('Prime Numbers')
plt.legend()
plt.show()
```

Squaring Prime

```
# Squaring Primes
squared_primes = [p**2 for p in primes]
```

```
plt.figure(figsize=(10, 6))
plt.scatter(range(len(squared_primes)), squared_primes, s=10,
c='purple')
plt.title('Squared Primes Distribution')
plt.xlabel('Index')
plt.ylabel('Squared Prime Numbers')
plt.show()
```

Cubes of Prime

```
# Cubes of Primes
cubed_primes = [p**3 for p in primes]
plt.figure(figsize=(10, 6))
plt.scatter(range(len(cubed_primes)), cubed_primes, s=10, c='brown')
plt.title('Cubed Primes Distribution')
plt.xlabel('Index')
plt.ylabel('Cubed Prime Numbers')
plt.show()
```

Computing Residues

```
import sympy
def calculate_cubes_modulo(N, m):
    # Generate the first N primes
    primes = list(sympy.primerange(1, sympy.prime(N) + 1))

    # Calculate cubes and their residues modulo m
    cubes_mod_m = [(p, p**3 % m) for p in primes]

    # Print the results
    print(f"\n\nPrimes and their cubes modulo {m}:")
    for prime, residue in cubes_mod_m:
        print(f"Prime: {prime}, Cube: {prime**3} mod {m}: {residue}")

# Parameters
N = 50  # Number of prime numbers
# m = 7 # Modulus

# # Calculate and print the cubes modulo m
# calculate_cubes_modulo(N, m)

for i in range(2,51):
  calculate_cubes_modulo(N,i)
```

Checking Execution time and Memory Usage

Method 1

```
import time
import tracemalloc
import math

def is_probably_prime(n):
    # Step 1: Check for small primes
    if n in (2, 3):
        return True

    # Step 2: Check divisibility by 2 or 3
    if n % 2 == 0 or n % 3 == 0:
        return False

    # Step 3: Determine if n can be expressed as 6k±1
    if (n - 1) % 6 == 0:
        k_form = "6k + 1"
    elif (n + 1) % 6 == 0:
        k_form = "6k - 1"
    else:
        return False

    # Step 4: Compute n^3 % 3
    n_cubed_mod_3 = (n ** 3) % 3

    # Step 5: Check the congruence condition
    if k_form == "6k + 1" and n_cubed_mod_3 == 1:
        form_check = True
    elif k_form == "6k - 1" and n_cubed_mod_3 == 2: # since -1 mod 3
is 2
        form_check = True
    else:
        return False

    # Step 6: Trial division up to √n
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6

    return True
```

```
def measure_range_performance(range_start, range_end):
    # Start measuring time
    start_time = time.time()

    # Start measuring memory
    tracemalloc.start()

    # Run the algorithm for all numbers in the range
    primes = []
    for n in range(range_start, range_end):
        if is_probably_prime(n):
            primes.append(n)

    # Stop measuring memory
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    # Stop measuring time
    end_time = time.time()

    # Calculate execution time and memory usage
    execution_time = end_time - start_time
    memory_usage = peak - current

    return execution_time, memory_usage, len(primes)

# Define ranges
ranges = [(0, 1000), (0, 10000), (0, 100000), (0, 500000), (0,
1000000)]

# Measure performance for each range
for range_start, range_end in ranges:
    exec_time, mem_usage, prime_count =
measure_range_performance(range_start, range_end)
    print(f"Range ({range_start}, {range_end}):")
    print(f"  Execution Time: {exec_time:.8f} seconds")
    print(f"  Memory Usage: {mem_usage} bytes")
    print(f"  Primes Found: {prime_count}\n")
```

Method 2

```
import time
import tracemalloc
import math
```

```
def is_probably_prime(n):
    # Step 1: Check for small primes
    if n in (2, 3):
        return True

    # Step 2: Check divisibility by 2 or 3
    if n % 2 == 0 or n % 3 == 0:
        return False

    # Step 3: Determine if n can be expressed as 6k±1
    if (n - 1) % 6 == 0:
        k_form = "6k + 1"
    elif (n + 1) % 6 == 0:
        k_form = "6k - 1"
    else:
        return False

    # Step 4: Compute n^3 % 6
    n_cubed_mod_6 = (n ** 3) % 6

    # Step 5: Check the congruence condition
    if k_form == "6k + 1" and n_cubed_mod_6 == 1:
        form_check = True
    elif k_form == "6k - 1" and n_cubed_mod_6 == 5:  # since -1 mod 6
is 5
        form_check = True
    else:
        return False

    # Step 6: Trial division up to √n
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6

    return True

def measure_range_performance(range_start, range_end):
    # Start measuring time
    start_time = time.time()

    # Start measuring memory
    tracemalloc.start()
```

```
    # Run the algorithm for all numbers in the range
    primes = []
    for n in range(range_start, range_end):
        if is_probably_prime(n):
            primes.append(n)

    # Stop measuring memory
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    # Stop measuring time
    end_time = time.time()

    # Calculate execution time and memory usage
    execution_time = end_time - start_time
    memory_usage = peak - current

    return execution_time, memory_usage, len(primes)

# Define ranges
ranges = [(0, 1000), (0, 10000), (0, 100000), (0, 500000), (0,
1000000)]

# Measure performance for each range
for range_start, range_end in ranges:
    exec_time, mem_usage, prime_count =
measure_range_performance(range_start, range_end)
    print(f"Range ({range_start}, {range_end}):")
    print(f"  Execution Time: {exec_time:.8f} seconds")
    print(f"  Memory Usage: {mem_usage} bytes")
    print(f"  Primes Found: {prime_count}\n")
```

Method 3

```
import time
import tracemalloc
import math

def is_probably_prime(n):
    # Step 1: Check for small primes
    if n in (2, 3):
        return True

    # Step 2: Check divisibility by 2 or 3
    if n % 2 == 0 or n % 3 == 0:
```

```
        return False

    # Step 3: Determine if n can be expressed as 6k±1
    if (n - 1) % 6 == 0:
        k_form = "6k + 1"
    elif (n + 1) % 6 == 0:
        k_form = "6k - 1"
    else:
        return False

    # Step 4: Compute n^3 % 9
    n_cubed_mod_9 = (n ** 3) % 9

    # Step 5: Check the congruence condition
    if k_form == "6k + 1" and n_cubed_mod_9 == 1:
        form_check = True
    elif k_form == "6k - 1" and n_cubed_mod_9 == 8:  # since -1 mod 9
is 8
        form_check = True
    else:
        return False

    # Step 6: Trial division up to √n
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6

    return True

def measure_range_performance(range_start, range_end):
    # Start measuring time
    start_time = time.time()

    # Start measuring memory
    tracemalloc.start()

    # Run the algorithm for all numbers in the range
    primes = []
    for n in range(range_start, range_end):
        if is_probably_prime(n):
            primes.append(n)

    # Stop measuring memory
```

```
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    # Stop measuring time
    end_time = time.time()

    # Calculate execution time and memory usage
    execution_time = end_time - start_time
    memory_usage = peak - current

    return execution_time, memory_usage, len(primes)

# Define ranges
ranges = [(0, 1000), (0, 10000), (0, 100000), (0, 500000), (0,
1000000)]

# Measure performance for each range
for range_start, range_end in ranges:
    exec_time, mem_usage, prime_count =
measure_range_performance(range_start, range_end)
    print(f"Range ({range_start}, {range_end}):")
    print(f"  Execution Time: {exec_time:.8f} seconds")
    print(f"  Memory Usage: {mem_usage} bytes")
    print(f"  Primes Found: {prime_count}\n")
```

Method 4

```
import time
import tracemalloc
import math

def is_probably_prime(n):
    # Step 1: Check for small primes
    if n in (2, 3):
        return True

    # Step 2: Check divisibility by 2 or 3
    if n % 2 == 0 or n % 3 == 0:
        return False

    # Step 3: Determine if n can be expressed as 6k±1
    if (n - 1) % 6 == 0:
        k_form = "6k + 1"
    elif (n + 1) % 6 == 0:
        k_form = "6k - 1"
```

```
    else:
        return False

    # Step 4: Compute n^3 % 18
    n_cubed_mod_18 = (n ** 3) % 18

    # Step 5: Check the congruence condition
    if k_form == "6k + 1" and n_cubed_mod_18 == 1:
        form_check = True
    elif k_form == "6k - 1" and n_cubed_mod_18 == 17: # since -1 mod
18 is 17
        form_check = True
    else:
        return False

    # Step 6: Trial division up to √n
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6

    return True

def measure_range_performance(range_start, range_end):
    # Start measuring time
    start_time = time.time()

    # Start measuring memory
    tracemalloc.start()

    # Run the algorithm for all numbers in the range
    primes = []
    for n in range(range_start, range_end):
        if is_probably_prime(n):
            primes.append(n)

    # Stop measuring memory
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()

    # Stop measuring time
    end_time = time.time()

    # Calculate execution time and memory usage
```

```
    execution_time = end_time - start_time
    memory_usage = peak - current

    return execution_time, memory_usage, len(primes)

# Define ranges
ranges = [(0, 1000), (0, 10000), (0, 100000), (0, 500000), (0,
1000000)]

# Measure performance for each range
for range_start, range_end in ranges:
    exec_time, mem_usage, prime_count =
measure_range_performance(range_start, range_end)
    print(f"Range ({range_start}, {range_end}):")
    print(f"  Execution Time: {exec_time:.8f} seconds")
    print(f"  Memory Usage: {mem_usage} bytes")
    print(f"  Primes Found: {prime_count}\n")
```

Checking Scalability

Method 1

```
import time
import tracemalloc

# Function to check primality using Method 1
def is_prime_method_1(n):
    # Calculate the cube of the number
    n_cubed = n ** 3

    # Check if n is in the form of 6k+1 or 6k-1
    if n % 6 == 1:
        return (n_cubed - 1) % 3 == 0
    elif n % 6 == 5:
        return (n_cubed + 1) % 3 == 0
    return False

# Start measuring time
start_time = time.time()

# Start measuring memory
tracemalloc.start()

# Calculate the number 2^82,589,933 - 1
number = 2**82589933 - 1
```

```
# Check if the number is prime using Method 1
is_prime = is_prime_method_1(number)

# Stop measuring memory
current, peak = tracemalloc.get_traced_memory()

# Stop measuring time
end_time = time.time()

# Calculate time and memory usage
execution_time = end_time - start_time
memory_usage = peak / (1024 * 1024)  # Convert from bytes to MB

# Print the results
print(f"The number 2^82,589,933 - 1 is {'prime' if is_prime else 'not
prime'} according to Method 1.")
print(f"Execution time: {execution_time} seconds")
print(f"Memory usage: {memory_usage} MB")

# Stop the memory tracing
tracemalloc.stop()
```

Method 2

```
import time
import tracemalloc

# Function to check primality using Method 2
def is_prime_method_2(n):
    # Calculate the cube of the number
    n_cubed = n ** 3

    # Check if n is in the form of 6k+1 or 6k-1
    if n % 6 == 1:
        return (n_cubed - 1) % 6 == 0
    elif n % 6 == 5:
        return (n_cubed + 1) % 6 == 0
    return False

# Start measuring time
start_time = time.time()

# Start measuring memory
```

```
tracemalloc.start()

# Calculate the number 2^82,589,933 - 1
number = 2**82589933 - 1

# Check if the number is prime using Method 2
is_prime = is_prime_method_2(number)

# Stop measuring memory
current, peak = tracemalloc.get_traced_memory()

# Stop measuring time
end_time = time.time()

# Calculate time and memory usage
execution_time = end_time - start_time
memory_usage = peak / (1024 * 1024)  # Convert from bytes to MB

# Print the results
print(f"The number 2^82,589,933 - 1 is {'prime' if is_prime else 'not
prime'} according to Method 2.")
print(f"Execution time: {execution_time} seconds")
print(f"Memory usage: {memory_usage} MB")

# Stop the memory tracing
tracemalloc.stop()
```

Method 3

```
import time
import tracemalloc

# Function to check primality using Method 3
def is_prime_method_3(n):
    # Calculate the cube of the number
    n_cubed = n ** 3

    # Check if n is in the form of 6k+1 or 6k-1
    if n % 6 == 1:
        return (n_cubed - 1) % 9 == 0
    elif n % 6 == 5:
        return (n_cubed + 1) % 9 == 0
    return False

# Start measuring time
```

```
start_time = time.time()

# Start measuring memory
tracemalloc.start()

# Calculate the number 2^82,589,933 - 1
number = 2**82589933 - 1

# Check if the number is prime using Method 3
is_prime = is_prime_method_3(number)

# Stop measuring memory
current, peak = tracemalloc.get_traced_memory()

# Stop measuring time
end_time = time.time()

# Calculate time and memory usage
execution_time = end_time - start_time
memory_usage = peak / (1024 * 1024)  # Convert from bytes to MB

# Print the results
print(f"The number 2^82,589,933 - 1 is {'prime' if is_prime else 'not
prime'} according to Method 3.")
print(f"Execution time: {execution_time} seconds")
print(f"Memory usage: {memory_usage} MB")

# Stop the memory tracing
tracemalloc.stop()
```

Method 4

```
import time
import tracemalloc

# Function to check primality using Method 4
def is_prime_method_4(n):
    # Calculate the cube of the number
    n_cubed = n ** 3

    # Check if n is in the form of 6k+1 or 6k-1
    if n % 6 == 1:
        return (n_cubed - 1) % 18 == 0
    elif n % 6 == 5:
        return (n_cubed + 1) % 18 == 0
```

```
    return False

# Start measuring time
start_time = time.time()

# Start measuring memory
tracemalloc.start()

# Calculate the number 2^82,589,933 - 1
number = 2**82589933 - 1

# Check if the number is prime using Method 4
is_prime = is_prime_method_4(number)

# Stop measuring memory
current, peak = tracemalloc.get_traced_memory()

# Stop measuring time
end_time = time.time()

# Calculate time and memory usage
execution_time = end_time - start_time
memory_usage = peak / (1024 * 1024)  # Convert from bytes to MB

# Print the results
print(f"The number 2^82,589,933 - 1 is {'prime' if is_prime else 'not
prime'} according to Method 4.")
print(f"Execution time: {execution_time} seconds")
print(f"Memory usage: {memory_usage} MB")

# Stop the memory tracing
tracemalloc.stop()
```

Testing New Algorithm

```
import math

def is_prime(n):
    # Step 1: Handle small primes
    if n == 2 or n == 3:
        return True

    # Step 2: Check for divisibility by 2 and 3
    if n % 2 == 0 or n % 3 == 0:
        return False
```

```
    # Step 3: Determine if n is of the form 6k + 1 or 6k - 1
    if (n - 1) % 6 == 0:
        k_form = "6k + 1"
    elif (n + 1) % 6 == 0:
        k_form = "6k - 1"
    else:
        return False

    # Step 4: Compute n^3 % 18
    n_cubed_mod_18 = (n ** 3) % 18

    # Step 5: Check the congruence condition
    if k_form == "6k + 1" and n_cubed_mod_18 == 1:
        form_check = True
    elif k_form == "6k - 1" and n_cubed_mod_18 == 17: # since -1 mod
18 is 17
        form_check = True
    else:
        return False

    # Step 6: Trial division up to √n
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6

    return True

# Example usage
test_numbers = [7879,7883,7885,7901,7907,7919  ]

results = {num: is_prime(num) for num in test_numbers}
results
```

Computing Memory Usage and Execution Time

```
import math
import time
import resource

def is_prime(n):
    # Step 1: Handle small primes
    if n == 2 or n == 3:
```

```
        return True

    # Step 2: Check for divisibility by 2 and 3
    if n % 2 == 0 or n % 3 == 0:
        return False

    # Step 3: Determine if n is of the form 6k + 1 or 6k - 1
    if (n - 1) % 6 == 0:
        k_form = "6k + 1"
    elif (n + 1) % 6 == 0:
        k_form = "6k - 1"
    else:
        return False

    # Step 4: Compute n^3 % 18
    n_cubed_mod_18 = (n ** 3) % 18

    # Step 5: Check the congruence condition
    if k_form == "6k + 1" and n_cubed_mod_18 == 1:
        form_check = True
    elif k_form == "6k - 1" and n_cubed_mod_18 == 17: # since -1 mod
18 is 17
        form_check = True
    else:
        return False

    # Step 6: Trial division up to √n
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6

    return True

def measure_execution_time_memory(start_range, end_range):
    prime_counts = {}
    total_time = 0.0
    peak_memory = 0

    for n in range(start_range, end_range + 1):
        start_time = time.time()
        is_prime_result = is_prime(n)
        end_time = time.time()
```

```
        execution_time = end_time - start_time
        total_time += execution_time

        if is_prime_result:
            prime_counts[n] = True

        # Measure memory usage
        current_memory =
resource.getrusage(resource.RUSAGE_SELF).ru_maxrss



    return total_time, current_memory

# Define different ranges to measure
ranges = [(0,1000)
,(0,10000),(0,50000),(0,100000),(0,500000),(0,1000000),(0,5000000)]

# Measure execution time and memory for each range
for start, end in ranges:
    total_time, peak_memory = measure_execution_time_memory(start,
end)
    print(f"Range [{start}, {end}]:")
    print(f"  Total execution time: {total_time:.6f} seconds")
    print(f"  Peak memory usage: {peak_memory } bytes")
    print("-" * 50)
```

Squares of Primes Analysis

```
import math

# List of exponents
exponents = [
    24036583,
    12596495,
    30402457,
    32582657,
    37156667,
    42643801,
    43112609,
    57885161,
    74207281,
    77232917,
    82589933
]
```

```
# Calculate log2 of squared results to express in 2^() format
log2_results = [2 * exp for exp in exponents]  # Since log2((2^exp -
1)^2) is approximately 2*exp

# Calculate log2 of differences between consecutive results
log2_differences = [log2_results[i+1] - log2_results[i] for i in
range(len(log2_results)-1)]



# Print the results in 2^() format
for i, result in enumerate(log2_differences):
    print(f"Difference {i+1}: 2^{result}")
```