An Efficient Deterministic Primality Test

Joseph M. Shunia

December 2023

[Draft]

Abstract

A deterministic primality test with a polynomial time complexity of $O(\log(n)^3 \log\log(n))$ is presented. The test posits that an integer n satisfying the conditions of the main theorem is prime. Combining elements of number theory and combinatorics, the proof operates on the basis of simultaneous modular congruences relating to binomial transforms of powers of two.

1 Introduction

Primality testing has seen remarkable advancements over the past few decades. A significant breakthrough in this field was the AKS primality test, introduced by Agrawal, Kayal, and Saxena (2002) [1]. The AKS test was the first to offer determinism and polynomial-time complexity, a monumental achievement that resolved a longstanding open question in computational number theory [2]. However, despite its theoretical importance, the AKS test has practical limitations due to its relatively high polynomial time complexity, rendering it inefficient for most applications. Agrawal, Kayal, and Saxena gave a time complexity of $O(\log(n)^{12})$ for the AKS test [1]. This bound was lowered significantly by Lenstra and Pomerance (2011) to $O(\log^6(n))$ [3]. Despite this reduction, AKS remains impractical and is mostly unused.

In the field of cryptography, the unique properties of prime numbers are widely exploited to create cryptographic primitives. It is often the case that many large primes must be generated in rapid succession [4]. To make these cryptographic operations practical, fast probabilistic primality tests such as the Baille-PSW primality test (BPSW) [5] or Miller-Rabin (MR) [6] [7] are used instead of AKS when searching for large primes. Probabilistic primality tests are by definition non-deterministic and may erroneously report a composite integer as being prime. Composite integers which pass a probabilistic primality test are relatively rare and are known as psuedoprimes (PSPs) for the respective test [8]. When generating primes for cryptographic purposes, probabilistic primality tests are often combined or repeated with different parameters in order to achieve an acceptable error-bound that makes it almost certain that no composite integer will pass. However, reducing the error-bound requires additional compute and increases running-time, creating a trade-off.

We present a new deterministic primality test that operates in polynomial time with a time complexity of $O(\log(n)^3 \log\log(n))$. This efficiency gain opens new avenues for practical applications, particularly in cryptography, where fast and reliable primality testing is desirable [9]. Our main theorem posits a condition for an odd integer n to be prime, based on specific modular congruences related to the binomial transforms of powers of 2. The basis for our test is the following main theorem: Let n be an odd integer satisfying $2^{n-1} \equiv 1 \pmod{n}$. Denote D as the least integer strictly greater than 2 and less than n which does not divide $n-1$. Then, n is prime if and only if a set of simultaneous modular congruences involving D, n, and binomial coefficients hold.

This paper is structured as follows: We begin by presenting the main theorem and its proof, substantiated by two critical lemmas. The first lemma demonstrates the test's validity for odd prime numbers, while the

second confirms its failure for odd composite numbers. Through these lemmas, we establish the deterministic nature of our test. We then describe the algorithm used to compute our test and analyze its computational complexity. We also give a pseudocode implementation for our test to show how it can be implemented.

2 Main Theorem

Theorem 1. Let n be an odd integer > 3 satisfying $2^{n-1} \equiv 1 \pmod{n}$. Denote D as the least integer greater than 2 and less than n which does not divide $n-1$. Then, n is prime if and only if the following congruence holds:

$$1 + 2^{\left\lfloor \frac{n-1}{D} \right\rfloor} \equiv \left(1 + 2^{\left\lfloor \frac{n-1}{D} \right\rfloor}\right)^n \equiv \sum_{k=0}^{n} \binom{n}{k} 2^{\left\lfloor \frac{k}{D} \right\rfloor} \pmod{n} \tag{1}$$

2.1 Supporting Lemmas

Lemma 1. Let n be an odd composite integer > 3 satisfying $2^{n-1} \equiv 1 \pmod{n}$. Denote D as the least integer greater than 2 and less than n which does not divide $n-1$. Then, $2^{\left\lfloor \frac{n-1}{D} \right\rfloor} \not\equiv 1 \pmod{n}$.

Proof. We are given odd composite $n > 3$ with $2^{n-1} \equiv 1 \pmod{n}$. By the properties of the order of an integer modulo composite n, the smallest k such that $2^k \equiv 1 \pmod{n}$, that is $k = \mathrm{ord}_n(2)$, must be a divisor of $n-1$. Hence, $\mathrm{ord}_n(2) \mid n-1$. Since $\left\lfloor \frac{n-1}{D} \right\rfloor$ is strictly less than $n-1$ and D does not divide $n-1$, it follows that $2^{\left\lfloor \frac{n-1}{D} \right\rfloor} \not\equiv 1 \pmod{n}$. □

2.2 Proof of the Main Theorem

Proof of Theorem 1. Let n be an odd integer > 3 satisfying $2^{n-1} \equiv 1 \pmod{n}$.

Define $f(x) = 2^{\left\lfloor \frac{x-1}{D} \right\rfloor}$ and D as the least integer greater than 2 that does not divide $n-1$. We begin with the congruence:

$$(1 + f(n))^n \equiv \sum_{k=0}^{n} \binom{n}{k} f(k+1) \pmod{n} \tag{2}$$

Using the binomial theorem, we expand the left-hand side:

$$\sum_{k=0}^{n} \binom{n}{k} f(n)^k \equiv \sum_{k=0}^{n} \binom{n}{k} f(k+1) \pmod{n} \tag{3}$$

Rewriting this, we must have:

$$\sum_{k=0}^{n} \binom{n}{k} \left(f(n)^k - f(k+1)\right) \equiv 0 \pmod{n}. \tag{4}$$

As a necessary condition of our test, we also require:

$$\sum_{k=0}^{n} \binom{n}{k} f(n)^k \equiv \sum_{k=0}^{n} \binom{n}{k} f(k+1) \equiv 1 + f(n) \pmod{n} \tag{5}$$

By the binomial theorem, we isolate the inner terms:

$$1 + f(n)^n + \sum_{k=1}^{n-1} \binom{n}{k} f(n)^k \equiv 1 + f(n) + \sum_{k=1}^{n-1} \binom{n}{k} f(k+1) \equiv 1 + f(n) \pmod{n} \tag{6}$$

Setting common terms to zero:

$$\sum_{k=1}^{n-1} \binom{n}{k} f(n)^k \equiv \sum_{k=1}^{n-1} \binom{n}{k} f(k+1) \equiv 0 \pmod{n} \tag{7}$$

We have defined n such that $2^{n-1} \equiv 1 \pmod{n}$, which implies $\sum_{k=1}^{n-1} \binom{n}{k} \equiv 0 \pmod{n}$. Plugging it in:

$$\sum_{k=1}^{n-1} \binom{n}{k} \equiv \sum_{k=1}^{n-1} \binom{n}{k} f(n)^k \equiv \sum_{k=1}^{n-1} \binom{n}{k} f(k+1) \equiv 0 \pmod{n} \tag{8}$$

Subtracting $\sum_{k=1}^{n-1} \binom{n}{k}$ gives:

$$\left(\sum_{k=1}^{n-1} \binom{n}{k} f(n)^k \right) - \sum_{k=1}^{n-1} \binom{n}{k} \equiv \left(\sum_{k=1}^{n-1} \binom{n}{k} f(k+1) \right) - \sum_{k=1}^{n-1} \binom{n}{k} \equiv 0 \pmod{n} \tag{9}$$

Combining the summations:

$$\sum_{k=1}^{n-1} \left(\binom{n}{k} f(n)^k - \binom{n}{k} \right) \equiv \sum_{k=1}^{n-1} \left(\binom{n}{k} f(k+1) - \binom{n}{k} \right) \equiv 0 \pmod{n} \tag{10}$$

$$\sum_{k=1}^{n-1} \left(\binom{n}{k} f(n)^k - \binom{n}{k} f(k+1) - \binom{n}{k} \right) \equiv 0 \pmod{n} \tag{11}$$

Factoring out the common $\binom{n}{k}$ from the inner terms gives:

$$\sum_{k=1}^{n-1} \binom{n}{k} \left(f(n)^k - f(k+1) - 1 \right) \equiv 0 \pmod{n} \tag{12}$$

Looking back at eq. (7), we must also have:

$$\sum_{k=1}^{n-1} \binom{n}{k} f(n)^k \equiv \sum_{k=1}^{n-1} \binom{n}{k} f(k+1) \equiv 0 \pmod{n} \tag{13}$$

$$\sum_{k=1}^{n-1} \binom{n}{k} \left(f(n)^k - f(k+1) \right) \equiv 0 \pmod{n} \tag{14}$$

Which implies:

$$\sum_{k=1}^{n-1} \binom{n}{k} \left(f(n)^k - f(k+1) - 1 \right) \equiv \sum_{k=1}^{n-1} \binom{n}{k} \left(f(n)^k - f(k+1) \right) \equiv 0 \pmod{n} \tag{15}$$

For the above congruence, there are three potential cases we must examine.

Case 1: n is prime

In this case, the congruence always holds. By the binomial theorem, $\binom{n}{k} \equiv 0 \pmod{n}$ for $0 < k < n$ and the congruence simplifies trivially to zero.

Case 2: n is composite with $f(n) \equiv 1 \pmod{n}$

In this case, the congruence may or may not hold. With $f(n) \equiv 1 \pmod{n}$, the congruence simplifies significantly, making it possible:

$$\sum_{k=1}^{n-1} \binom{n}{k} f(k+1) \equiv \sum_{k=1}^{n-1} \binom{n}{k} (1 - f(k+1)) \equiv 0 \pmod{n} \tag{16}$$

$$\sum_{k=1}^{n-1} \binom{n}{k} f(k+1) \equiv \sum_{k=1}^{n-1} \left(\binom{n}{k} - \binom{n}{k} f(k+1) \right) \equiv 0 \pmod{n} \tag{17}$$

$$\sum_{k=1}^{n-1} \binom{n}{k} f(k+1) \equiv \sum_{k=1}^{n-1} \binom{n}{k} - \sum_{k=1}^{n-1} \binom{n}{k} f(k+1) \equiv 0 \pmod{n} \tag{18}$$

$$\sum_{k=1}^{n-1} \binom{n}{k} f(k+1) \equiv \sum_{k=1}^{n-1} \binom{n}{k} f(k+1) \equiv 0 \pmod{n} \tag{19}$$

$$\sum_{k=1}^{n-1} \binom{n}{k} f(k+1) \equiv 0 \pmod{n} \tag{20}$$

Case 3: n is composite with $f(n) \not\equiv 1 \pmod{n}$

In this case, the congruence cannot hold.

Since $\sum_{k=1}^{n-1} \binom{n}{k} \equiv 0 \pmod{n}$, we may add or subtract $\sum_{k=1}^{n-1} \binom{n}{k}$ from our congruence an arbitrary number of times and n must still divide the sum. As a result, we must have:

$$\sum_{k=1}^{n-1} \binom{n}{k} \left(f(n)^k - f(k+1) - X \right) \equiv 0 \pmod{n}, \text{ for all } X \in \mathbb{Z}. \tag{21}$$

For composite n, there is at least one k such that $\binom{n}{k} \not\equiv 0 \pmod{n}$. Therefore, if X can be any value, then so to can our sum:

$$\sum_{k=1}^{n-1} \binom{n}{k} \left(f(n)^k - f(k+1) - X \right) \tag{22}$$

Hence, in order for our congruence to hold, all integers must be equivalent to $0 \pmod{n}$. Since $n \neq 1$, this is impossible.

Conclusion:

When n is an odd prime integer > 3, the congruence holds. When n is an odd composite integer > 3, by Lemma 1 we have $f(n) \not\equiv 1 \pmod{n}$ and therefore, the congruence cannot hold. Hence, the theorem is proven.

This completes the proof. □

3 Algorithm

INPUT: An integer $n > 1$.

1. If $n \equiv 0 \mod 2$:
   (a) If n equals 2, output PRIME.
   (b) Otherwise, output COMPOSITE.
2. If n equals 3, output PRIME.
3. If $2^{n-1} \not\equiv 1 \pmod{n}$, output COMPOSITE.
4. Find the least integer D that is greater than 2 and less than n which does not divide $n-1$.
5. Set $A = 2^{\lfloor \frac{n-1}{D} \rfloor} \bmod n$.
6. Set $B = (1+A)^n \bmod n$.
7. If $B \not\equiv 1 + A \pmod{n}$, output COMPOSITE.
8. Set $C = \sum_{k=0}^{n} \binom{n}{k} 2^{\lfloor \frac{k}{D} \rfloor} \bmod n$.
9. If $C \not\equiv 1 + A \pmod{n}$, output COMPOSITE.
10. Output PRIME;

3.1 Time Complexity Analysis

3.1.1 Algorithm Overview

The given algorithm is a primality test that involves several computational steps, including modular arithmetic and polynomial exponentiation in the ring $\mathbb{Z}/n\mathbb{Z}$.

3.1.2 Analysis of Individual Operations

1. **Check for Even** n:
   This step involves calculating $n \bmod 2$ and has a time complexity of $O(1)$.
2. **Modular Exponentiation** $2^{n-1} \mod n$:
   This step requires modular exponentiation with a $\log(n)$-digit base and a $\log(n)$-digit exponent. The time complexity of modular exponentiation is $O(\log(n)M(n))$.
3. **Finding** D:
   Finding the least integer $D > 2$ that does not divide $n-1$ takes at most $O(\log(n))$ steps, with each step requiring $O(1)$ time for the mod operation. Hence, the overall complexity is $O(\log(n))$.
4. **Computing** A **and** B:
   Each of these steps involves modular exponentiation similar to Step 2, and thus each has a time complexity of $O(\log(n)M(n))$.
5. **Computing** C:
   Computing C involves exponentiating a polynomial in the ring $\mathbb{Z}/n\mathbb{Z}$ with $O(\log(n))$ terms and summing the coefficients.
   (a) Summing the polynomial coefficients can be done in $O(\log(n)^2)$ time.

(b) Exponentiation using repeated squaring takes $O(\log(n))$ steps, and each step requires $O(M(n)\log(n))$ time due to the multiplication of polynomials of size $O(\log(n))$.

Therefore, the overall complexity for computing C is $O(\log(n)^2 M(n))$.

6. **Comparisons**:

The final steps involve comparisons which are $O(1)$ operations.

3.1.3 Overall Time Complexity

The dominant time complexity in the algorithm comes from computing C. Therefore, the overall time complexity of the algorithm is $T(n) = O(\log(n)^2 M(n))$.

Harvey and van Der Heoven (2021) have given an algorithm for integer multiplication which has a time complexity $M(n) = O(\log(n)\log\log(n))$ [10]. This would give our algorithm an overall time complexity of:

$$T(n) = O(\log(n)^2 M(n)) = O(\log(n)^2 \log(n)\log\log(n)) = O(\log(n)^3 \log\log(n)) \tag{23}$$

3.1.4 Conclusion

The overall complexity is polynomial in the size of n when expressed in terms of bit operations, making the algorithm efficient for large values of n.

3.2 Psuedocode Implementation

To demonstrate how our test may be implemented, we offer a pseudocode implementation of the key functions involved.

3.2.1 IsPrime Function

Require: An integer $n > 1$
  function IsPrime(n)
    if $n \bmod 2 = 0$ **then**
      if $n = 2$ **then**
        return true ▷ n is prime
      else
        return false ▷ n is composite
      end if
    end if
    if $n = 3$ **then**
      return true ▷ n is prime
    end if
    $fermat \leftarrow$ Pow$(2, n-1, n)$ ▷ Fermat pseudoprime test to base 2
    if $fermat \neq 1$ **then**
      return false ▷ n is composite
    end if
    $D \leftarrow 2$
    $log \leftarrow$ Log2(n)
    for $i \leftarrow 3$ **to** Max$(log, 3)$ **do**
      $D \leftarrow i$
      $m \leftarrow (n-1) \bmod D$

```
        if m ≠ 0 then
            break
        end if
    end for
    A ← Pow(2, ⌊(n − 1)/D⌋, n)
    expectedValue ← (A + 1) mod n
    B ← Pow(A + 1, n, n)
    if B ≠ expectedValue then
        return false                                          ▷ n is composite
    end if
    polyDegree ← D − 1                  ▷ Subtract 1 to account for zero indexing in arrays
    poly ← PolyPow([1, 1], n, n, polyDegree, [2])
    C ← PolyEval(poly, 1) mod n                  ▷ Evaluate at x=1 to sum coefficients
    if C ≠ expectedValue then
        return false                                          ▷ n is composite
    end if
    return true                                               ▷ n is prime
end function
```

3.2.2 PolyPow Function

```
function PolyPow(poly_A, k, n, d, polyMapping)
    polyB ← [1]                  ▷ Initialize polyB as a polynomial with constant term 1
    while k > 0 do
        if k mod 2 = 1 then
            polyB ← PolyMul(polyA, polyB, n)              ▷ Multiply polynomials modulo n
            polyB ← PolyReduce(polyB, d, polyMapping, n)          ▷ Reduce degree of polyB
            if k = 1 then
                break
            end if
        end if
        polyA ← PolyMul(polyA, polyA, n)                  ▷ Square polyA modulo n
        polyA ← PolyReduce(polyA, d, polyMapping, n)          ▷ Reduce degree of polyA
        k ← k/2
    end while
    return polyB
end function
```

3.2.3 PolyReduce Function

```
function PolyReduce(polyA, d, mappingPoly, n)
    if Length(polyA) ≤ d then
        return polyA
    end if
    polyB ← Clone(polyA)                          ▷ Copy the polynomial to a new array
    polyDegree ← Degree(mappingPoly)              ▷ Find degree of the mapping polynomial
    degreeDelta ← d − polyDegree
    for i ← Length(polyB) − 1 downto d + 1 do
        if polyB[i] = 0 then
            continue
        end if
```

```
        for j ← i − 1 − degreeDelta, k ← polyDegree downto 0 do
            polyB[j] ← polyB[j] + polyB[i] × mappingPoly[k]          ▷ Degree reduction step
        end for
        polyB[i] ← 0
    end for
    polyC ← new array of size d + 1
    for i ← 0 to min(LENGTH(polyC), LENGTH(polyB)) − 1 do            ▷ Take coefficients modulo n
        polyC[i] ← polyB[i]
        if n ≠ 0 then
            polyC[i] ← polyC[i] mod n
        end if
    end for
    return polyC
end function
```

References

[1] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in p. *Annals of Mathematics*, pages 781–793, 2002.

[2] Oded Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 2008.

[3] Hendrik W Lenstra and Carl Pomerance. Primality testing with gaussian periods. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 369(1951): 3376–3390, 2011.

[4] Hendrik W Lenstra. Factoring integers with elliptic curves. *Annals of mathematics*, pages 649–673, 1987.

[5] Robert Baillie and Samuel S Jr Wagstaff. Lucas pseudoprimes. *Mathematics of Computation*, 35(152): 1391–1417, 1980.

[6] Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1): 128–138, 1980.

[7] Gary L Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[8] Samuel S Jr Wagstaff. Pseudoprimes and a generalization of artin's conjecture. *Acta Arithmetica*, 41 (2):141–150, 1983.

[9] Carl Pomerance. The use of elliptic curves in cryptography. *Advances in Cryptology*, pages 203–208, 1984.

[10] Joris van Der Hoeven David Harvey. Integer multiplication in time o(n log n). *Annals of Mathematics*, 2021. doi: 10.4007/annals.2021.193.2.4.hal-02070778v2.