Primality Tests

Janice Guo (vdq8tp), Edward Wei (nyw6dh), Jonathan Le (pqq2hu)

April 21, 2025

1 Introduction

In this project, we present a class of algorithms for verifying primes. Such algorithms are fundamental to modern cryptography because they exemplify the concept of "one-way functions", which are operations that are computationally easy to verify but difficult to reverse. Prime numbers serve as a crucial backbone to cryptographic systems because verification is efficient; given a large number, we can verify its primality relatively quickly using probabilistic or deterministic tests that run in polynomial time. Conversely, given a large composite number (especially the product of two large primes), finding its prime factors is computationally intensive.

This asymmetry between verification and computation creates a "trapdoor" function. This fundamental asymmetry enables many cryptographic protocols, including RSA encryption, where the security relies on the practical impossibility of factoring very large semiprime numbers, while verification remains tractable. Our algorithms explore this verification process, providing efficient methods to determine primality with mathematical certainty – a critical first step in building secure cryptographic systems that leverage the inherent computational hardness of the related factorization problem.

In the following sections, we present our methods for implementation and teaching methods for assisting undergraduate students to understand and build tools necessary for finding primes. We elaborate comprehensively on inner workings of each algorithm generating primes and then propose a programming challenge related to modern cryptocurrency applications designed to help students apply primality tests to practical scenarios.

2 Approach

The first intuitive approach is simply just to try and divide the number in question, n, by all integers less than it, excluding 1. This approach will always return the correct answer because it checks every single possible factor, but becomes too slow when n is large. Thus, we turn to Fermat's Little Theorem, which states that for some prime number n and a co-prime integer a, then $a^{n-1} \equiv 1 \bmod n$ must hold true.

Fermat's Little Theorem allows us to mathematically determine whether a number is prime or not without trying to divide n by every single valid divisor. While Fermat's Little Theorem generally does not hold for composite numbers, there are situations where n may pass the theorem if a is co-prime to n. Thus, the idea is to repeat this equality check with different random values of a. If we cannot find an a that does not satisfy the equality after several iterations, we can then conclude that n has a high probability of being prime.

To extend Fermat's Little Theorem, we know that a prime number must be odd. Thus, $n-1$ is even, and can be factorized and represented as $2^s \cdot d$. From this, we can factorize Fermat's Little theorem as follows:

$$
\begin{aligned}
a^{n-1} \equiv 1 \bmod n &\iff a^{2^s \cdot d} - 1 \equiv 0 \bmod n \\
&\iff (a^{2^{s-1} \cdot d} + 1)(a^{2^{s-1} \cdot d} - 1) \equiv 0 \bmod n \\
&\iff (a^{2^{s-1} \cdot d} + 1)(a^{2^{s-2} \cdot d} + 1)(a^{2^{s-2} \cdot d} - 1) \equiv 0 \bmod n \\
&\vdots \\
&\iff (a^{2^{s-1} \cdot d} + 1)(a^{2^{s-2} \cdot d} + 1) \cdots (a^d + 1)(a^d - 1) \equiv 0 \bmod n
\end{aligned}
$$

In order to satisfy the equality, one of the expressions in the parentheses must equal 0, so either $a^d \equiv 1 \bmod n$ or $a^{2^r \cdot d} \equiv -1 \bmod n$ must hold true for some value $0 \leq r \leq s-1$. While these equalities are stricter than the original Fermat inequality, there still exists a chance that a composite number will satisfy these with the right a, so we must again perform several iterations to conclude whether n is a composite or a strong probable prime. This method is described as the Miller-Rabin test.

3 Implementation

3.1 Trial Division

Implementing naive trial division is trivially straight forward as follows:

```
isPrime(n):
    for i = 2 ... n:
        if n is divisible by i:
            return false
    return true
```

This approach can be optimized by stopping at the largest integer less than or equal to $\sqrt{n}$, since if d is a divisor of n, either $d \leq \sqrt{n}$ or $n/d \leq \sqrt{n}$. Additionally, if a number is not divisible by 2, then it is not divisible by any other even number [fCP25]. Thus, we have as follows:

```
isPrime(n):
    if n is divisible by 2:
        return false
    i = 3
    while i <= sqrt(n):
        if n is divisible by i:
            return false
        i += 2
    return true
```

3.2 Fermat's Test

Theoretically, implementing Fermat's test is relatively straight forward as follows:

```
fermatPrime(n):
    for a = 2 ... n-2:
        if pow(a, n - 1) % n != 1:
            return false
    return true
```

However, checking all values for a is actually much slower than trial division, so we simply just run Fermat's test k times, each with some randomly chosen a to get a probably prime number test. Additionally, we can manually check some small primes. Finally, we optimize using binary exponentiation, instead of multiplying the base some number of times in a row [fCP23]. Binary exponentiation takes advantage of the fact that $a^{x+y} = a^x \cdot a^y$ and $a^{2x} = a^x \cdot a^x = (a^x)^2$. Thus, we can vastly improve

runtime when calculating a^b if we simply multiply the powers $a^1, a^2, a^4, \cdots$ for each power that exists in the binary representation of b. These powers are very easy to compute, since their values are simply the square of the previous power. For example:

$$\begin{aligned} 3^{13} &= 3^{1101_2} \\ &= 3^{1000_2} \cdot 3^{100_2} \cdot 3^{1_2} \\ &= 3^8 \cdot 3^4 \cdot 3 \end{aligned}$$

Since the modulo operator does not affect multiplication, we can add a modulus at every step in the binary exponentiation. Thus, we have an optimized implementation as follows:

```
binpowmod(a, b, m):
    result = 1
    while b > 0:
        a = a % m
        if binary b ends in 1:
            result = result * a % m
        raise a to the next power of 2
        bitshift b right by 1
    return result

probablyFermatPrime(n, k):
    if n < 4:
        if n equals 2 or 3, return true
        else return false
    for i = 1 ... k:
        choose random number in range [2, n - 2] for base a
        if binpowmod(a, n - 1, n) != 1:
            return false
    return true
```

3.3 Miller-Rabin Test

The factorized extension of Fermat's Little Theorem is called the Miller-Rabin test, which is theoretically implemented with a helper function as follows:

```
isComposite(a, n, d, s):
    num = a^(n - 1) % n
    if num == 1 or num == n - 1:
        return false
    for r = 1 ... s - 1:
        num = num * num % n
        if num == n - 1:
            return false
    return true

millerRabinPrime(n):
    s = 0
    d = n - 1
    while d is even:
        divide d by 2
        s++
    for a = 2 ... n-2:
        if isComposite(a, n, d, s):
            return false
    return true
```

Again, checking all values for a is extremely slow, so we simply just run the Miller-Rabin test k times, each with some randomly chosen a to get another probably prime number test. Additionally, we can manually check some small primes and some small prime divisors, along with using binary exponentiation. Thus, we have an optimized implementation as follows:

```
isComposite(a, n, d, s):
    num = binpowmod(a, d, n)
    if num == 1 or num == n - 1:
        return false
    for r = 1 ... s - 1:
        num = num * num % n
        if num == n - 1:
            return false
    return true

probablyMillerRabinPrime(n, k):
    if n < 4:
        if n equals 2 or 3, return true
        else return false
    if n is even or divisible by 3:
        return false
    s = 0
    d = n - 1
    while d is even:
        divide d by 2
        s++
    for i = 1 ... k:
        choose random number in range [2, n - 2] for base a
        if isComposite(a, n, d, s):
            return false
    return true
```

It has actually been proven that only a specific list of primes need to be used as base values to check for primality [fCP24]. For testing a 32-bit integer, it is only necessary to test the first four prime bases, and for a 64-bit integer only the first 12 prime bases. Assuming that n is a 64-bit integer or smaller, we can have the following deterministic implementation instead of trying random values for a:

```
deterministicMillerRabinPrime(n):
    if n < 2:
        return false
    s = 0
    d = n - 1
    while d is even:
        divide d by 2
        s++
    for a = the next prime value in the first 12 primes:
        if n == a:
            return true
        if isComposite(a, n, d, s):
            return false
    return true
```

4 Programming Challenge

For our programming challenge, we demonstrate a common application of prime verification in cryptocurrency. The students are tasked with verifying transactions on a simplified blockchain that simply appends transactions to a chain (of text), where transactions are approved only when the resulting

chain hashes to a prime number. This is a common principle behind proof-of-work blockchains, where the "miner" submits a nonce that can be easily computationally verified as "correct" (a prime hash in our case), but must have utilized some significant computational work to produce such an outcome (the "proof-of-work").

The key twists are that students must implement the hashing mechanism to convert the chain to a number, perform prime validation very rapidly (a naive approach will not survive the rapidly growing hash sizes), and track the valid transactions to be applied in downstream transactions. The hashing algorithm is a simple XOR-based hash modified to produce odd numbers, while transactions can be tracked with some string addition.

5 Key Solution Ideas

The primary idea of the programming challenge is to utilize the fastest approach for prime verification. The naive solution will most fail test cases due to time constraints, growing rapidly. Methods based on Fermat's Little Theorem will fail in test cases that generate Carmichael numbers. The intended solution is to utilize the Miller-Rabin test, either doing a probabilistic search (with reasonable iterations to guarantee success, about 10-20), or use a set of deterministic bases (while the first 12 will only guarantee verification for a 64 bit number, the test cases generate up to 128 bit numbers. The first 13 primes will guarantee verification up to 3,317,044,064,679,887,385,961,981).

6 Conclusion

In this project, we have presented a comprehensive overview of primality testing algorithms, ranging from the straightforward trial division to the more sophisticated probabilistic methods like Fermat's test and the Miller-Rabin test. These algorithms form a critical foundation for modern cryptographic systems by leveraging the fundamental asymmetry between verification and factorization – the very essence of one-way functions that cryptography relies upon.

Our implementation section demonstrated the progression from naive approaches to optimized algorithms, providing clear pseudocode examples that undergraduate students can readily understand and implement. By incorporating techniques such as binary exponentiation, we have shown how mathematical insights can dramatically improve algorithmic efficiency.

While both the Fermat and Miller-Rabin tests are probabilistic in nature, it is worth noting that their failure rate is remarkably low. The Fermat test does have a non-trivial set of numbers known as Carmichael numbers that will consistently pass despite being composite. However, the Miller-Rabin test does not suffer from this particular weakness. In practice, the probability of a composite number passing the Miller-Rabin test with k random bases is at most 4^{-k}, making it effectively correct for cryptographic applications, due to geometrically decreasing probability with respect to number of random bases.

Through our programming challenge and teaching methods, we have provided undergraduate students with the tools necessary to not only understand these algorithms theoretically but also implement them in practical scenarios related to modern cryptocurrency applications. This approach bridges the gap between abstract mathematical concepts and real-world applications, fostering a deeper appreciation for the role of prime numbers in securing digital communications.

References

[fCP23] Algorithms for Competitive Programming. Binary exponentiation, 2023.

[fCP24] Algorithms for Competitive-Programming. Primality tests, Apr 2024.

[fCP25] Algorithms for Competitive Programming. Integer factorization, 2025.