Prime Number Generation and Testing Algorithms

Hallee Pham

University of Missouri – Kansas City

Spring 2024 COMP-SCI 404

Table of Contents

[INTRODUCTION 4](#_Toc163921166)

[OBJECTIVES 4](#_Toc163921167)

[METHODOLOGY 4](#_Toc163921168)

[Research Phase 4](#_Toc163921169)

[Algorithm Implementation Phase 5](#_Toc163921170)

[Generating Random Test Cases Phase 5](#_Toc163921171)

[Testing Phase 6](#_Toc163921172)

[BACKGROUND 7](#_Toc163921173)

[The Importance of Prime Numbers 7](#_Toc163921174)

[Sieve of Eratosthenes 7](#_Toc163921175)

[*[Background]* 7](#_Toc163921176)

[*[Steps]* 7](#_Toc163921177)

[*[Implementation]* 8](#_Toc163921178)

[Sieve of Atkin 8](#_Toc163921179)

[*[Background]* 8](#_Toc163921180)

[*[Steps]* 9](#_Toc163921181)

[*[Implementation]* 9](#_Toc163921182)

[Fermat’s Primality Test 10](#_Toc163921183)

[*[Background]* 10](#_Toc163921184)

[*[Steps]* 10](#_Toc163921185)

[*[Implementation]* 11](#_Toc163921186)

[Rabin-Miller’s Primality Test 11](#_Toc163921187)

[*[Background]* 11](#_Toc163921188)

[*[Steps]* 11](#_Toc163921189)

[*[Implementation]* 12](#_Toc163921190)

[RESULTS 12](#_Toc163921191)

[DATA ANALYSIS 14](#_Toc163921192)

[Prime Number Generation Algorithms 14](#_Toc163921193)

[Prime Number Testing Algorithms 16](#_Toc163921194)

[CONCLUSIONS 17](#_Toc163921195)

[Limitations 17](#_Toc163921196)

[Future Considerations 18](#_Toc163921197)

[Summary 19](#_Toc163921198)

[References 21](#_Toc163921199)

[APPENDICES 22](#_Toc163921200)

[Appendix A – Sieve of Eratosthenes Code 22](#_Toc163921201)

[Appendix B – Sieve of Atkin Code 23](#_Toc163921202)

[Appendix C – Fermat’s Primality Test Code 24](#_Toc163921203)

[Appendix D – Rabin-Miller Primality Test Code 25](#_Toc163921204)

# INTRODUCTION

In this project, I discuss my research on the importance of prime numbers in technology, then explore/implement different algorithms that generate and test for prime numbers. The purpose of this documentation is to provide a comprehensive overview of the implemented algorithms, their time complexities characterized by their running time, and insights gained from the experimentation.

Keywords: Prime Numbers, Algorithms, Time Complexity

# OBJECTIVES

The main objectives of this project are:

* Implement prime generation algorithms (Sieve of Eratosthenes & Sieve of Atkin).
* Implement prime testing algorithms (Fermat’s Primality Test & Rabin-Miller Primality Test).
* Measure the execution time of the prime generation and testing algorithms using test cases.
* Analyze the time complexity of the algorithms.
* Visualize the data to demonstrate performance differences.

# METHODOLOGY

## Research Phase

The initial stage of this project involved research on the importance of prime numbers in the technology field through simple search engines. The prompts given to the search engines were along the lines of “What is the importance of prime numbers in computer science?”. The information found from this research stage can be found under the section “The Importance of Prime Numbers”.

After research about prime numbers, research was done about the different prime number generation and testing algorithms. The two prime number generation algorithms that were researched and later implemented were the Sieve of Eratosthenes and the Sieve of Atkin. The two prime number testing algorithms that were researched and later implemented were Fermat’s Primality Test and the Rabin-Miller Test. The main focus of this part of the research phase was to understand the steps of each algorithm and its general time/space efficiency. This involved looking at multiple sources and their different implementations/descriptions of the algorithms. Another goal of this research phase was to narrow down the best implementation of each algorithm to accurately analyze complexities in the testing phase.

## Algorithm Implementation Phase

The implementation of the previously stated algorithms was done by translating the theoretical concepts found from the research phase into executable code. The algorithms were coded in Python and run on Replit, an online compiler. Each algorithm was in a separate program to isolate the execution time of each one. With the information found in the research phase and thorough testing, the best implementation (according to time complexity) of each algorithm was implemented and ready for testing.

## Generating Random Test Cases Phase

After finding the best implementation of the algorithms, test cases were needed to be able to start the experimentation process. This involved using the random library in Python and the randint() method. The method takes in two integer parameters representing the lower and upper bound of the possible test cases. This functionality will be used in the testing process to test the efficiency of the algorithms according to an increasing number of digits possible. These test cases were essential for evaluating the performance and scalability of the implemented algorithms under different input conditions. Each program had the same implementation of the test case generating function to ensure that the execution time of each algorithm was not affected by the time it took to generate a test case.

## Testing Phase

Timers were integrated into the code to record their execution times. The timers used functions from the Time module available in Python. To isolate the running time of the algorithm from the execution of the test case function and any printing operations, the timer was started before the call of the prime generator/testing function and immediately stopped after the function call ended.

The information to estimate the time complexities of the prime number generation algorithms (Sieve of Eratosthenes and Sieve of Atkin) was collected by conducting three different tests, each with 10 random test cases. The three different tests differed by the number of digits in the range of the possible test cases to be randomly generated. The information to estimate the time complexities of the prime number testing algorithms (Fermat’s Primality Test and the Rabin-Miller Test) was tested in a similar process as the prime number generation algorithms.

After each algorithm is finished, the program prints out the total running time by subtracting the start time from the ending time recorded by the timer. This information was recorded (Table 1 in the RESULTS section).

# BACKGROUND

## The Importance of Prime Numbers

Prime numbers are fundamental to many different fields of technology, acting as the foundation for an array of algorithms, protocols, and cryptography systems. Their importance comes from their unique mathematical properties, which make them crucial in guaranteeing security, efficiency, and reliability in a wide range of technological applications.

One of the most useful properties of prime numbers is that it is very difficult to factor a large number into its prime factors. As of right now, there has not been an algorithm implemented/found to efficiently execute this task. This is why prime numbers play a pivotal role in creating secure cryptographic keys. To give an idea of the scale of computational effort: “factoring a 500-digit number into its primes could take as long as a planet’s formation” (Maier, 2021).

## Sieve of Eratosthenes

*[Background]* The Sieve of Eratosthenes is an algorithm that finds all the prime numbers less than or equal to a given number. This is an ancient method introduced by Greek mathematician Eratosthenes in the third century B.C. (Kaur, 2021). The general concept of this method is to create a list of numbers from 2 to the given number *n*. The program will then traverse from 2 to the square root of *n* and eliminate all numbers that are not prime. The reason why we only traverse until the square root of *n* is that there will not exist any factor of *n* that is greater than the square root of *n*.

### *[Steps]*

The following steps were sourced by GeeksForGeeks (GfG, 2023)

1. Generate numbers from 2 to *n*
2. Traverse from the smallest prime number (*num* = 2)
3. Mark all multiples of num that are less than or equal to *n* (This helps remove composite numbers to reduce the number of comparisons)
4. Update the current value of *num* to the next prime number
5. Repeat step three until *num* <= the square root of
6. Traverse the whole list and print out all unmarked numbers, which are all prime numbers that are less than or equal to *n*

*[Implementation]* The following code is a modified version of the Sieve of Eratosthenes from GeeksForGeeks. The code was modified to be compatible with a timer and to take in a random input. The rest of the code implementation can be found in Appendix A.

Figure 1 – Portion of Sieve of Eratosthenes Code

A screenshot of a computer program

Description automatically generated

## Sieve of Atkin

*[Background]* The Sieve of Atkin is an algorithm that prints all prime numbers small or equal to a given limit *n*. This method is similar to the Sieve of Eratosthenes, however, it does preliminary work before traversing through the list of numbers and marks off the multiples of squares of primes, instead of just the multiple of primes. The logic is much more complicated but is said to have a better theoretical time complexity.

### *[Steps]*

1. Create a “results list”, filled in with, 2, 3 and 5
2. Create a “sieve list” with all possible integers to the given limit *n*.
3. For each number *n* in the sieve list, with modulo-sixty remainder *r*:
   1. If *r* is 1, 13, 17, 29, 37, 41, 49, or 53, flip the entry for each possible solution to 4*x*2 + *y*2 = *n*.
   2. If *r* is 7, 19, 31, or 43, flip the entry for each possible solution to 3*x*2 + *y*2 = *n*
   3. If *r* is 11, 23, 47, or 59, flip the entry for each possible solution to 3*x*2 – *y*2 = *n* when *x* > *y*
   4. If *r* is something else, ignore it completely
4. Start with the lowest number in the sieve list.
5. Take the next number in the sieve list, still marked prime.
6. Include the number in the results list.
7. Square the number and mark all multiples of that square as non-prime. Note that the multiples that can be factored by 2, 3, or 5 need not be marked, as these will be ignored in the final enumeration of primes.
8. Repeat steps four through seven.

*[Implementation]* The following code is a section of a modified version of the Sieve of Atkin from GeeksForGeeks. The code was modified to be compatible with a timer and to take in a random input. The rest of the code implementation can be found in Appendix B.

Figure 2 – Portion of Sieve of Atkin Code

A screenshot of a computer program

Description automatically generated

## Fermat’s Primality Test

*[Background]* Fermat’s Primality Test is a method to identify if a given number is prime. This method is based on Fermat’s Little Theorem. Fermat’s Little Theorem says that if *n* is a prime number and if *a* is an integer that is relatively prime to *n*, then the following congruence relationship holds:

*an*-1 = 1(*mod n*)

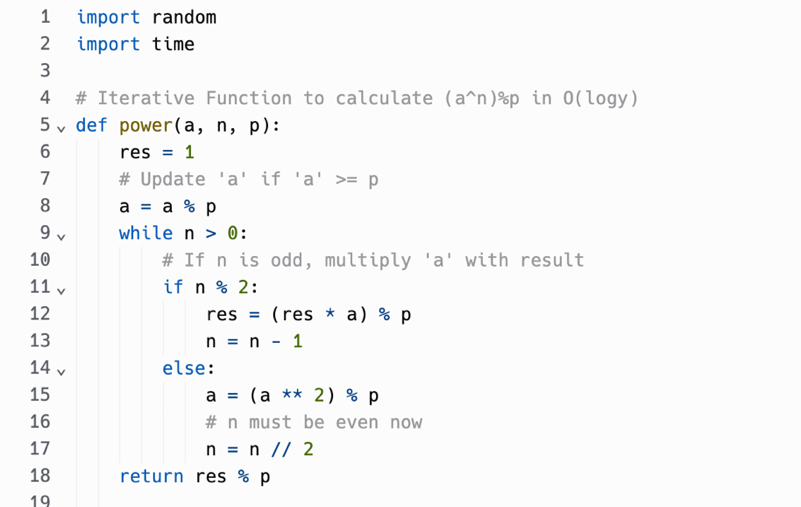
This property can be used to identify if a number is prime (Ma, 2013). This algorithm is different in the fact that the output result is “probably prime” instead of definitively prime. This is because the method has a factor *k*, which is a number initialized before the algorithm starts that represents the number of iterations the program will do to determine if a number is composite or prime. A higher value of *k* indicates a probability of correct results for composite inputs becoming higher (GfG, 2023).

*[Steps]* The following steps are a modified version from Dan Ma (Ma, 2013).

1. Choose a random integer a ∈ {2, 3, …, *n* − 1} where *n* is the number to be tested.
2. Compute the greatest common divisor of *a* and *n*. If *GCD*(*a*, *n*) > 1, stop because *n* is a composite number. If not, continue to the next step.
3. Compute *an*-1 (*mod n*)
   1. If *an*-1 != 1(*mod n*), stop because *n* is a composite number.
   2. If *an*-1 = 1(*mod n*), then *n* MAY be a prime number, continue.
4. Return to step one and repeat the process with a new *a*.

*[Implementation]* The following code is a section of a modified version of the Sieve of Atkin from GeeksForGeeks. The code was modified to be compatible with a timer and to take in a random input. The screenshot shown below is only a portion of the program. The rest of the code implementation can be found in Appendix C. The code uses *k* = 3.

Figure 3 – Portion of Fermat’s Primality Test Code



## Rabin-Miller’s Primality Test

*[Background]* Rabin-Miller’s Primality Test is an extension of Fermat’s Primality Test, but tests for primality at a much higher probability (Ginni, 2022). This method involves choosing random bases of *a* and checking multiple conditions for primality, unlike Fermat’s Test which only tests one condition.

### *[Steps]*

1. Select a random integer *a* such that 1< *a* < *n*−1, where n is the number being tested for primality.
2. Compute*x* = *ad* (*mod n*) where *d* is the largest power of 2 that divides evenly into *n* – 1.
   1. If *x* = 1(*mod n*) or *x* = -1(*mod n*), then *n* passes this iteration of the test
   2. If none of the above conditions are true, continue to the next steps
3. Repeat the following process *k* – 1 times (where *k* is the number of iterations chosen for the test):
   1. Square *x*, so *x* = *x2* (*mod n*)
   2. If *x* = -1 (*mod n*), *n* passes this iteration of the test.

*[Implementation]* The following code is a section of a modified version of the Sieve of Atkin from GeeksForGeeks. The code was modified to be compatible with a timer and to take in a random input. The screenshot shown below is only a portion of the program. The rest of the code implementation can be found in Appendix D. The code uses *k* = 3

Figure 4 – Portion of the Rabin-Miller Primality Test Code

A screenshot of a computer program

Description automatically generated

# RESULTS

Table 1 – Sieve of Eratosthenes

|  |  |  |  |
| --- | --- | --- | --- |
| Trial Number | 2 digits (μs) | 4 digits (μs) | 6 digits (μs) |
| 1 | 257.0 | 10277.7 | 5663437 |
| 2 | 468.3 | 23623.9 | 5373438 |
| 3 | 283.7 | 8996.5 | 638054 |
| 4 | 125.2 | 24919.0 | 2387793 |
| 5  6  7  8  9 | 216.5  223.4  7668.5  127.1  111.3 | 10999.0  7031.0  16903.6  10189.1  5679.4 | 1120843  5737152  4026678  3667132  5987158 |
| 10 | 418.7 | 22745.4 | 2877832 |
| Average | 989.97 | 14136.46 | 3747951.7 |

Table 2 – Sieve of Atkin

|  |  |  |  |
| --- | --- | --- | --- |
| Trial Number | 2 digits (μs) | 4 digits (μs) | 6 digits (μs) |
| 1 | 55.7 | 101353.9 | 10011654 |
| 2 | 284.7 | 27794.4 | 2525876 |
| 3 | 426.5 | 18753.5 | 7211703 |
| 4 | 193.6 | 3447.5 | 6171796 |
| 5  6  7  8  9 | 139.7  721.2  82.3  1075.0  162.4 | 24867.3  59017.7  7260.8  78552.5  64552.1 | 3237323  3359365  1361438  8035103  2972143 |
| 10 | 145.0 | 81522.7 | 341120 |
| Average | 328.61 | 46712.24 | 4522752.1 |

Table 3 – Fermat’s Primality Test

|  |  |  |  |
| --- | --- | --- | --- |
| Trial Number | 2 digits (μs) | 4 digits (μs) | 6 digits (μs) |
| 1 | 66.3 | 70.8 | 78.4 |
| 2 | 56.3 | 67.0 | 69.8 |
| 3 | 38.9 | 93.7 | 359.1 |
| 4 | 50.3 | 84.6 | 183.3 |
| 5  6  7  8  9 | 63.9  44.3  44.6  156.9  858.8 | 79.2  55.1  47.9  76.8  83.9 | 87.3  82.7  406.5  93.0  75.6 |
| 10 | 55.3 | 69.6 | 105.4 |
| Average | 143.56 | 72.86 | 154.11 |

Table 4 – Rabin-Miller’s Primality Test

|  |  |  |  |
| --- | --- | --- | --- |
| Trial Number | 2 digits (μs) | 4 digits (μs) | 6 digits (μs) |
| 1 | 97.5 | 58.7 | 78.9 |
| 2 | 64.4 | 48.9 | 77.5 |
| 3 | 68.2 | 78.4 | 196.2 |
| 4 | 62.0 | 80.8 | 50.3 |
| 5  6  7  8  9 | 55.8  78.2  46.3  59.4  78.8 | 104.0  43.6  68.0  45.1  105.4 | 90.6  76.3  79.9  191.5  2608.5 |
| 10 | 69.4 | 74.4 | 149.0 |
| Average | 68 | 70.73 | 359.87 |

# DATA ANALYSIS

## Prime Number Generation Algorithms

From the research done before the experiments, many sources have stated that the time complexity of the Sieve of Eratosthenes is worse than the time complexity of the Sieve of Atkin, as the Sieve of Atkin is a more modern version of the Sieve of Eratosthenes.

Doing a simple analysis of the implementation of the algorithms by counting the number of primitive operations, the Sieve of Eratosthenes has a theoretical time complexity of O(*n\*log* (*log n*)), where *n* is the input number. This is because the algorithm iterates through all numbers up to the input number and marks multiples of each prime number as a composite. The inner loop runs for each prime number found, which is approximately *log log n* times.

The Sieve of Atkin has a theoretical time complexity of O(*n*), where *n* is the limit. This is because we are using a sieve array of size *n*+1 to store boolean values for each number up to the limit. Comparing these two functions, Atkins has a better theoretical time complexity.

To analyze the data collected, the averages of the ten trials for each of the three tests will be compared to one another. This data can be seen as follows:

Table 5 – Comparison of the Average Times of Eratosthenes and Atkin

|  |  |  |
| --- | --- | --- |
| Number of Digits | Eratosthenes | Atkin |
| 2 | 989.97 | 328.61 |
| 4 | 14136.46 | 46712.24 |
| 6 | 3747951.7 | 4522752.1 |

It can be seen from the data that the Sieve of Atkin has a faster running time than the Sieve of Eratosthenes when calculating all prime numbers less than or equal to a two-digit number. However, it has a slower running time when computing the method for four- and six-digit numbers. This does not follow the theoretical complexities formerly predicted. As shown in Graph 1, Aktin should have a faster running time than Eratosthenes for *n* less than 1010 or for *n* with 11 digits if following the theoretical complexities. According to the experimental data collected, Atkin only runs faster with inputs around 2 digits. The factors that may have contributed to the contradictions between the theoretical and experimental conclusions will be discussed in the limitation section.

Graph 1 – Graphed Time Complexities of the Sieve of Eratosthenes and Atkin

A graph of a graph on a grid

Description automatically generated

\_\_\_\_\_\_\_

\_\_\_\_\_\_\_

Eratosthenes

Atkin

## Prime Number Testing Algorithms

From the research done before the collection of the experimental values, sources, and mathematicians state that the Rabin-Miller Primality Test is more efficient than Fermat’s Primality Test because the Rabin-Miller test is an improvement on Fermat’s Test.

By counting the primitive operations of the implementations of the two algorithms, an estimated time complexity was found. The theoretical time complexity of Fermat’s Test is O(*k* \* *log n*), where *k* is the set number of iterations of the test and *n* is the input. This is because the time complexity of the `power` function is O(*log n*) because it iterates through the bits of the exponent 'n' in a binary representation. The time complexity of the `isPrime` function is O(*k log n*) because it runs the `power` function *k* times with different random values of .

The theoretical time complexity of the Rabin-Miller Test is O(*k \* log3 n*), where *k* is the set number of iterations of the test and *n* is the input. The time complexity is determined by the number of iterations (*k*) and the complexity of the power function used in the algorithm. The power function has a time complexity of O(*log y*), where *y* is the exponent in the power calculation. Since the power function is called multiple times in the algorithm, the overall time complexity becomes O(*k \* log3 n*).

To analysis of the data collected for the prime testing algorithms was done with the same process as the analysis of the prime generation algorithms. The data can be seen as follows:

Table 6 – Comparison of Fermat’s and Rabin-Miller’s Primality Tests

|  |  |  |
| --- | --- | --- |
| Number of Digits | Fermat | Rabin-Miller |
| 2 | 143.56 | 68 |
| 4 | 72.86 | 70.73 |
| 6 | 154.11 | 359.87 |

It can be seen from the data from Table 6 that the Rabin-Miller Test had a faster runtime than Fermat’s Test with two and four-digit inputs and a slower runtime for inputs with 6-digits. With the theoretical values, the Rabin-Miller Test should have a faster runtime than Fermat’s Test when the input is 10 or lower. Though the experimental values do not follow the theoretical values, it does follow a similar pattern of the Rabin-Miller Test being better for smaller inputs, while the Fermat Test is faster for larger inputs. However, the complexity of these algorithms is heavily dependent on the number of iterations k. The higher the value of *k* is, the larger the range of values the Rabin-Miller Test would be theoretically faster than the Fermat Test.

A graph with a red line and blue line

Description automatically generatedGraph 2 - Graphed Time Complexities of Fermat’s and the Rabin-Miller Primality Tests

Rabin-Miller

Fermat’s

\_\_\_\_\_\_\_

\_\_\_\_\_\_\_

# CONCLUSIONS

## Limitations

From the data analysis section, it can be seen that the experimental values collected only follow the theoretical values gathered from prior research very loosely. Though this does not make the conclusions derived from the test data invalid, there are many factors and limitations that could have contributed to the large variance between the actual and expected results.

Though the were tried to keep as stable as possible, there were many factors related to the computing environment that most likely caused erroneous results. For example, the programs were written and executed on an online compiler, meaning that changes in internet connection could have affected the run times of the algorithms. Another limitation of this experiment was the lack of computing power the device and compiler could handle. For example, the original plan was to also test 8 and 16-digit numbers, but the program would stall heavily and sometimes result in crashes. This resulted in a very small possible test range that didn’t capture the full potential of the algorithms.

Another factor that could have contributed to erroneous results was the implementation of the algorithms. The implementations were tested as much as possible to find the best ones, however, it is not impossible that there were more optimal solutions. For example, when initially testing the prime generation algorithms, it was found that printing the numbers on separate lines generally resulted in a longer run time than printing the numbers on the same line separated by a space. The process of printing the numbers was edited to be as similar as possible. Many similar problems could have been overlooked, resulting in inaccurate results.

## Future Considerations

Taking into consideration the limitations and possible errors of the experiment, there are many ways this experiment could be modified to produce more accurate results. One thing that could be done is to find a more stable compiler to test the programs in, especially one that does not rely on an internet connection. This would increase the likelihood of accurate results.

Another modification to this experiment that can be done is the collection of more data. Doing more than 10 trials for each test would greatly increase the accuracy of the averages of each test. Testing with larger values of *k* in the prime testing algorithms would result in a more accurate relationship between the size of the input and the running time. Also extending the range of the tests to numbers with more than 6 digits would better show how the runtimes are affected by larger numbers. Data collected from testing very large numbers would show how pivotal the properties of prime numbers are to cryptography, because many security methods use very large prime numbers, as they require a lot of computational power and time.

## Summary

In this project, two prime number-generating algorithms were researched and tested to compare their time complexities. The two algorithms are called the Sieve of Eratosthenes and the Sieve of Atkin. The estimated time complexity of the Sieve of Eratosthenes was O(*n \* log (log n*)), which is worse than the estimated time complexity O(*n*) of the Sieve of Atkin. To find the actual running time of the algorithms, data was collected by testing each of the algorithms for different-sized inputs. This resulted in values that only vaguely followed the functions of the estimated time complexities. The Sieve of Atkin was found to run faster for inputs with a smaller amount of digits, but slower than the run time of the Sieve of Eratosthenes for inputs with more digits.

Two prime testing algorithms were also researched and tested. These algorithms are called Fermat’s Primality Test and the Rabin-Miller Primality Test. The estimated time complexity of Fermat’s Test was O(*k \* log n*), which should be slower than the estimated time complexity of O(*k \* log 3n*) of the Rabin-Miller Test. The experimental results showed this relationship very loosely, most likely due to the small value of *k*, the number of iterations.

In conclusion, this experiment shows that some implementations of finding and identifying prime numbers are better than others. Also, this project provides a good foundation for future experiments and research involving prime numbers and prime number generation/testing algorithms.

The source code files for this project can be found at <https://github.com/halleepham/CS404-MiniProject.git>

References

GfG. (2022, November 14). *Primality test: Set 3 (Miller–Rabin)*. GeeksforGeeks. https://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/

GfG. (2023a, March 31). *Sieve of atkin*. GeeksforGeeks. https://www.geeksforgeeks.org/sieve-of-atkin/

GfG. (2023b, June 1). *Fermat method of Primality Test*. GeeksforGeeks. https://www.geeksforgeeks.org/fermat-method-of-primality-test/

GfG. (2024, March 6). *Sieve of eratosthenes*. GeeksforGeeks. https://www.geeksforgeeks.org/sieve-of-eratosthenes/

Kaur, K. (2021). *Sieve of eratosthenes algorithm*. Top Website Designers, Developers, Freelancers for Your Next Project. https://www.topcoder.com/thrive/articles/sieve-of-eratosthenes-algorithm

Ma, D. (2015, March 29). *Using Fermat’s little theorem to test primality*. Exploring Number Theory. https://exploringnumbertheory.wordpress.com/2013/08/10/using-fermats-little-theorem-to-test-primality/

Maier, A. (2021, August 16). *Prime numbers and their importance to modern life*. CodeCoda.com. https://codecoda.com/en/blog/entry/prime-numbers-and-the-importance-to-modern-life

# APPENDICES

# Appendix A – Sieve of Eratosthenes Code

A screenshot of a computer program

Description automatically generated

## Appendix B – Sieve of Atkin Code

A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

## Appendix C – Fermat’s Primality Test Code

A screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

## A screen shot of a computer code Description automatically generatedA screenshot of a computer code Description automatically generatedAppendix D – Rabin-Miller Primality Test Code

A screenshot of a computer program

Description automatically generated