MATH 350 Final Project

Srikur Kanuparthy, Nicolas Morabito, Tyler Basile

Spring 2024

1 Introduction & Background

The natural numbers greater than 1 that govern our world can be categorized into two groups – prime or composite. Prime numbers, the building blocks of arithmetic, are natural numbers greater than 1 that have no positive divisors other than 1 and themselves. The importance of prime numbers lies in their pivotal role in number theory, particularly through the Fundamental Theorem of Arithmetic, which states that every integer greater than 1 is either a prime itself or can be factored uniquely as a product of primes, disregarding the order of factors. This distinct property underscores the centrality of primes within the mathematical landscape and as its name suggests, is the building block for a vast amount of modern mathematics.

The study of prime numbers is not a recent phenomenon; rather, mathematicians throughout history have attempted to methods for computing such numbers. Ancient mathematicians such as Euclid and Eratosthenes contributed significantly to our understanding and identification of primes. Euclid, in his seminal work Elements, provided a theorem (now known as Euclid's Theorem) that there are an infinite amount of primes. Furthermore, Eratosthenes developed the Sieve of Eratosthenes, an efficient algorithm to filter out prime numbers up to a specified integer, demonstrating an early insight into algorithmic design for prime identification.

Over the centuries, the quest to discover larger primes and to develop faster and more efficient methods for testing primality has continued to evolve. From the simple trial division method to more sophisticated techniques such as the Sieve of Atkin, mathematicians have striven to reduce the computational complexity associated with prime testing. This ongoing development reflects the increasing importance of primes in more complex mathematical theories and in various applications.

The real-world applications of prime numbers are both broad and significant, particularly in the realm of cryptography. Prime numbers are integral to several encryption algorithms, including RSA, one of the first public-key cryptography systems that is still widely used for secure data transmission. The security of RSA is fundamentally dependent on the difficulty of factoring large integers, a task that becomes more challenging as the primes used increase in size. This cryptographic application underscores the necessity for efficient primality test-

ing, which ensures the quick and reliable identification of large prime numbers.

The ASK Primality Test represents a significant advancement in the field of computational mathematics, offering a novel approach to the challenge of quickly verifying the primality of numbers. It was developed by Manindra Agrawal, Neeraj Kayal, and Nitin Saxena (AKS), three computer scientists at the Indian Institute of Technology, Kanpur. Developed from a synthesis of algebraic and number-theoretic concepts, the ASK test leverages specific mathematical properties to enhance the speed and efficiency of primality testing, beyond what previous methods have achieved. In this paper, we will explain the theoretical framework of the algorithm, compare its effectiveness and efficiency with other modern methods, and discuss its potential implications for both theoretical mathematics and practical applications in technology and cryptography.

2 Formal Definitions

2.1 Definition of primality; naive testing methods

It is widely accepted that Euclid was the first to publish on prime numbers in approximately 300BC. In his *Elements*, Euclid defined prime numbers as "measured by a unit alone". Euclid also proved, famously elegantly, that there are infinitely many primes.

Formally, a prime number is a positive integer that cannot be divided by a positive integer other than itself or one. This definition leads to the first and most naive algorithm:

Algorithm 1 Naive Test for Primality

```
function PRIME(n)                                  ▷ n is an integer greater than 1
    i ← 2
    while i < n do
        if n  mod i = 0 then                       ▷ This is the same as saying n/i ∈ ℤ⁺
            return false
        end if
        i ← i + 1
    end while
    return true
end function
```

This algorithm, while intuitive, requires a number of multiplications that scales linearly with the size of n. We can reduce the upper bound of i to $\sqrt{n}$ (since all $a * b = n$ has either a or $b \leq \sqrt{n}$). However, the algorithm is exponential in the size of the input n, meaning it is not a polynomial-time algorithm.

3 Basic Background for AKS

3.1 Asymptotic Notation and Runtime

There a few things we should note about our notation for runtime analysis. First, all logarithms are in base 2. We will also use Big-O notation for runtimes. Informally, they are defined as follows:

$O(f)$ indicates that f is a tight upper bound.
$\tilde{O}(f)$ indicates that f is an upper bound, ignoring logarithmic factors.
$\Omega(f)$ indicates that f is a tight lower bound.
$\Theta(f)$ indicates that f is a tight bound. That is, $O(f) = \Omega(f)$.

Note the following properties of Big-O notation:

$O(f) + O(g) = O(\max(f, g))$
$O(f) \cdot O(g) = O(f \cdot g)$

An algorithm is said to be *polynomial-time* if given an input of length n, all computations are completed in $O(n^k)$ time for some $k \geq 0$.

3.2 Algebraic Structures

Much of the work which follows involves algebraic structures, namely groups, rings, and fields. This section will discuss some of the necessary basics.

Definition. A *group* $(G, +, 0)$ is a set G with a binary operation $+ : G \times G \to G$ such that the following *group criteria* are held. For $a, b, c \in G$,

(i). $(a + b) + c = a + (b + c)$
(ii). $a + 0 = a = 0 + a$
(iii). For any element a, there exists an inverse $-a$ such that $a + (-a) = 0$.

In other words, a group is simply a set with an associative binary operation with inverses. For example, $\mathbb{Z}/n$ with addition is a group. A group where $+$ is commutative is called an *abelian group*.

Definition. A *ring* $(R, \cdot, +, 1, 0)$ is a set R with two binary operations,

$$+ : R \times R \to R$$
$$\cdot : R \times R \to R$$

such that the following *ring criteria* are held. For $a, b, c \in R$,

(i). $(R, +, 0)$ forms an abelian group.
(ii). $(a \cdot b) \cdot c = a \cdot (b \cdot c)$

(iii). $a \cdot (b + c) = ab + ac$
(iv). $(a + b) \cdot c = ac + bc$.

In other words, $(R, +, 0)$ is an abelian group and $\cdot$ is associative and distributive with $+$. An example of a ring are the integers with addition and multiplication.

Definition. A *field* is a special type of ring in which additive inverses and multiplicative inverses always exist. One can think of a field as a ring which induces two new binary operations of subtraction and division.

Fact 1: For all primes p, $\mathbb{Z}/p$ is a field. This field is usually denoted $\mathbb{F}_p$.
Fact 2: Even stronger, every finite field F has p^n elements where p is prime and $n \geq 1$. This field is usually denoted $\mathbb{F}_q$ where $q = p^n$ or $\mathbb{F}_{p^n}$.

A *subfield* is a subset $K \subseteq F$ where F is a field and K is also a field with respect to the operations in F.

The *minimal polynomial* is $f(x) \in F[x]$ of an element $a \in F$ with the lowest degree monic polynomial such that a is a root. The minimal polynomial is unique.

3.3 Cyclotomic Polynomials

We now share some results regarding cyclotomic polynomials. Let μ_n be the group of n^{th} roots of unity over $\mathbf{Q}$.

Fact 1: As groups, $\mu_n \cong \mathbf{Z}/n$.
Fact 2: Primitive roots of unity can be obtained by the residue classes which are coprime with n. In other words, there are $\phi(n)$ primitive n^{th} roots of unity.

Definition. The n^{th} *cyclotomic polynomial*, denoted $\Phi_n(x) \in \mathbf{C}$ is the polynomial with roots exactly the $\phi(n)$ primitive n^{th} roots of unity.

Fact 3: $\deg(\Phi_n(x)) = \phi(n)$.

Definition. Let $f(x) \in \mathbb{Z}[x]$ be $a_n x^n + \cdots + a_0$. $f(x)$ is said to be *primitive* if all the a_i, $0 \leq i \leq n$, are pairwise coprime. That is, $\gcd(a_n, a_{n-1}, \ldots, a_0) = 1$.

Lemma. Let $p(x) \in \mathbb{Z}[x]$ be primitive. Then:

(i). If $q \mid f(x) \cdot g(x)$, with q prime and $f, g \in \mathbb{Z}[x]$, then $q \mid f(x)$ or $q \mid g(x)$.

(ii). If $p(x)$ is reducible in $\mathbb{Q}[x]$, then it is reducible in $\mathbb{Z}[x]$.

Proof. (i). Consider $\mathbb{F}_q$. Since, $q \mid f \cdot g$, it must be that $f \cdot g \equiv 0$ in $\mathbb{F}_q$. By properties of modulus operator, this implies that $f = 0$ or $g = 0$ in the ring.

(ii). Suppose $f \mid p$ in $\mathbb{Q}[x]$ with f having integer coefficients. Then there exists some g such that $f \cdot g = p$. Suppose $d \in \mathbb{Z}$ clears the denominator of $f \cdot g$. Then we have that $d \cdot p = f \cdot g$. Let p' be an arbitrary prime factor of d. From (i), we can conclude $p' \mid f$ or $p' \mid g$. But f is primitive, so this leaves only the case $p' \mid g$. Since p' is arbitrary, this applies to all prime factors of d; hence, $d \mid g$. Therefore, $p = f \cdot \frac{g}{d}$, which means p is reducible. □

Proposition. $\Phi_n(x)$ has integer coefficients. That is, $\Phi_n(x) \in \mathbb{Z}[x]$.

Proof. We prove via induction on n. For the base case, $n = 1$, $\Phi_1(x) = x - 1$ is trivial. For our induction hypothesis, suppose that $\Phi_k(x) \in \mathbb{Z}[x]$ for all $1 \leq k$. Then, letting $f(x) = \prod_{d|n, d \neq n} \Phi_d(x)$, we have

$$x^n - 1 = \Phi_n(x) \cdot f(x).$$

f divides $x^n - 1$ in $\mathbb{Q}(\zeta_n)[x]$ and hence $f \mid x^n - 1$ in $\mathbb{Q}[x]$. Applying the previous lemma, we can conclude f divides $x^n - 1$ in $\mathbb{Z}[x]$. Therefore, $\Phi_n(x) \in \mathbb{Z}[x]$. □

An important fact regarding cyclotomic polynomials in $\mathbb{F}_p$:

Lemma. Let n be a natural number and q be prime such that $\gcd(n, q) = 1$. Then $\Phi_n(x)$ factors into a product of $\frac{\phi(n)}{o_n(q)}$ distinct monic irreducibles in $\mathbb{F}_q[x]$.

Proof. We first show that Φ_n factors into $\frac{\phi(n)}{o_n(q)}$ monic irreducibles. We achieve this by showing that $o_n(q)$ is minimal such that $\zeta_n \in \mathbb{F}_{q^{o_n(q)}}$. We use the following fact.

Fact: $\zeta_n \in \mathbb{F}_{q^k} \iff \zeta_n^{q^k} = \zeta_n \iff q^k \equiv 1 \mod n$.

By definition of order, the minimal k satisfying this is $o_n(q)$. Therefore, $\zeta_n \in \mathbb{F}_{q^{o_n(q)}}$ but none of its subfields. So the degree of the unique minimal polynomial for ζ_n over $\mathbb{F}_q$ is of order $o_n(q)$. Thus, we have that Φ_n factors into $\frac{\phi(n)}{o_n(q)}$ distinct monic polynomials.

To see that these are distinct, consider $\Phi_n(x)$ viewed in $\mathbb{F}_q[x]$. Let $\Phi_n'(x)$ be its derivative (using the constant and power rules learned from calculus). Let $g \in \mathbb{F}_q[x]$ be irreducible and such that $g^2 \mid \Phi_n$. We then have that $\gcd(\Phi, \Phi') \neq 1$. Since $\Phi_n \mid (x^n - 1)$, we obtain

$$\gcd(x^n - 1, nx^{n-1}) \neq 1.$$

However, we now notice the following

$$\begin{aligned} &\gcd(x^n - 1, nx^{n-1}) \\ &= \gcd(x^n - 1, x^{n-1}) && \text{since } n \neq 0 \text{ in } \mathbb{F}_q \\ &= 1, \end{aligned}$$

contradicting what we found previously. Thus, $g^2 \nmid \Phi_n$. Therefore, the factors of Φ_n must be distinct. □

This concludes the knowledge required about cyclotomic polynomials to understand the AKS algorithm.

4 Ideas behind the Algorithm

4.1 Key Motivation

The key motivation for this algorithm comes from the following property of prime numbers.

Lemma. Let $a \in \mathbf{Z}, n \in \mathbf{N}$ with $\gcd(a, n) = 1$. Then

$$n \text{ is prime } \iff (x + a)^n \equiv x^n + a \mod n$$

While this lemma indeed yields itself to a primality test, it is very inefficient as it involves computing n coefficients. We could possibly reduce the number of computations needed by evaluating the polynomial modulo n and $x^r - 1$ for some small degree r. This natural means of thinking actually lends itself to a fascinating fact:

Fact: For a particular value of r, if

$$(x + a)^n \equiv x^n + a \mod (x^r - 1, n)$$

is true for a sufficient number of a's, then n is prime. We will discuss the proof of this later.

4.2 Picking the Correct r

Lemma. There exists $r \in \mathbf{N}$ such that:

(i). $r \leq \max(3, \lceil \log^5(n) \rceil)$
(ii). $o_r(n) > \log^2(n)$
(iii). $\gcd(r, n) = 1$

Proof. A fact from number theory states that for a natural number $m \geq 7$, $\text{lcm}([m]) \geq 2^m$. Then, assuming that $n \geq 2$, we have that $\lceil \log^5(n) \rceil > 10$. Thus, we see

$$\text{lcm}([\lceil \log^5(n) \rceil]) > 2^{\lceil \log^5(n) \rceil}.$$

note: $[n]$ indicates the set $\{1, 2, \ldots, n\}$.

Let r be such that it is the smallest integer which does *not* divide

$$N := n \cdot \prod_{i=1}^{\lfloor \log^2(n) \rfloor} (n^i - 1).$$

We claim that r satisfies the three given conditions. To see that (i) is satisfied, through some manipulation, we can determine

$$N \leq n^{1+2+\cdots+\log^2(n)} = n^{\frac{1}{2}(\log^4(n)+\log^2(n))} < n^{\log^4(n)} = 2^{\log^5(n)}.$$

Hence, using the fact from earlier, $N < \text{lcm}([\lceil \log^5(n) \rceil])$. Thus there exists $\ell \leq \lceil \log^5(n) \rceil$ with $\ell \nmid N$. However, r also has this property and r is minimal. Thus, $r < \lceil \log^5(n) \rceil$, satisfying the desired property.

(ii) is satisfied because for each $(n^i - 1)$ terms, we have that $n^i \not\equiv 1 \mod r$ (since we defined it such that r doesn't divide N).

To see that (iii) is satisfied, consider $\gcd(r, n)$. Clearly $\gcd(r, n) < r$ since $r \nmid N$ but $n \mid N$. Thus, $\frac{r}{\gcd(r,n)} \leq \max(3, \log^5(n))$ is such that it does not divide N. But, r is minimal. So $\frac{r}{\gcd(r,n)} = r$. Thus, $\gcd(r, n) = 1$. □

4.3 Introspectivity

We now introduce the following notion of introspectivity as it will be vital to the discussion of the algorithm's correctness.

Definition. Let $f \in \mathbf{Z}[x]$ and $m \in \mathbf{N}$. Then m is *introspective* for f if

$$f(x)^m \equiv f(x^m) \mod (x^r - 1, p)$$

for the carefully chosen r as in section 4.2 and p a prime divisor of n.

Example: p is introspective for $(x + a)$.

We now showcase some non-trivial properties of these instrospectives.

Lemma. Let $n \in \mathbf{N}$ have prime divisor p. If n, p are introspective for $(x + a)$, then $\frac{n}{p}$ is introspective for $(x + a)$.

Proof. From Fermat's Little Theorem, we can conclude that for $f \in (\mathbb{Z}/p)[x]$, $f(x)^p = f(x^p)$. Then we see

$$\begin{aligned} (x^{\frac{n}{p}} + a)^p &\equiv (x^n + a) \\ &\equiv (x + a)^n && n \text{ introspective} \\ &\equiv (x + a)^{p \cdot \frac{n}{p}} && p \text{ introspective} \end{aligned}$$

Letting $f = (x^{\frac{n}{p}} + a)$ and $g = (x + a)^{\frac{n}{p}}$, we see that $f^p = g^p$ in $(\mathbb{Z}/p[x])/(x^r - 1)$. Thus, taking the additive inverse of g, it must be that $f^p - g^p = 0$.

Fact: In the ring $\mathbb{F}_p[x]$, for all polynomials d, b,

$$(d + b)^p = d^p + b^p.$$

Applying this to f and $-g$, we have that $(f - g)^p = 0$. Notice that if $f - g = 0$, then $\frac{n}{p}$ is introspective in $(x + a)$. For clarity, let $h := (f - g)$.

From cyclotomic polynomials, we have that $(x^r - 1)$ factors into distinct irreducibles in $(\mathbb{Z}/p[x])$. Denote the i-th irreducible by $h_i(x)$. The Chinese Remainder Theorem then yields:

$$h^p \in \frac{(\mathbb{Z}/p)[x]}{x^r - 1} = \frac{(\mathbb{Z}/p)[x]}{\prod_i h_i(x)} = \prod_i \frac{(\mathbb{Z}/p)[x]}{h_i(x)}.$$

As $(x^r - 1) \mid h^p$, it must be that each $h_i(x)$ divides h. Therefore, $(x^r - 1) \mid h$ and so $h \equiv 0$. □

Lemma: Introspective numbers are closed under integer multiplication - Let $a, b \in \mathbb{N}$. If a, b are introspective for $f(x))$, then $a^i b^j$ is introspective for $f(x)$.

Proof. Because a is introspective for f, we have:

$$f(x)^{a*b} \equiv (f(x^a)^b) \mod (x^r - 1, p)$$

And by definition, for b we have

$$f(x)^b \equiv f(x^b) \mod (x^r - 1, p)$$

We can then substitute y^b for x in the above identity to obtain

$$f(y^a)^b \equiv f(y^{a*b} \mod (y^{a*t} - 1, p)$$

We can expand $(y^{a*r} - 1)$ to $(y^r - 1)(y^{r(a-1)} + y^{r(a-2)} + ... + y^r + 1)$. Note $(y^{a*r} - 1)$ is divisible by $(y^r - 1)$, the substitution above also holds mod $y^r - 1$ and thus

$$f(y^a)^b \equiv f(y^{a*b} \mod (y^r - 1, p)$$

.

Putting $f(x)^{a*b} \equiv (f(x^a)^b) \mod (x^r - 1, p)$ and $f(y^a)^b \equiv f(y^{a*b} \mod (y^r - 1, p)$ together gives us

$$f(x)^{a*b} \equiv f(x^{a*b}) \mod (x^r - 1, p)$$

□

Lemma: Introspective numbers are closed under polynomial multiplication - Let a be introspective for $f(x)$ and $g(x)$. Then a is introspective for $f(x) * g(x)$.

Proof. $(f(x) * g(x))^a = f(x)^a * g(x)^a \equiv f(x^a) * g(x^a) = (f * g)(x^a) \mod (x^r - 1, p)$ □

5 Algorithm

Note that Line 8 deals with edge cases regarding small primes.

Algorithm 2 AKS(n), n $\in \mathbb{Z} \geq 1$

1: If $\exists a, b > 1 \in \mathbb{N}$ such that $n = a^b$, then output COMPOSITE;
2: Find the minimal $r \in \mathbb{N}$ such that $o_r(n) > log^2(n)$;
3: **for** $a = 1$ to r do **do**
4: **if** $1 < \gcd(a, n)$ **then**
5: then Output COMPOSITE;
6: **end if**
7: **end for**
8: **if** $r \geq n$ **then**
9: Output PRIME
10: **end if**
11: **for** $a = 1 \rightarrow \lfloor \sqrt{\phi(r)} * log(n) \rfloor$ **do**
12: **if** $(x + a)^n \not\equiv x^n + a \mod (x^r - 1, n)$ **then** Output COMPOSITE;
13: **end if**
14: **end for**
15: Output PRIME;

6 Proof of Correctness and Runtime

6.1 Correctness Proof

Theorem. The given algorithm outputs PRIME $\iff$ n is prime.

Proof. ($\Longleftarrow$) Suppose n is prime. Then $n \neq a^b$ for any $a, b > 1$. Thus, line 1 of the algorithm will not output COMPOSITE. We can also say for all natural numbers m, $\gcd(m, n) = 1$ or n. Thus lines 4-6 will also not yield COMPOSITE. If n is prime, we know that $(x+a)^n \equiv (x^n+a)$ mod n, so the following for-loop in lines 11-14 does not return COMPOSITE. Thus, the only possibility is that the algorithm returns PRIME in line 15.

($\Longrightarrow$) Suppose the algorithm returns PRIME. If it returns PRIME when checking $r >= n$, then it must be that for all $m < n$, $\gcd(m, n) = 1$ at this point in the algorithm. Hence n is prime.

Suppose then that the algorithm returns PRIME in the final step. We will make the following assumptions:

(i). $p \mid n$ is a prime divisor.
(ii). $r < \lceil \log^5(n) \rceil$ and $o_r(n) > \log^2(n)$.
(iii). $\gcd(r, n) = 1$.
(iv). $l := \lfloor \sqrt{\phi(r)} \cdot \log(n) \rfloor$.

The for-loop of the algorithm checks that for all $1 \leq a \leq l$, whether n is introspective for $(x + a)$. Let $I = \{(\frac{n}{p})^i \cdot p^j \mid i, j \geq 0\}$ and $P = \{\prod_{a=0}^{l} (x + a)^{e_a} \mid e_a > 0\}$. The work done on introspectives in **(4.2)** show that all elements of I are introspective in all elements of P.

We define a group $I_r := \{i \mod r \mid i \in I\}$. In fact, this group is a multiplicative subgroup of $\mathbb{Z}/r$. Let $|I_r| = t$. Since $o_r(n) > \log^2(n)$, we also have that $t > \log^2(n)$. Moreover, I_r is generated by n and p.

Consider $\Phi_r(x)$ over $\mathbb{F}_p$. We know we can factor it into irreducibles of degree $o_r(p)$. Let $h(x)$ be an irreducible factor of $x^r - 1$ over $\mathbb{F}_p[x]$ with order $o_r(p)$.

We define another group $G := \{f \mod (h(x), p) \mid f \in P\}$. This group is generated by elements $x, x+1, x+2, \ldots, x+l$ in the field $\mathbb{F}_p(\zeta) \cong \mathbb{F}_p[x]/(h(x))$ where ζ is an r-th root of unity. Note that since $\deg(h(x)) = o_r(p)$, we have that $|\mathbb{F}_p(\zeta)| = p^{o_r(p)}$. This can be used to prove the following lemma.

Lemma. Let $f(x) \in G$. Suppose $f(x)^n = f(x^n)$ and $f(x)^p = f(x^p)$. Then $f(x)^{\frac{n}{p}} = f(x^{\frac{n}{p}})$.
note that this is an alternative to a lemma proven in section (4.3)

We now provide the following two lemmas.

Lemma A. $|G| \geq \binom{t+l}{t-1}$.

Proof. Let $F := \mathbb{F}_p[x]/(h(x))$. Since $h(x) \mid \Phi_r(x)$, x is an r-th root of unity in F. Let $f, g \in P$ be distinct such that both of their degrees are less than t.

Suppose for contradiction that $f = g$ in F. Given $m \in I$, we know that it is introspective with f and g. Thus, $f(x^m), g(x^m) \in F$. x^m is then a root of $f(z) - g(z)$ for all $m \in I_r$. By assumption (iv), $\gcd(m, r) = 1$, which implies that x^m is a primitive root of unity for all m. But $|I_r| = t$ and we assumed the degrees f and g to be less than t. Therefore, by contradiction, $f \neq g$ in F.

Thus, $x, x+1, x+2, \ldots, x+l$ are distinct in F, giving us $l+1$ polynomials in G. Through some combinatorics, we can conclude that there are $\binom{l+k}{k}$ polynomials with degree k in G. Therefore, a lower bound on the order of G is $\sum_{k=0}^{t-1} \binom{l+k}{k} = \binom{t+l}{t-1}$. □

Lemma B. If n is not a power of p, then $|G| < n^{\sqrt{t}}$.

Proof. Let $I' = \{(\frac{n}{p})^i \cdot p^j \mid 0 \leq i, j \leq \lfloor\sqrt{t}\rfloor\} \subset I$. Suppose that n is not a power of p. Then,

$$|I'| \geq (1 + \lfloor\sqrt{t}\rfloor)^2 > t.$$

Since $|I_r| = t$, then by the pigeonhole principle, there exists $u, v \in I'$ such that $u \equiv v \mod r$. Moreover, $x^u \equiv x^v \mod (x^r - 1)$. Let $f \in P$ be arbitrary. Then by introspectivity of u and v:

$$f(x)^u = f(x^u) = f(x^v) = f(x)^v.$$

In other words, $f(x)^u \equiv f(x)^v \mod (x^r - 1, p)$ in $F := \mathbb{F}_p[x]/(h(x))$. We claim that $g = x^v - x^u$ has at least $|G|$ roots in F. Noting that I' has a maximal element of $(\frac{n}{n}^{\lfloor\sqrt{t}\rfloor} \cdot p^{\lfloor\sqrt{t}\rfloor})$, we obtain

$$\deg(g) = v \leq (\frac{n}{p} \cdot p)^{\lfloor\sqrt{t}\rfloor} = n^{\lfloor\sqrt{t}\rfloor}.$$

Therefore, $G \leq n^{\sqrt{t}}$. □

With these two lemmas, we can now prove that n is prime. Since we know $t \geq o_r(n) > \log^2(n)$, we have that $t^2 > t \cdot \log^2(n)$, and hence yielding $t > \lfloor\sqrt{t}\log(n)\rfloor$. Recall now that $l = \lfloor\sqrt{\phi(r)} \cdot \log(n)\rfloor$. Then, from Lemma A., we obtain

$$\begin{aligned} |G| \geq \binom{t+l}{t-1} &\geq \binom{t+1+\lfloor\sqrt{t}\log(n)\rfloor}{\lfloor\sqrt{t}\log(n)\rfloor} \\ &\geq \binom{2\lfloor\sqrt{t}\log(n)\rfloor+1}{\lfloor\sqrt{t}\log(n)\rfloor}* \\ &> 2^{\lfloor\sqrt{t}\log(n)\rfloor+1} \geq 2^{\sqrt{t}\log(n)} = n^{\sqrt{t}}. \end{aligned}$$

*: this comes from I_r being a multiplicative subgroup of $\mathbb{Z}/r$.

Applying Lemma B., $|G| \leq n^{\sqrt{t}}$ if n is not a power of p. By the contrapositive, this means that n is a power of p in our scenario. However, the algorithm has already concluded that n is not a perfect power. Thus, $n = p^1$. Therefore, n is prime. □

6.2 Runtime Analysis

The runtime of the specific algorithm outlined in section 5 is very straightforward.

- Step 1, checking if n is composite, can be done several ways. In practice, it is simplest to check all the valid b'th roots of n for integer values. If any are integers, we know $n = a^b$ for some integers a and b. In this case we can just range b from 1 to $\lceil log_2(n)\rceil$ since we know 2 is the smallest integer value possible for a as a base. At worst this step is $log(n) * log(n) \in O(log^2(n))$.
- Step 2, finding r, is done in polynomial time since we know the upper bound for r is $log^5(n)$ by our earlier proof. Then we can just iterate through all the values from 1 to $log^5(n)$, and returning the first r such that $k > log^2(n)$ for $n^k \mod r \equiv 1$. The most practical way to determine if the order of n mod r is greater than $log^2(n)$ is to compute n^k mod r for all values of k from 1 to $log^2(n)$. If n^k mod r is ever equivalent to 0 or 1 we know the order is less than $log^2(n)$ and we can omit it. The first r to have n^k mod $r \not\equiv 1\ \forall k < log^2(n)$ will be our viable r. This step takes at most $log^2(n)$ multiplications for every r tested, and we test at most $log^5(n)$ values for r. Thus our runtime for this part is $O(log^2(n)) * O(log^5(n)) \in O(log^7(n))$.
- Step 3, checking the GCD of all numbers between 1 and r requires at most $log^5(n)$ checks if r is maximal. We know Euclid's GCD runs in $O(log^2(n))$ so this part takes $O(log^7(n))$

- Step 4, Checking if $r \geq n$, takes $O(1)$ time and is trivial to the total runtime.
- Step 5, checking the polynomial congruence for all values of a from 1 to $\lfloor \sqrt{(}\phi(r)) * log(n) \rfloor$ is the most computationally intensive part of the algorithm. In order to calculate $(x+a)^n$ we must $O(log(n))$ multiplications of polynomials of degree less than r with coefficients of size $O(log(n))$ (since all coefficients and polynomials are reduced by n and $x^r - 1$. Thus each congruence takes $O(log^7(n))$ operations in total, and since there are $\lfloor \sqrt{(}\phi(r)) * log(n) \rfloor$ congruences to calculate, we get $O(log^8(n) * \sqrt{\phi(r)}$ which is $O(log^8(n)) * \sqrt{(O(log^5(n))} = O(log^{8*5/2}(n)) = O(log^{10.5}(n))$

6.3 History of Runtime; Conjectures

The original paper by Agrawal, Kayal, and Saxena provided a proof for the time complexity of the algorithm to be $\tilde{O}(log^{12}(n))$. However, soon after the paper was published, many variants were proposed, and the authors released an update version of their original paper with the time complexity given as $\tilde{O}(log^{10.5}(n))$ (this is the version we explore). Within three years of the original paper's publication, the runtime had been reduced to $\tilde{O}(log^6(n))$ based on a variant by Carl Pomerance and Hendrik Lenstra. The authors have proposed a new variant that would reduce the theoretical time complexity all the way down to $\tilde{O}(log^3(n))$, but it depends on a conjecture - known as Agrawal's conjecture - which Pomerance and Lenstra suggest is probably false. The conjecture formally states that if we let n and r be two coprime positive integers, then if:

$$(X-1)^n \equiv X^n - 1 \bmod (n, X^r - 1)$$

then either n is prime or $n^2 \equiv 1 \bmod r$. Computationally, this result has been verified for $r < 100, n < 10^{10}$ and $r = 5, n < 10^{11}$, but a heuristic argument by Pomerance and Lenstra contends that there are infinitely many counterexamples. Their arguments are far too complex to be reasonably included here, but their heuristic shows that counterexamples to the conjecture appear with density $\frac{1}{n^\epsilon}$.

Roman Popovych has proposed a modified conjecture that may be true, assuming the heuristic argument above proves Agrawal's conjecture to be false. The modified conjecture stats that if we let n and r be two coprime positive integers, then if both of the statements below hold, then either n is prime or $n^2 \equiv 1 \bmod r$:

$$(X-1)^n \equiv X^n - 1 \bmod (n, X^r - 1)$$

$$(X+2)^n \equiv X^n + 2 \bmod (n, X^r - 1)$$

7 Properties of the Algorithm

In the world of primality-proving algorithms, AKS was the first to satisfy four specific properties. For several centuries, there have been algorithms that satisfy up to three of the following properties, but AKS was the first to satisfy all four. The properties are:

- Generality: The algorithm is compatible with any given input. There exist deterministic algorithms that are fast but only work with specific types of numbers, such as Mersenne numbers $(2^n - 1)$ or Fermat numbers $(2^{2^n} - 1)$.

- Polynomial-Time: As defined earlier in the paper, an algorithm runs in polynomial time if all computations take $O(n^k)$ operations, as opposed to exponential time ($O(2^{n^k})$) or worse.

- Deterministic: As opposed to a probabilistic algorithm, which might have a small, but non-zero chance of incorrectly returning prime for a composite number, a (correct) deterministic algorithm returns the same output value for the same input values.

- Unconditionally correct: The AKS algorithm does not depend on any unproven hypotheses. There exist other algorithms, such as the Miller-Rabin primality test, which has a fast, deterministic variant, but which depends on the generalized Riemann hypothesis, an important conjecture in mathematics that is as of yet unproven.

8 Importance

The AKS algorithm has immense theoretical value, especially since it satisfies the four properties above. However, even though it is polynomial time, it's high time complexity can easily be beat by several variants of probabilistic algorithms made to be deterministic. As such, the AKS algorithm is referred to as a galactic algorithm. The term, coined by computer scientists Richard Lipton and Ken Regan, is based on the idea that these algorithms "never will be used - at least not on terrestrial data sets".

The idea behind the term is that algorithms, such as AKS, are theoretically far superior to others, but their advantages (1) only appear for data so large that they never occur, or (2) their increase in complexity over other algorithms is much higher than their relatively small gain in performance.

In practical applications, probabilistic algorithms will almost always be used in place of AKS.

9 Experimental Implementation

In practice, AKS is computationally inefficient even though it is asymptotically polynomial. Often times, probabilistic methods are used in real world applications. However, we did code the algorithm into python, and we included some optimizations for polynomial multiplication and exponentiation that weren't necessarily baked into the default libraries.

1. **Testing**:

   On a set of 100,000 integers, our AKS implementation agreed with sympy's isprime function 100% of the time (as expected). Additionally, we tested famous primes of very large magnitude and got the expected correct results. This concluded the experimental testing portion of our AKS algorithm.

2. **Runtime**: The runtime of our AKS was rather slow in practice, especially on primes. Composite numbers showed asymptotic logarithmic behavior with the size of the input, but had constant coefficients orders of magnitudes below prime inputs. We speculate this is due to the fact that many composite integers are filtered out before even getting to

the polynomial congruence, and those that do quickly show inequality in the polynomial equivalency. See the last page for screenshots of the code snippets as well as the various patterns and logarithmic fits of experimental runtimes.

```python
import math
import array as arr
from fractions import gcd
import matplotlib.pyplot as plt
from sympy.ntheory.factor_ import totient
from sympy import symbols, expand, simplify, mod_inverse
```

```python
def is_perfect_power(n):
    # python's pow function is O(log(n)) with respect to either input, so this works in place of a manual binary search
    for b in range(2, int(math.ceil(math.log(n)))):
        a = math.pow(n, 1/b)
        #print(f"Testing value b: {b}. This yields a: {a:.15f}.")
        if int(a) ** b == n:
            return [n, a, b, True]

    return [n, 0, 0, False]
```

```python
def find_r(n):
   # Find smallest r such that the order of n mod r > log2(n)^2.

   maxK = math.log2(n)**2
   maxR = math.log2(n)**5
   nexR = True
   r = 1
   while nexR == True:
       r +=1
       nexR = False
       k = 0
       while k <= maxK and nexR == False:
           k = k+1
           if fastMod(n,k,r) == 0 or fastMod(n,k,r) == 1:
               nexR = True
   return(r)
```

```python
# this catches any n such that r is found to be gcd(n, r) != 1.
# otherwise the order of n mod r doesn't exist but n must be composite in this case
# returns true of any integers between 1 and r share a divisor with n besides 1 itself

def check_gcd(n, r):
    for a in range(1, r + 1):
        if math.gcd(a, n) > 1:
            # edge case that should never show up in the intended use case, but if r == n then n could still be prime
            if a == r and r == n:
                return False
            # n is definitely composite
            return True
    # n could still be prime
    return False
```

Figure 1: Initial Functions: Perfect Power, Find R, and Check_GCD

10 Code Snippets and Runtime Charts

```
def fastMod(base,power,n):
# Fast modular exponentiation
    r=1
    while power > 0:
        if power % 2 == 1:
            r = r * base % n
        base = base**2 % n
        power = power // 2
    return(r)
```

```
def fastPoly(base, power,r):
# Fast polynomial exponentiation mod x^r - 1
    x = arr.array('d',[],)
    a = base[0]

    for i in range(len(base)):
        x.append(0)
    x[(0)] = 1
    n = power

    while power > 0:
        if power % 2 == 1:
            x = multi(x,base,n,r)
        base = multi(base,base,n,r)
        power = power // 2

    x[(0)] = x[(0)] - a
    x[(n % r )] = x[(n % r )] - 1
    return(x)
```

```
def multi(a,b,n,r):
# Multiply two polynomials together quickly
    x = arr.array('d',[])
    for i in range(len(a)+len(b)-1):
        x.append(0)
    for i in range(len(a)):
        for j in range(len(b)):
            x[(i+j) % r ] += a[(i)] * b[(j)]
            x[(i+j) % r] = x[(i+j) % r] % n
    for i in range(r,len(x)):
            x=x[:-1]
    return(x)
```

Figure 2: More Optimal Arithmetic Functions

```
def aks(n):
# The main algorithm
    if is_perfect_power(n) == True:                    #step 1
        return('Composite')
    r = find_r(n)
    l = math.floor(math.sqrt(totient(r)*math.log(n)))

                                    #step 2

    for a in range(2,min(r,n)):                        #step 3
        if math.gcd(a,n) > 1:
            return('Composite')

    if n <= r:                                         #step 4
        return('Prime')

    x = arr.array('l',[],)                            #step 5
    for a in range(1,l):
        x = fastPoly(arr.array('l',[a,1]),n,r)
        if  any(x):
            return('Composite')
    return('Prime')                                    #step 6
```

Figure 3: Full AKS Algorithm

```python
# Store results
from sympy import isprime
times_prime_deterministic = []
inputs_prime_deterministic = []
begin = time.time()
for i in range(10, 10**3, 1):
    if i % 10000 == 0:
        print(f"Checkpoint at {i}")
    if isprime(i):
        start_time = time.time()
        aks(i)
        end_time = time.time()

        elapsed_time = end_time - start_time
        times_prime_deterministic.append(elapsed_time)
        inputs_prime_deterministic.append(i)
end = time.time()
```

```python
print(f"Total elapsed time: {end - begin} seconds")
plt.figure(figsize=(10, 5))
plt.plot(inputs_prime_deterministic, times_prime_deterministic, marker='o', linestyle='-', color='b')
plt.title('Prime-only AKS Performance')
plt.xlabel('Input Size')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()
```

```
Total elapsed time: 33.143385887145996 seconds
```

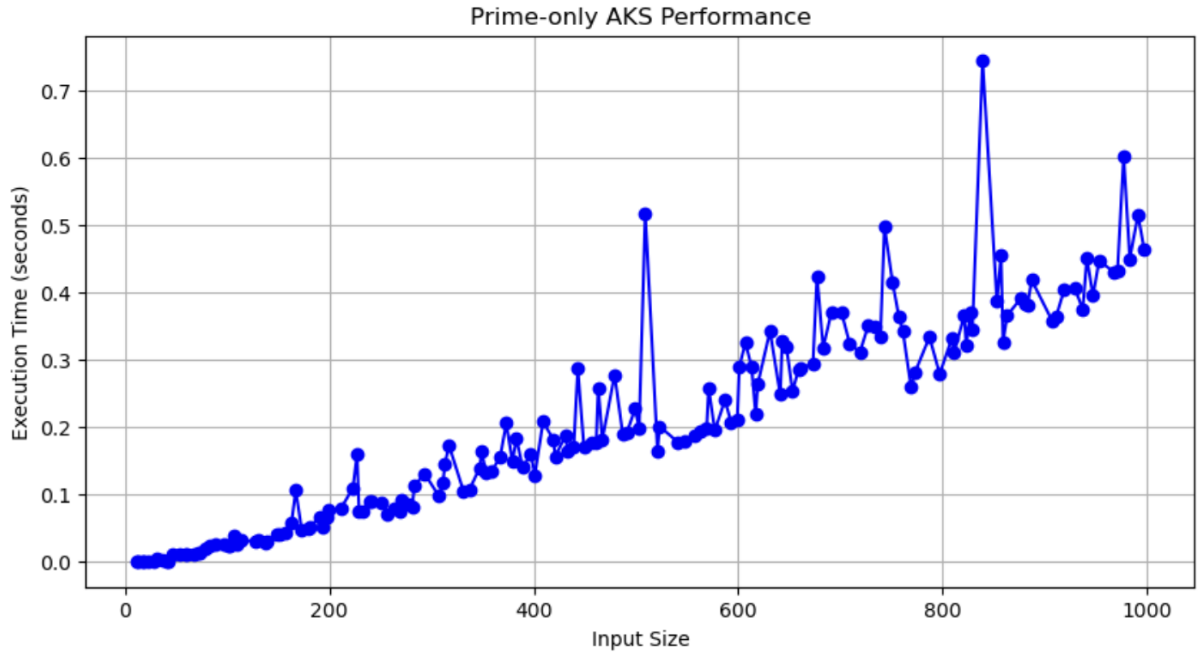

Figure 4: Performance of AKS on First 1000 Primes

```python
from sympy import isprime
times_prime_randomized = []
inputs_prime_randomized = []
begin = time.time()
for i in range(10, 10**5, 200):
    rand = random.randint(1, 100)
    if i % 10000 == 0:
        print(f"Checkpoint at {i}")
    if isprime(i + rand):
        print(f"Checking {i + rand}")
        start_time = time.time()
        aks(i + rand)
        end_time = time.time()

        elapsed_time = end_time - start_time
        times_prime_randomized.append(elapsed_time)
        inputs_prime_randomized.append(i)
end = time.time()
print(f"Total elapsed time: {end - begin} seconds")
plt.figure(figsize=(10, 5))
plt.plot(inputs_prime_randomized, times_prime_randomized, marker='o', linestyle='-', color='b')
plt.title('Randomized Prime-only AKS Performance')
plt.xlabel('Input Size')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()
```

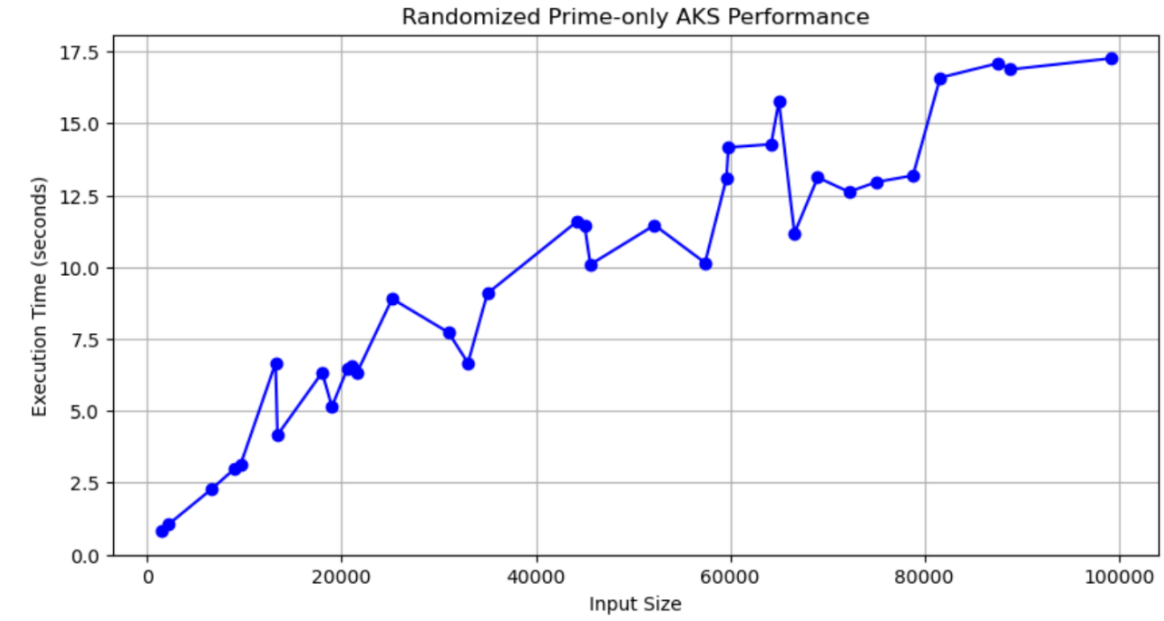

Figure 5: Runtime of Randomized Primality Testing on First 10^5 Integers - Prime Only

```python
from sympy import isprime
times_composite_randomized = []
inputs_composite_randomized = []
begin = time.time()
for i in range(10, 10**5, 200):
    rand = random.randint(1, 100)
    if not isprime(i + rand):
        start_time = time.time()
        aks(i + rand)
        end_time = time.time()

        elapsed_time = end_time - start_time
        times_composite_randomized.append(elapsed_time)
        inputs_composite_randomized.append(i)
end = time.time()
print(f"Total elapsed time: {end - begin} seconds")
plt.figure(figsize=(10, 5))
plt.plot(inputs_composite_randomized, times_composite_randomized, marker='o', linestyle='-', color='b')
plt.title('Randomized Composite-only AKS Performance')
plt.xlabel('Input Size')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()
```

```
Total elapsed time: 0.591555118560791 seconds
```

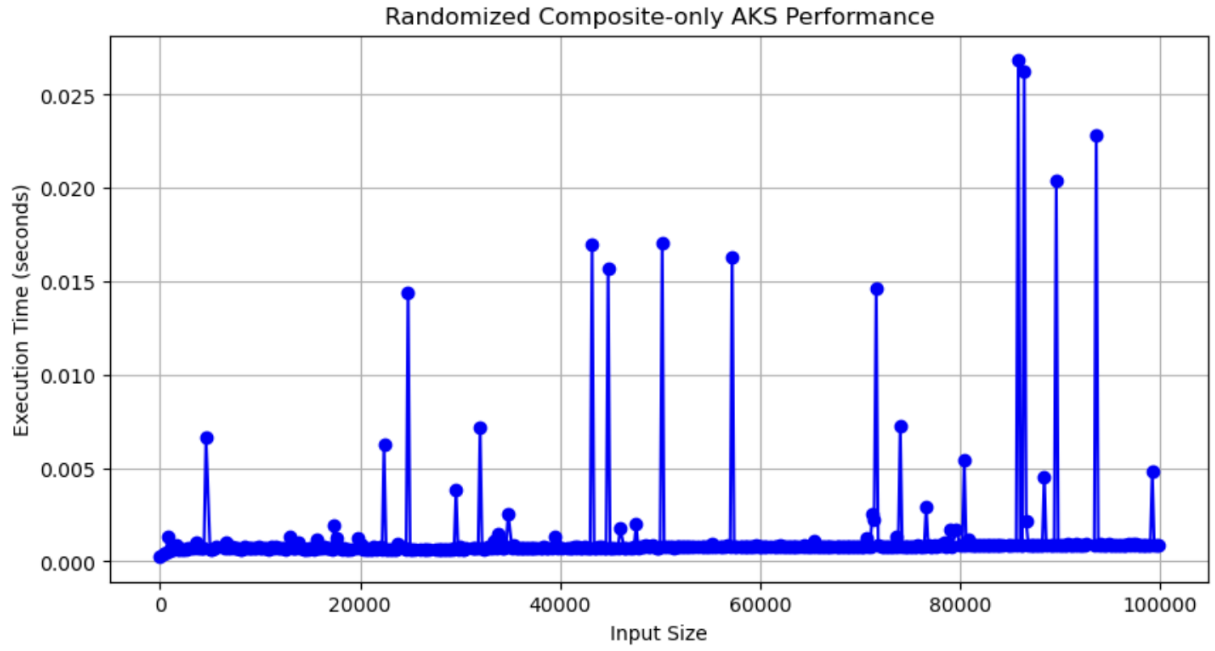

Figure 6: Runtime of Randomized Primality Testing on First 10^5 Integers - Composite Only

References

[1] Benjamin Linowitz. An Exposition of the AKS Polynomial Time Primality Testing Algorithm [master's thesis]. Philadelphia, PA: University of Pennsylvania; 2006.

[2] Lipton, R. J., & Regan, K. W. (2013). *People, problems, and proofs: Essays from Gödel's lost letter: 2010*. Springer-Verlag Berlin Heidelberg.

[3] [aks] M. Agrawal, N. Kayal and N. Saxena, PRIMES is in P, Ann. of Math. (2) 160 (2004), 781-793.

[4] American Institute of Mathematics. (2003). Future directions in algorithmic number theory. Retrieved from https://www.aimath.org/WWN/primesinp/primesinp.pdf