Exploring the Efficiency and Accuracy of Primality Testing Techniques

Michael Maloney
University of North Carolina
Wilmington
Wilmington, North Carolina
mdm1975@uncw.edu

Keywords – Primality Testing, Trial Division, Fermat's Primality Test, Sieve of Eratosthenes, Miller-Rabin Test, AKS Test, Probabilistic Primality Testing, Deterministic Primality Testing

ABSTRACT

Primality testing is a vital aspect of computer science that aims to efficiently and accurately determine whether a given number is prime or composite. This project investigates various primality testing techniques, such as trial division, Fermat's primality test, the Sieve of Eratosthenes, and the Miller-Rabin test, with a focus on their applications in public key cryptography and the RSA cryptosystem. The analysis evaluates the efficiency and accuracy of these methods, specifically in the context of large input sizes.

Seminal works in primality testing are also examined, concentrating on the distinctions between deterministic and probabilistic tests in RSA encryption and other cryptographic systems. Experimental results are presented and discussed, revealing the strengths and weaknesses of each algorithm as well as their relative performance concerning Big-O notation.

I. INTRODUCTION

Prime numbers are positive integers with two distinct positive divisors: one and themselves. In contrast, composite numbers possess more factors as the input size increases. For instance, the number 8 is composite as it has four factors: one, two, four, and itself, while the number 7 is prime with two factors: one and seven.

The complexity of primality testing grows as the input size increases, primarily due to the expanded search space for potential factors. Various primality testing methods necessitate examining integers up to a specific limit, such as the square root of the input number, to identify factors. Notable techniques addressing this challenge include trial division, Fermat's primality test, the Sieve of Eratosthenes, and the Miller-Rabin test.

This computational challenge has far-reaching implications in areas such as public key cryptography, where secure encryption and decryption rely on the properties of prime numbers. For example, in the widely used RSA cryptosystem, participants must "select at random two large prime numbers p and q such that $p \neq q$" to create their public and secret keys [2]. The process of primality testing becomes increasingly expensive as the size of the input number becomes large—this is due to the expanding search space for possible factors. Finding large "random" primes is crucial for many cryptography applications. Cormen [2] asserts that "fortunately, large primes are not too rare, so that it is feasible to test random integers of the appropriate size until we find a prime." Some well-known algorithms will be implemented to address this issue. These algorithms include trial division, Fermat's primality test, the Sieve of Eratosthenes, and the Rabin Miller test.

The subsequent sections will offer an in-depth examination of the primality testing algorithms, focusing on their implementations and experimental outcomes regarding efficiency and accuracy. Section II delves into primality testing algorithm implementations and related works, highlighting the problems they were designed to solve and the efficacy of the implemented solution. Section III explores the methodologies used in primality testing, encompassing probabilistic approaches like Miller-Rabin to deterministic techniques such as AKS (Agrawal–Kayal–Saxena). This section details the algorithms' implementations and outlines the experimental procedures utilized. Lastly, section IV presents the experimental results and discusses the findings, emphasizing the strengths and weaknesses of each algorithm.

II. RELATED WORK

Primality testing has long been an area of significant interest within the computer science and mathematics community, given its essential role in algorithmic approaches to determining whether a given number is prime or composite while considering both efficiency and accuracy. Effectively handling the increasing computational complexity that emerges with larger input sizes to produce arbitrarily large integers for cryptographic purposes has prompted researchers to develop a myriad of techniques to tackle these computational challenges.

In this section, seminal works and advances in primality testing algorithms are investigated and existing works on primality testing techniques are explored such as trial division, Fermat's primality test, the Sieve of Eratosthenes, and the Miller-Rabin test, emphasizing their applications in public key cryptography—specifically, the RSA cryptosystem, assessing the use of deterministic and probabilistic primality tests in RSA encryption and other cryptographic systems.

In the subsequent subsections, a more comprehensive analysis of the related works will be offered, delving into the intri-

cacies of each primality testing technique and their relevance to the problems they were designed to solve.

A. Basic Primality Testing Techniques: Trial Division and Sieve of Eratosthenes

Trial division is a simple method for testing primality. As described by Crandall, it involves "sequentially trying test divisors into a number n so as to partially or completely factor n" [1]. By dividing the given input by each integer up to its square root, prime numbers can be determined if "we reach a trial divisor that is greater than the square root of the unfactored portion, we may stop, since the unfactored portion is prime" [1]. This method is efficient for small numbers but becomes increasingly inefficient as the input size increases due to the exponentially increasing number of possible factors. To improve the efficiency of trial division, an implementation can leverage the fact that after 2, the primes are odd. So two and the odd numbers may be used as trial divisors [1]. However, this still needs to address the inefficiency of large input sizes fully.

In contrast, the Sieve of Eratosthenes is a widely known and simple algorithm for finding prime numbers up to a given integer. Based on sieving—a mathematical term for the process of filtering out unwanted elements from a set—it is considered "a highly efficient means of determining primality and factoring when one is interested in the results for every number in a large, regularly spaced set of integers" [1]. The algorithm iteratively marks the multiples of prime numbers as composites, leaving only prime numbers unmarked at the end of the process.

The algorithm starts with an array of ones, representing the numbers from 2 to n. The first unmarked number is considered prime, and all its multiples in the array are marked as composites. This process is repeated for the following unmarked number until no more unmarked numbers whose square is less than or equal to n [1].

Despite its simplicity, the Sieve of Eratosthenes has some drawbacks. One of the significant limitations is the amount of space it consumes, as it requires an array large enough to hold all numbers up to n [1]. Additionally, the number of steps in the sieve is proportional to the sum of n divided by each prime less than or equal to n [1]. As a result, the algorithm becomes computationally expensive for large input sizes, making it less efficient for generating prime numbers in those cases.

B. Probabilistic Primality Testing: Fermat's and Miller-Rabin Tests

While trial division and the Sieve of Eratosthenes are deterministic algorithms that guarantee exact results for any given input, their efficiency significantly decreases when applied to large numbers [7]. Deterministic algorithms do not rely on chance or probability to determine whether a number is prime or composite. However, their performance limitations for large input sizes call for alternative approaches, such as probabilistic algorithms.

In contrast, probabilistic algorithms offer an alternative approach where the result obtained depends on chance, meaning that the outcome may not always be accurate. Nevertheless, probabilistic algorithms often have a high probability of being true [7]. Examples of probabilistic primality tests include the Fermat Primality Test and the Miller-Rabin Test.

Based on Fermat's Little Theorem, Fermat's primality test is a probabilistic test that can identify pseudoprimes. Agrawal [4] surveyed three primality testing algorithms based on Fermat's Little Theorem, including the Solovay-Strassen Test, the Miller-Rabin Test, and the AKS Test. The Miller-Rabin Test, a modification of the Fermat Primality Test, is one of the most efficient probabilistic primality tests. It can be repeated multiple times with random initializations to reduce the probability of a composite number not being recognized as such [3].

Large prime numbers are essential in various applications, particularly in modern cryptography. Using large numbers' prime factors, most cryptographic systems operate securely [7], as they are challenging to factorize—making them more resilient to attacks. Besides their use in cryptography, prime numbers are employed in random number generators, error-correcting codes, and hash functions, either directly or indirectly [7].

III. Methodology

In this section, each primality testing algorithm's implementation and its time complexity are explored. A concise pseudocode description is provided to facilitate a better understanding of the algorithm.

A. Trial Division

Trial division is a deterministic algorithm that tests whether a given number is prime or composite by dividing the number by integers up to its square root.

This algorithm initiates by dividing the number by integers, starting with two and continuing until the square root of the number is reached. If any of these integers divide the number with a remainder of 0, the number is deemed composite and not prime.

For instance, to confirm whether 71 is prime, the algorithm divides 71 by integers up to $\sqrt{8} \approxeq (8.43)$. The algorithm divides 71 by 2, 3, 4, 5, 6, 7, and 8, as illustrated below:

- $71 \div 2 = 35 \Rightarrow$ *remainder 1*
- $71 \div 3 = 23 \Rightarrow$ *remainder 2*
- $71 \div 4 = 17 \Rightarrow$ *remainder 3*
- $71 \div 5 = 14 \Rightarrow$ *remainder 1*
- $71 \div 6 = 11 \Rightarrow$ *remainder 5*
- $71 \div 7 = 10 \Rightarrow$ *remainder 1*
- $71 \div 8 = 8 \Rightarrow$ *remainder 7*

In the example above, it is only necessary to check integers up to 8 since 9 is greater than the square root of 71. As none of the integers from 2 to 8 divide 71 evenly, 71 is considered prime. The algorithm's efficiency can be improved by dividing only by prime numbers, excluding composite numbers. If 2 cannot divide a number evenly, then any composite number that includes 2 as a factor also cannot divide the number evenly. Consequently, it is only required to test the number against prime divisors.

The Big-O time complexity of the trial division for determining whether a given number is prime is $O(\sqrt{n})$, where n is the number being tested for primality. This is attributed to the algorithm only needing to test the divisibility of n by integers up to the square root of n. The algorithm's maximum number of iterations is $\sqrt{n}$. Although the trial division algorithm is a relatively simple and straightforward method for determining primality, it can become computationally expensive for large values of n, as the number of iterations required increases with $\sqrt{n}$.

The trial division algorithm is implemented in the *TrialDivision* function, as shown below:

```
TrialDivision(n) ⇒
  if n ≡ 2 or n ≡ 3 then
    return true

  if n ⩽ 1 or mod(n, 2) ≡ 0 then
    return false

  for i from 2 to √n do
    if mod(n, i) ≡ 0 then
      return false

  return true
```

Figure 1: Trial Division Algorithm

This figure tests the input number n for primality by iterating through all integers from 2 to the square root of n. If any of these integers divide n evenly, then n is composite and not prime. If none of the integers divide n evenly, then n is prime.

B. Sieve of Eratosthenes

The Sieve of Eratosthenes is a simple and efficient deterministic algorithm for finding all prime numbers up to a specified integer. The algorithm begins by creating a list of integers from 2 to n, where n is the upper bound of the range of integers to be tested for primality [1]. The algorithm then iterates through the list of integers, starting with 2, and removes all multiples of the current integer from the list. The algorithm continues to iterate through the list of integers, removing all multiples of each integer until the square root of n is reached or surpassed. The remaining integers in the list are all prime numbers.

The Sieve of Eratosthenes algorithm is highly efficient for finding all prime numbers within a specified range of numbers. The sieve algorithm performs significantly faster than the trial division algorithm, which checks each number individually for primality. This makes the Sieve of Eratosthenes an ideal choice when a rapid identification of all prime numbers within a particular range is necessary.

In contrast to the trial division algorithm's time complexity of $O(\sqrt{n})$ [1], the *Sieve of Eratosthenes* algorithm presents a time complexity of $O(n \log \log n)$, where n represents the upper limit of the range of numbers to be checked for primes [1].

```
PrimeSieve(n) ⇒
  var sieve = [true, ..., true] of size n+1
  var sieve[0] = false
  var sieve[1] = false

  for i from 2 to √n do
    for j from (i×i) to (n+1) do
      sieve[j] = false

  var primes = list where primes[i] is true

  return primes
```

Figure 2: Sieve of Eratosthenes Algorithm

In this figure, the input number n is used to create a list of boolean values, where each value represents the primality of the corresponding index. All values are initially set to true, except for the first two values, which are set to false. The algorithm then iterates through the list of integers, starting with 2, and removes all multiples of the current integer from the list. The algorithm continues to iterate through the list of integers, removing all multiples of each integer until the square root of n is reached or surpassed. The remaining integers in the list are all prime numbers.

C. Fermat's Test

Fermat's test is a probabilistic algorithm for determining whether a given number is prime based on Fermat's Little Theorem, which Euler proved in 1736 [10]. The theorem states that for any natural number a and prime p, $a^{p-1} \equiv 1 \pmod{p}$ [10]. Fermat's test randomly selects an integer a between 2 and $p-1$, where p is the number tested for primality. The algorithm consists of two steps [10]:

1. Randomly select a number a for which $1 < a < p$.
2. Test if the congruence $a^{p-1} \equiv 1 \pmod{p}$ is satisfied.

If the congruence $a^{p-1} \equiv 1 \pmod{p}$ is satisfied, then p may or may not be a prime number [10]. If the congruence is not satisfied, the number p is composite, and a is called Fermat's witness for the compositeness of p [10]. The algorithm repeats this process k times, where k is the number of iterations enforced at runtime. If the algorithm returns true for all iterations, p is considered prime. If the algorithm returns false for any iteration, then p is composite.

The time complexity of Fermat's test is $O(k \log n)$, where k is the number of iterations and n is the number being tested for primality [10]. The algorithm is probabilistic, meaning that it is possible for the algorithm to return a false positive, incorrectly identifying a composite number as prime. These *Fermat liars*—known as *pseudoprimes*—are rare but do exist. However, the probability of observing a Fermat liar decreases exponentially as the number of iterations k increases. Increasing k reduces the probability of encountering a Fermat liar to an acceptably low level.

The algorithm is implemented in the *Fermat* function, as shown below:

```
Fermat(p, k = 5) ⇒
  if p ≡ 2 or p ≡ 3 then
    return true
```

```
if p ⩽ 1 or mod(p, 2) ≡ 0 then
  return false

repeat k times
  var a = random integer from 2 to p-1
  if mod(a^(p-1), p) ≠ 1 then
    return false

return true
```

Figure 3: Fermat's Test Algorithm

In this figure, the algorithm evaluates whether p is equivalent to 2 or 3, in which case it returns true. Subsequently, it ascertains if p is less than or equal to 1 or an even integer, returning false if either criterion is met. The algorithm proceeds to execute k iterations, with k denoting the defined number of iterations. During each iteration, the algorithm randomly selects an integer a within 2 to $p-1$. The algorithm returns false if the congruence $a^{p-1} \equiv 1 \pmod{p}$ is unsatisfied. In the event that the congruence is satisfied across all iterations, the algorithm assumes that p is prime and returns true.

D. Miller-Rabin Test

The Miller-Rabin test is a well-known probabilistic algorithm that efficiently determines the likelihood of a given number being prime without providing absolute certainty. It has become the de facto probabilistic primality test, with more than 75% of numbers from 2 to $n-1$ serving as witnesses in the Miller–Rabin test for an odd composite $n > 1$ [12]. The time complexity of the Rabin-Miller primality test is $O(k\ log^{3(n)})$, where n represents the candidate prime number and k is the count of test iterations.

Expanding on the Fermat test, which leverages congruence modulo prime numbers and Fermat's Little Theorem [12], the Miller-Rabin test employs a similar concept but incorporates a system of congruences [12]. In this context, a "witness" denotes a value that signifies the tested number is composite. An odd prime number does not possess any Miller–Rabin witnesses; therefore, if n has a Miller–Rabin witness, it must be composite [12].

The Miller-Rabin test's error rate depends on the number of testing rounds k. As k increases, the likelihood of accurately identifying a prime number also rises [12]. The *MillerRabin* function, detailed below, implements the algorithm:

```
MillerRabin(n, k = 5) ⇒
  if p ≡ 2 or n ≡ 3 then
    return true

  if n ⩽ 1 or mod(n, 2) ≡ 0 then
    return false

  var r = n - 1
  var s = 0

  while mod(r, 2) ≡ 0:
    r = r / 2
    s = s + 1

  repeat k times
    var a = random integer from 2 to n-1
    var x = mod(a^(r), n)

    if x ≠ 1 then
      var i = 0
      while x ≠ n - 1 do
        if i ≡ s - 1 then
          return false
        else:
          i = i + 1
          x = mod(x^(2), n)

  return true
```

Figure 4: Miller-Rabin Test Algorithm

In Figure 4, the algorithm first checks whether n is equivalent to 2 or 3, and returns true if so. Next, it verifies if n is less than or equal to 1 or an even integer, returning false if either condition is satisfied. Subsequently, it computes the values of r and s by decrementing n by 1 and halving r until it is no longer even. The algorithm performs k iterations, where k is the specified number of iterations. Within each iteration, it randomly selects an integer a from the range 2 to $n-1$. The algorithm returns false if the congruence $a^r \equiv 1 \pmod{n}$ is unsatisfied. If the congruence holds true across all iterations, the algorithm assumes that n is prime and returns true.

E. Agrawal–Kayal–Saxena (AKS)

The AKS algorithm is a deterministic primality algorithm that determines whether a given number is prime in polynomial time. The AKS primality test is based on the primality characterization theorem, which states that an integer n greater than 2 is prime if and only if the polynomial congruence relation $(X+a)^n = (X^n + a)\ (mod\ n)$, holds for some coprime to n [8].

The algorithm first checks the input number to determine if it equals or exceeds 1. It then determines if the input number is a perfect power, in which case it would not be prime. The algorithm then determines the bounds for checking the primality condition by finding the smallest value of r, such that n modulo r is greater than the square root of the most significant prime factor of r. With the boundary for the primality check set, the algorithm then determines if the input number has the greatest common denominator. The last step is determining if the polynomial convergence equation holds for all integer values between 1 and the square root of $r+1$.

If the equation holds, the input value, n, is a prime number [8]. The time complexity of the AKS algorithm is $O(log^6 n)$ [8], where n is the number being tested for primality. While the AKS algorithm is noted for its theoretical importance for being the first deterministic primality test, AKS is by no means the most efficient algorithm to determine primality [8]. While the algorithm can efficiently test more significant prime numbers, other probabilistic algorithms, such as the Miller-Rabin test mentioned earlier.

The *AKS* function, detailed below, implements the algorithm:

```
IsPower(n) ⇒
```

```
  if n ⩽ 1 then
    return false

  for b = 2 to log(n) do
    a =n^(1/b)

    if a^b ≡ n then
        return true

  return false

GCD(a, b) ⇒
  while b ≠ 0 do
    t = b
    b = mod(a, b)
    a = t

  return a

FindR(n) ⇒
  r = 2
  while r ⩽ log(n)^2 do
    if GCD(n, r) ≠ 1 then
      return r
    else:
      r = r + 1

  return r

AKS(n) ⇒
  if n ⩽ 1 then
    return false

  if IsPower(n) then
    return false

  var r = FindR(n)

  for a from 2 to r+1 do
    if GCD(a, n) > 1 then
      return false

  for a from 1 to √r+1 do
    var x = mod(a^(n), n)
    var y = mod((x − a)^(n), n)

    if x ≠ y then
      return false

  return true
```

Figure 5: AKS Algorithm

IV. Experimental Results

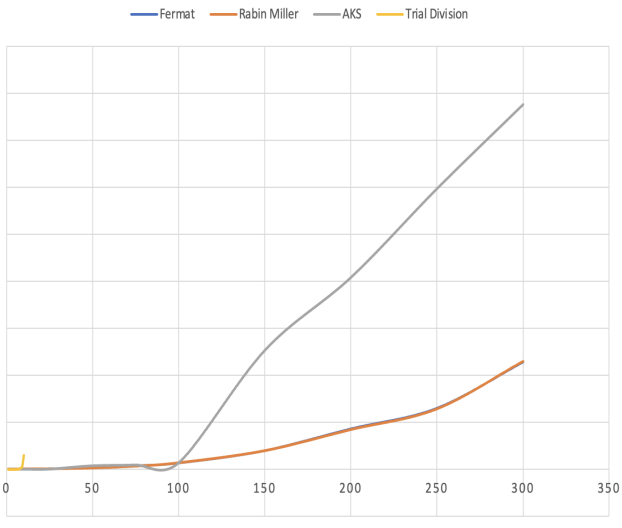
This section presents the experimental results obtained by evaluating four primality testing algorithms discussed in the previous section. Although the Sieve of Eratosthenes was reviewed earlier, it was excluded from the tests. This exclusion stems from the fact that the Sieve of Eratosthenes is primarily designed to find all prime numbers up to a specified integer rather than determining the primality of individual large numbers.

The performance of each algorithm was assessed by measuring the time taken to determine the primality of numbers with varying sizes. The number of digits for each test ranged from 1 to 300, equivalent to a maximum of 996 bits.

A. Findings

The following table displays the time taken, in seconds, to determine the primality of numbers with different digit counts. The experimental results were obtained by implementing each algorithm programmatically.

Digits	Fermat	Rabin Miller	AKS	Trial Division
1	0.000008	0.000008	0.000008	0.000001
2	0.000006	0.000004	0.000005	0.000001
3	0.000005	0.000005	0.000006	0.000001
4	0.000004	0.000004	0.000008	0.000002
5	0.000005	0.000005	0.000009	0.000004
6	0.000006	0.000006	0.000007	0.000019
7	0.000007	0.000006	0.000011	0.000065
8	0.000007	0.000007	0.000009	0.000261
9	0.000007	0.000007	0.000008	0.000525
10	0.000025	0.000022	0.000015	0.002931
25	0.00005	0.000048	0.000032	#N/A
50	0.000222	0.000218	0.00074	#N/A
75	0.000658	0.000614	0.00091	#N/A
100	0.001294	0.001327	0.00138	#N/A
150	0.003903	0.003902	0.0253	#N/A
200	0.008531	0.008401	0.0408	#N/A
250	0.012914	0.01281	0.0597	#N/A
300	0.022804	0.022918	0.0776	#N/A



The results reveal that trial division is the fastest method for smaller numbers (up to 5 digits). However, its performance

quickly deteriorates as the input size increases, becoming impractical for numbers beyond 10 digits. Due to its inefficiency for larger inputs, trial division was not tested for numbers more significant than 10 digits.

Fermat's primality test and the Miller-Rabin test exhibit similar performance trends, with the latter having a slight edge in efficiency for larger input sizes. Both algorithms demonstrate better scalability than trial division and maintain reasonable execution times even for numbers with several hundred digits. The AKS algorithm, exhibits slower performance compared to the probabilistic Fermat and Miller-Rabin tests.

In summary, our experimental results indicate that the Miller-Rabin test offers the best balance between efficiency and accuracy for primality testing of large numbers, particularly in cryptographic applications. While the Fermat test also performs well, the Miller-Rabin test provides a higher degree of confidence in the primality of the tested numbers. As a deterministic test, the AKS algorithm offers a guarantee of correctness but at the cost of reduced efficiency compared to probabilistic tests. Trial division remains a viable option for smaller numbers due to its simplicity and speed.

REFERENCES

[1] Crandall, R. and Pomerance, C. *Prime Numbers: A Computational Perspective*. Springer, 2005.

[2] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. *Introduction to Algorithms*. The MIT Press, 2022.

[3] Schoof, Rene. *Four Primality Testing Algorithms*. 2008.

[4] Agrawal, M. *Primality Tests Based on Fermat's Little Theorem*. In *Distributed Computing and Networking: 8th International Conference, ICDCN 2006, Guwahati, India, December 27-30, 2006. Proceedings*, vol. 8, pp. 288-293, 2006.

[5] Abdullah, D., Rahim, R., Apdilah, D., Efendi, S., Tulus, T., and Suwilo, S. *Prime Numbers Comparison Using Sieve of Eratosthenes and Sieve of Sundaram Algorithm*. *Journal of Physics: Conference Series*, vol. 978, pp. 012123, 2018.

[6] Narayanan, S. *Improving the Speed and Accuracy of the Miller-Rabin Primality Test*. MIT PRIMES-USA, 2014.

[7] Rajput, J. and Bajpai, A. *Study on deterministic and probabilistic computation of primality Test.. Proceedings Of International Conference On Sustainable Computing In Science, Technology And Management (SUSCOM) Amity University Rajasthan, Jaipur-India*. 2019.

[8] Agrawal, M., Kayal, N. & Saxena, N. PRIMES is in P. *Annals Of Mathematics*. pp. 781-793 (2004)

[9] Zhiyong Zheng *Modern Cryptography Volume 1. A classical Introduction to Informational and Mathematical Principle*, vol. 1 pp. 58-61, 2022

[10] Ďuriš, V. *Solving some specific tasks by Euler's and Fermat's Little theorem*. 2019. vol. 37, pp. 39-48. Ratio Mathematica.

[11] Hurd, J. Verification of the Miller–Rabin probabilistic primality test. *The Journal Of Logic And Algebraic Programming*. **56**, 3-21 (2003)

[12] Conrad, K. The Miller–Rabin Test. *Encyclopedia Of Cryptography And Security*. (2011)