CSE470 - 1st Homework Report

by Şiyar Tarık Özcaner

200104004021

08/12/2023

Introduction

Primality testing in number theory and cryptography is a fundamental problem. It is important to know whether a given number is prime or composite for a variety of applications, including ensuring that cryptographic systems are secure. We will look into and compare seven primality testing algorithms in this research, emphasising their advantages, disadvantages, and computational challenges.

1. Trial Division

Trial division is the most straightforward primality testing algorithm. It checks for divisibility by testing all possible divisors up to the square root of the number.

Pros:

Simple and easy to understand.

Guarantees accuracy.

Cons:

Inefficient for large numbers.

Time complexity: O(sqrt(n)).

2. Miller-Rabin

Miller-Rabin is a probabilistic algorithm that uses randomization to determine primality. It performs multiple rounds of tests with different random bases.

Pros:

Fast and efficient.

Probabilistic correctness.

Cons:

Possibility of false positives (rare).

Time complexity: O(k * log^3(n)), where k is the number of rounds.

3. Solovay-Strassen

Solovay-Strassen is another probabilistic algorithm based on Euler's criterion. It combines modular exponentiation and Jacobi symbol calculations.

Pros:

Relatively fast.

Probabilistic correctness.

Cons:

May produce false positives.

Time complexity: O(k * log^3(n)), where k is the number of rounds.

4. AKS (Agrawal-Kayal-Saxena)

AKS is a deterministic algorithm that determines primality using polynomial-time computations. It is based on cyclotomic polynomials and modular arithmetic.

Pros:

Deterministic correctness.

Polynomial time complexity.

Cons:

Slower in practice compared to some probabilistic algorithms.

Time complexity: O((log n)^6).

5. Sieve of Eratosthenes

The Sieve of Eratosthenes is an ancient algorithm for finding all primes up to a given limit. It is not a primality test but is included for comparison.

Pros:

Efficient for finding all primes in a range.

Time complexity: O(n log log n).

Cons:

Not suitable for testing individual numbers.

6. Sieve of Atkin

The Sieve of Atkin is a modern variation of the classical sieve algorithm. It efficiently finds primes up to a given limit.

Pros:

Faster than the Sieve of Eratosthenes for large ranges.

Time complexity: O(n / log log n).

Cons:

Not designed for primality testing of individual numbers.

7. Fermat's Little Theorem

Fermat's Little Theorem is a probabilistic algorithm based on modular exponentiation. It tests whether $a^{(n-1)} \equiv 1 \pmod{n}$ for a randomly chosen 'a'.

Pros:

Simple and easy to implement.

Probabilistic correctness.

Cons:

May produce false positives.

Time complexity: O(k * log^3(n)), where k is the number of rounds.

Comparison and Analysis

Deterministic vs. Probabilistic:

Deterministic algorithms, like AKS, provide guaranteed correctness but may be slower.

Probabilistic algorithms, like Miller-Rabin and Solovay-Strassen, offer speed with a small probability of error.

Time Complexity:

Deterministic algorithms often have higher time complexity, making them less suitable for large inputs.

Probabilistic algorithms can be more efficient for practical purposes.

False Positives:

Probabilistic algorithms may yield false positives, but the probability can be reduced by increasing the number of rounds.

Use Cases:

Trial division and AKS are suitable for small inputs.

Miller-Rabin and Solovay-Strassen are practical for large inputs in cryptography.

Sieve algorithms are efficient for finding primes in ranges but not for individual tests.

Implementations of the Algorithms

1.Miller-Rabin

```python
def miller_rabin(n, k=1000):
    if n == 1:
        return False
    if n == 2 or n == 3:
        return True
    if n % 2 == 0:
        return False

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

    for _ in range(k):
        a = random.randint(2, n - 2)
        x = pow(a, s, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True
```

Fundamentals:

The Miller-Rabin primality test is a probabilistic algorithm used to determine if a given number n is likely to be a prime number.

It is based on the properties of modular exponentiation.

Algorithm Steps:

It first handles special cases for small values of n (1, 2, and 3).

For odd numbers (n % 2 == 1), it expresses n - 1 as 2^r * s by repeatedly dividing by 2 until it becomes odd. This helps in expressing n - 1 as an even number (2^r) multiplied by an odd number (s).

It then performs the Miller-Rabin test k times, where k is a parameter that determines the accuracy of the test.

In each iteration of the test, a random integer a is chosen between 2 and n - 2. Modular exponentiation is used to compute x = a^s mod n.

If x is 1 or n - 1, the test continues to the next iteration. Otherwise, a loop is executed to square x r - 1 times. If the final result is not equal to n - 1, then n is declared composite.

Time Complexity:

The time complexity of the Miller-Rabin primality test is probabilistic.

The algorithm performs the test k times, and each iteration involves modular exponentiation, which can be computed in O(log n) time.

Therefore, the overall time complexity is O(k * log n).

Accuracy:

The accuracy of the Miller-Rabin test depends on the parameter k. Increasing k increases the probability that the result is correct, but also increases the running time.

Probabilistic Nature:

Miller-Rabin is a probabilistic algorithm, meaning that it may produce incorrect results with a small probability. However, the probability of error can be made arbitrarily small by increasing the number of iterations (k).

2.Sieve of Eratosthenes

```python
def sieve_eratosthenes(limit):
    primes = []
    is_prime = [True] * (limit + 1)
    for num in range(2, isqrt(limit) + 1):
        if is_prime[num]:
            primes.append(num)
            for multiple in range(num * num, limit + 1, num):
                is_prime[multiple] = False
    for num in range(isqrt(limit) + 1, limit + 1):
        if is_prime[num]:
            primes.append(num)
    return primes
```

Fundamentals:

The Sieve of Eratosthenes is an ancient algorithm for finding all prime numbers up to a specified limit.

It works by iteratively marking the multiples of each prime, starting from 2, as composite (not prime).

Algorithm Steps:

The algorithm maintains a boolean array (is_prime) where each index represents a number from 0 to the given limit. It initializes all elements of is_prime to True initially, assuming all numbers are prime.

Starting from the first prime number, which is 2, the algorithm iterates through the array. For each prime number found (starting with 2), it marks its multiples as non-prime by setting the corresponding elements in the is_prime array to False.

After completing this process for all numbers up to the square root of the limit, the remaining True values in the array represent prime numbers.

Time Complexity:

The time complexity of the Sieve of Eratosthenes is generally considered to be O(n log log n), where n is the limit.

The inner loop, marking multiples of each prime, runs in linear time for each prime found.

The outer loop iterates up to the square root of the limit.

Space Complexity:

The space complexity of the algorithm is O(n).

It uses an additional boolean array (is_prime) to keep track of whether each number is prime or not.

Optimizations:

The algorithm can be optimized in various ways, such as starting the iteration from the square of the current prime, skipping even numbers (except for 2), and using bitsets instead of boolean arrays to reduce memory usage.

3.Atkin's Sieve

```python
def sieve_atkin(limit):
    is_prime = [False] * (limit + 1)
    sqrt_limit = int(isqrt(limit)) + 1

    for x in range(1, sqrt_limit):
        for y in range(1, sqrt_limit):
            n = 4 * x**2 + y**2
            if n <= limit and (n % 12 == 1 or n % 12 == 5):
                is_prime[n] = not is_prime[n]

            n = 3 * x**2 + y**2
            if n <= limit and n % 12 == 7:
                is_prime[n] = not is_prime[n]

            n = 3 * x**2 - y**2
            if x > y and n <= limit and n % 12 == 11:
                is_prime[n] = not is_prime[n]

    for i in range(5, sqrt_limit):
        if is_prime[i]:
            for j in range(i**2, limit + 1, i**2):
                is_prime[j] = False

    primes = [2, 3]
    for num in range(5, limit + 1):
        if is_prime[num]:
            primes.append(num)

    return primes
```

Fundamentals:

The Sieve of Atkin is an optimized algorithm for finding prime numbers up to a specified limit. It works by marking certain integers as either prime or composite based on specific patterns derived from modulo arithmetic.

Algorithm Steps:

The algorithm maintains a boolean array (is_prime) where each index represents a number from 0 to the given limit. It initializes all elements of is_prime to False initially.

It iterates through all possible pairs of x and y values within certain ranges, calculating values of n according to specific quadratic forms. For each calculated n, if it is within the limit and satisfies certain conditions (congruences modulo 12), the algorithm toggles the primality status of n.

After completing this process, the algorithm further eliminates non-prime numbers by marking multiples of the squares of prime numbers as composite.

Time Complexity:

The time complexity of the Sieve of Atkin is considered to be O(n / log log n), which makes it more efficient than the Sieve of Eratosthenes for large values of n.

The key improvement comes from avoiding the need to sieve all numbers up to the square root of n, as is done in the Sieve of Eratosthenes.

Space Complexity:

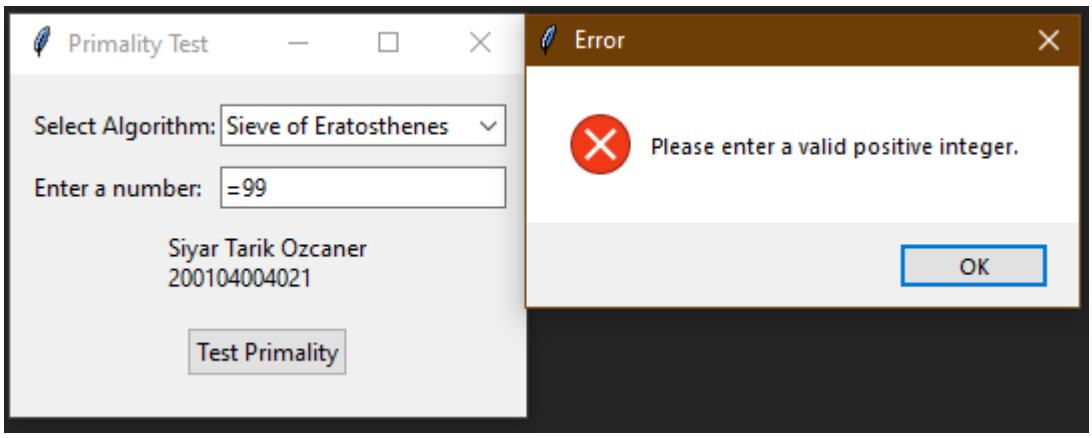
The space complexity is O(n) due to the boolean array used to store the primality status of each number.

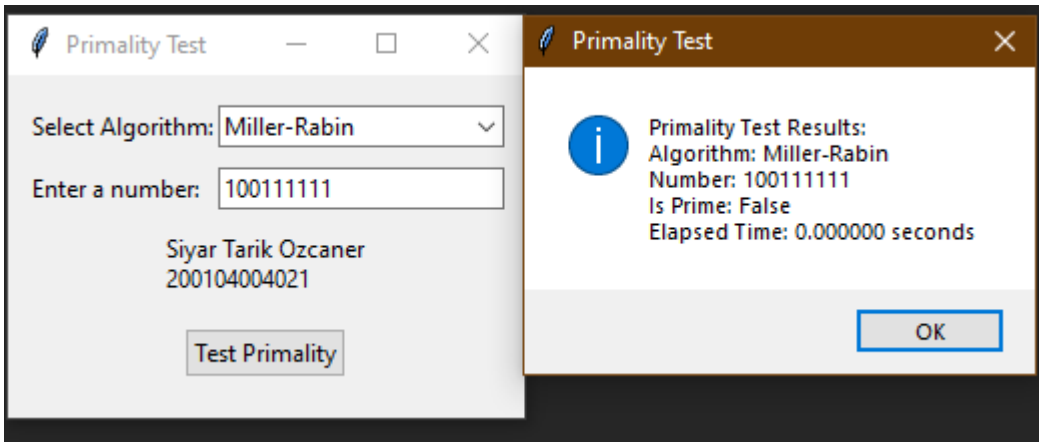
Optimizations:

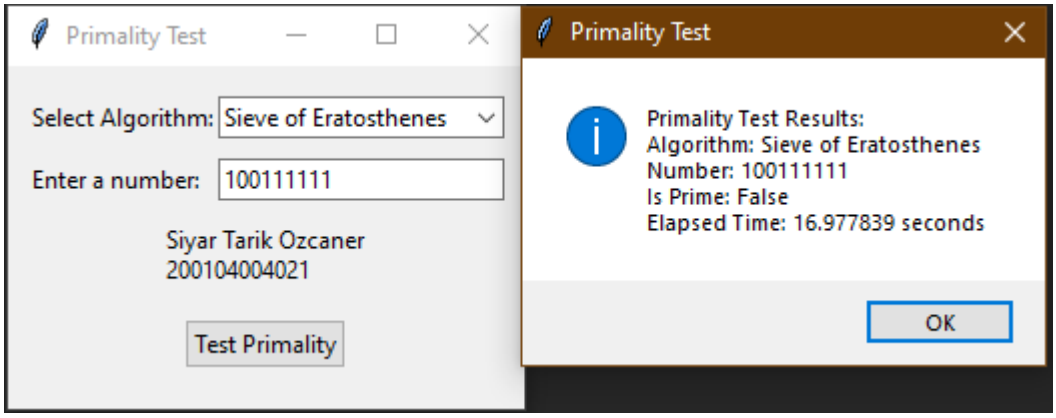
The algorithm utilizes specific patterns derived from modulo arithmetic to eliminate the need to sieve all numbers up to the square root of n.

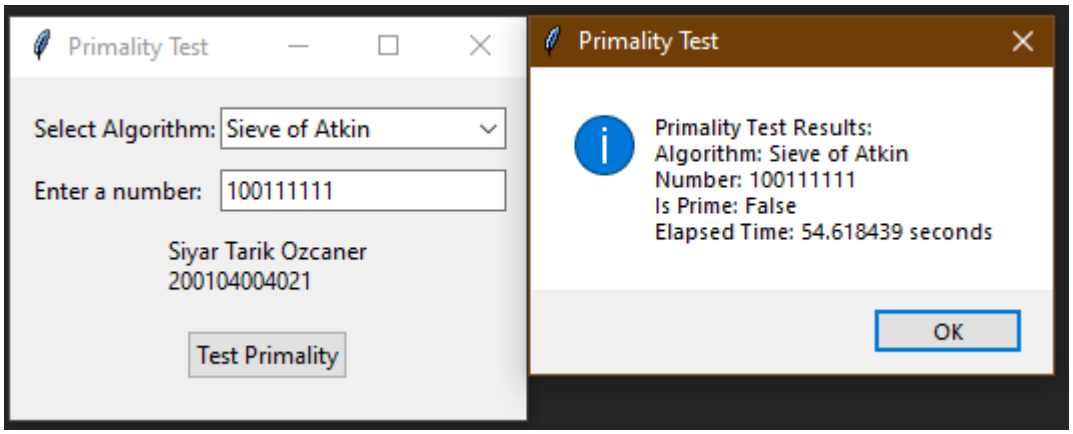
It skips marking multiples of certain values in the inner loops, making it more efficient.

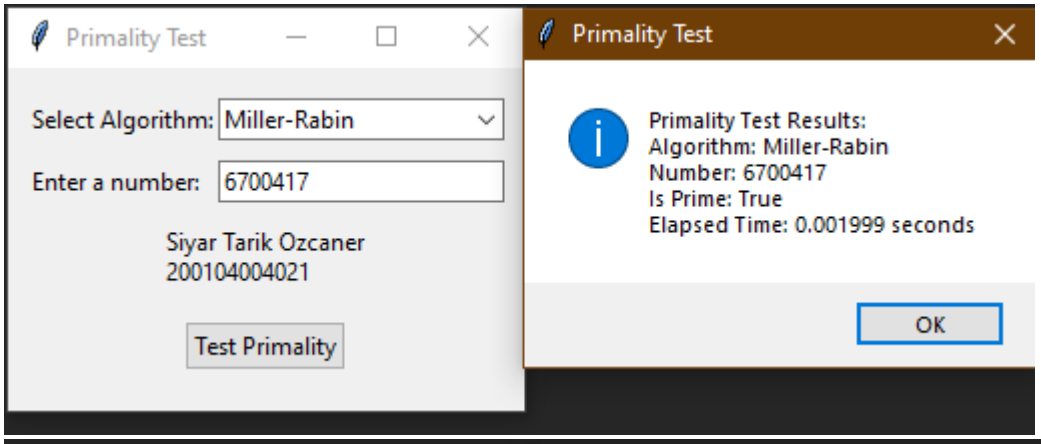
Outputs of the Program

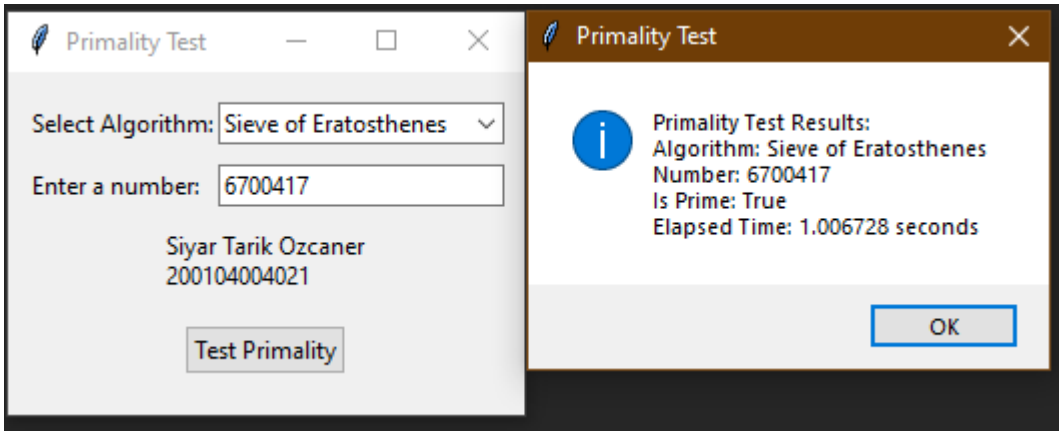


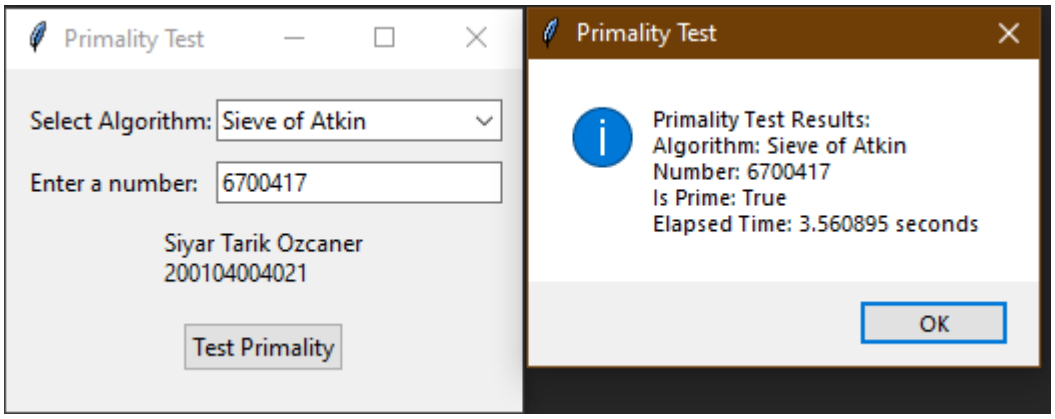





Primality Test
Select Algorithm: Sieve of Atkin
Enter a number: 100111111
Siyar Tarik Ozcaner
200104004021
Test Primality
Primality Test
Primality Test Results:
Algorithm: Sieve of Atkin
Number: 100111111
Is Prime: False
Elapsed Time: 54.618439 seconds
OK


Primality Test
Select Algorithm: Miller-Rabin
Enter a number: 6700417
Siyar Tarik Ozcaner
200104004021
Test Primality
Primality Test
Primality Test Results:
Algorithm: Miller-Rabin
Number: 6700417
Is Prime: True
Elapsed Time: 0.001999 seconds
OK


Primality Test
Select Algorithm: Sieve of Eratosthenes
Enter a number: 6700417
Siyar Tarik Ozcaner
200104004021
Test Primality
Primality Test
Primality Test Results:
Algorithm: Sieve of Eratosthenes
Number: 6700417
Is Prime: True
Elapsed Time: 1.006728 seconds
OK


Primality Test
Select Algorithm:
Sieve of Atkin
Enter a number:
6700417
Siyar Tarik Ozcaner
200104004021
Test Primality
Primality Test
Primality Test Results:
Algorithm: Sieve of Atkin
Number: 6700417
Is Prime: True
Elapsed Time: 3.560895 seconds
OK