AKS algorithm: a poly-time decider for primality

Yutong Li
University of Southern California
yli81711@usc.edu

April 30, 2022

Abstract

Prime numbers are integers only divisible by 1 and itself. Identifying primes efficiently has been a longstanding open question in complexity theory. The AKS test paper presented a polynomial time algorithm for deciding prime numbers, marking a milestone in the way towards understanding prime numbers [AKS04]. The authors came up with a modified form of Fermat's primality test with reduced runtime. They proved that the performance of a limited number of that test is sufficient to decide whether a integer is prime. With the fact that both the runtime of a single test and the number of tests needed are on the order of polynomial, the total time complexity of the algorithm is polynomial.

1 Introduction

Prime numbers have been a widely studied topic in mathematics. In addition to its theoretical value, primes are useful in practical fields such as cryptography. Therefore, identifying whether a given number is prime with efficiency becomes a fundamental problem, and researchers have been working on it for centuries and keeping pushing the runtime limit of such algorithms.

1.1 PRIMES

Let PRIMES denote the set of all primes. The intuitive approach to determine if a number n is in PRIMES is brute-force: try dividing n by every positive integer smaller than it. We can further reduce the range to integers $\leq \sqrt{n}$ since a factor $p \geq n$ must be accompanied by a factor $q = n/p$ and $q \leq n$.

1.2 Polynomial and pseudo-polynomial

Let's analyze the runtime of the above algorithm. One might state that the solution is polynomial, since the input is n and the algorithm takes $O(\sqrt{n})$ time. However, that's no the case; it only demonstrates the solution is *pseudo-polynomial*. When we talk about runtime, it's determined in terms of the size of the input or the number of bits needed to represent the input rather than the numeric value of the input. Given the input size is $\lceil \log n \rceil$ and runtime is $\sqrt{n}$, the algorithm is actually exponential.

2 AKS algorithm

In a nutshell, the main part of AKS algorithm relies on the following mathematical fact. Let $a \in \mathcal{Z}, n \in \mathcal{N}, (a, n) = 1$, n is a prime iff

$$(X + a)^n = X^n + a \pmod{n} \tag{1}$$

However, even though it's guaranteed that numbers satisfying the equation would definitely be prime, this test would still render exponential runtime complexity because the left-hand side could potentially be a polynomial with n terms. So the authors of the AKS test develop a clever approach that avoid speeding expensive runtime on evaluating all of the n coefficients. By modulo the LHS by a polynomial of form $X^r - 1$ in which r is an appropriately selected small integer, they effectively limit the degree of the LHS polynomial (equivalently the number of coefficients we need to evaluate) to r.

$$(X + a)^n = X^n + a \pmod{X^r - 1, n} \tag{2}$$

Now the issue is, satisfying one equation for a particular a no longer implies primality. Nevertheless, the authors show that with a properly chosen r, only a limited number of a's needs to be tested to determine the primality of n. And such r and a's are polynomial in terms of input size, i.e. $(O \log n)$. All of these work together to limit both the time complexity of each test and the total number of tests that need to be performed, making the total runtime polynomial.

Algorithm 1 AKS algorithm for primality test

```
 1: Input: integer $n > 1$
 2: if $n = a^b$ for $a \in \mathcal{N}$ and $b > 1$ then                    (1)
 3:     return COMPOSITE
 4: Let $r$ = the smallest integer s.t. $o_r(n) > \log^2 n$.                (2)
 5: if there exists $a \le r$, $1 < (a, n) < n$ then                         (3)
 6:     return COMPOSITE
 7: if $n \le r$ then                                                        (4)
 8:     return PRIME
 9: for $a = 1$ to $\lfloor \sqrt{\phi(r)} \log n \rfloor$ do               (5)
10:     if $(X + a)^n \neq X^n + a \pmod{X^r - 1, n}$ then
11:         return COMPOSITE
12: return PRIME                                                             (6)
```

3 Correctness of AKS test

The correctness proof in the AKS paper constitutes the hardcore number theory part of that paper. Due to hardness of that part, this paper will mainly explore the runtime analysis part while only briefly going over the overall structure and rationale of the correctness proof.

To prove the algorithm correctly identifies prime numbers, we need to show that it will output PRIME if the input integer is prime, and its converse, the input is indeed prime if the algorithm outputs PRIME. It's straightforward to prove the first direction. If the input is prime, step 1 and 3 can never return COMPOSITE. Step 5 also can't return COMPOSITE because the equation in step 5 holds true for every prime number. Therefore, the algorithm must return PRIME in step 4 or 6.

But it's hard to show its converse, and the AKS paper devotes most of space on proving it. The general line of logic is as follows. We let p be a prime factor of n. Then we define $\mathcal{G}$ to be the set of polynomials $f \in \mathbb{Z}_p[X] \pmod{h(X)}$ satisfying

$$f(X^n) = f(X)^n \pmod{X^r - 1, n} \tag{3}$$

where $h(X)$ is some irreducible factor of $X^r - 1$. If n is prime, $\mathcal{G}$ would simply be the set of all polynomials in $\mathbb{Z}_p[X] \pmod{h(X)}$. Since now n is acting like a prime by passing all the tests in step 5, we know that all polynomials $(X + a)$ for $a = 1, 2, ..., \lfloor \sqrt{\phi(r)} \log n \rfloor$ are in $\mathcal{G}$, which implies $\mathcal{G}$ is large. On the other hand, we show that $\mathcal{G}$ can't be too large [TS]. More specifically, we show that (1) $|\mathcal{G}| \geq n^{\sqrt{t}}$, and (2) $|\mathcal{G}| \leq n^{\sqrt{t}}$ if n is not a power of p, where t is some fixed value. This implies that n is a power of p, i.e. $n = p^k$ for some $k \geq 1$. But if $k > 1$, the first step of the algorithm should have returned COMPOSITE. Therefore, $k = 1$ and $n = p$. Given that p is a prime factor of n, we can conclude that n is prime.

We also need *Lemma 4.3* from the original paper for the next section: r is upper-bounded by $\lceil \log^5 n \rceil$.

4 Runtime analysis

We need some background knowledge that is not covered in class to better analyze the algorithm.

4.1 Background knowledge

First and foremost, we need to clarify the meaning of a notation extensively used in the AKS paper, $\boldsymbol{O^{\sim}}$ (pronounced "soft Oh"), which is defined as

$$O^{\sim}(t(n)) = O(t(n) \cdot \text{poly}(\log t(n))) \tag{4}$$

To put this in simpler language, the extra tilde means "nearly". For example, $O^{\sim}(n) = O(n \cdot \text{poly}(\log n))$ denotes nearly linearly upper-bounded. More precisely, compared with the corresponding Big O expression, the adding of tilde allows for a leeway and ignores the logarithmic factors, which ensures the overall runtime is still approximately the same as $t(n)$.

Besides, we need to know the runtimes for some basic operations to be able to analyze the complexity of AKS algorithm. The original paper omits most of the proofs and extensively uses results from *Modern Computer Algebra* [vzGG13]. All results needed to comprehend the AKS algorithm are listed below and the full proofs are not included due to space constraints.

- Addition, multiplication, and division operations between two n bits numbers can be performed in time $O^{\sim}(n)$.

  It's quite surprising that the runtime for multiplication and division is nearly linear. As most people anticipated, naive multiplication operation takes quadratic runtime: $O(n^2)$. But it turns out later computer scientists and mathematicians developed some algorithms with incredibly improved time complexity using techniques such as Fast Fourier Transform. With Karatsuba's algorithm, the runtime was reduced to $O(n^{\log 3})$. In 1971, Schönhage and Strassen came up with a nearly linear runtime, in the form of $O(n \log n \log \log n)$, or $O^{\sim}(n)$.

- Likewise, addition, multiplication, and division operations on two polynomials upper-bounded by degree d with coefficients at most m bits in size can be done in time $O^{\sim}(d \cdot m)$ steps.

- Modulo between two polynomials of degrees upper-bounded by d with coefficients at most m bits can be computed in $O^{\sim}(d \cdot m)$ since we can get the remainder as a byproduct of computing their division.

- Modular multiplication between three polynomials of degrees upper-bounded by d with coefficients at most m bits, i.e $f(X)g(X) \mod h(X)$, can be computed in $O^{\sim}(d \cdot m)$. This can be achieved by performing a polynomial multiplication followed by a polynomial modulo.

- The gcd of two n bits numbers can be computed in $O^{\sim}(n)$.

4.2 Step-by-step analysis

Equipped with everything we need, let's analyze the runtime of the algorithm step by step.

The first step is actually the *Exercise 9.44* question from *Modern Computer Algebra*. Though the solution is not covered in the book, the basic idea is to try out all possible b's and for each of them check if n is the bth power of an integer. We can easily see that b is upper-bounded by $\log n$. According to *Theorem 9.28* from the book, each single integer roots computation takes $O(M(\log n))$. The notation $M(n)$ is defined in the book as the runtime of a multiplication operation between two length n integers. What we need to know is that when we talk about $M(n)$, it's algorithm specific. So for the classical multiplication, $M(n) = O(n^2)$. But for some fast multiplication algorithms we mentioned above, $M(n)$ can be smaller. For instance, Schönhage and Strassen's multiplication algorithm corresponds to a $M(n)$ value of $O^{\sim}(n)$.

Given that the paper claims the total runtime of the first step is $O^{\sim}(\log^3 n)$. we have strong reason to suspect that the authors used the classical multiplication: $O^{\sim}(\log^3 n) = O(\log n) \cdot O(\log^2 n)$. If Schönhage and Strassen's algorithm was used, the time complexity would be improved to $O(\log n) \cdot O^{\sim}(\log n) = O^{\sim}(\log^2 n)$.

Out of curiosity, after further research I found out a paper titled *Detecting perfect powers in essentially linear time* that provides even more promising runtime [Ber98]. It proves that there exists a perfect-power classification algorithm that used time $O((\log n)^{1+o(1)}) = O^{\sim}(\log n)$.

In step 2, we find the smallest r such that $o_r(n) > \log^2 n$. By *Lemma 4.3* from the paper we know that only $O(\log^5 n)$ different r's need to be tried. Therefore we can find r by trying out integers up to $\lceil \log^5 n \rceil$ and testing if $n^k \neq 1 \pmod r$ for every $k \leq \log^2 n$. For a particular r, this will involve at most $O(\log^2 n)$ multiplications modulo r, each of which takes $O^{\sim}(\log r)$, summing up to $O^{\sim}(\log^2 n \log r)$. Considering that there are at most $O(\log^5 n)$ r's, the total time complexity of this step is $O^{\sim}(\log^7 n)$.

The third step involves computing gcd. We need to compute $\gcd(a, n)$ for all positive integers $a \leq r$. Since each gcd computation takes $O^{\sim}(\log n)$, the total time of this step is $O^{\sim}(r \log n) = O^{\sim}(\log^6 n)$.

Step 4 requires a simple integer comparison, which takes $O(\log n)$.

Step 5 is the main part of the AKS algorithm and has the dominating runtime. It requires the verification of $\lfloor \sqrt{\phi(r)} \log n \rfloor$ equations. We use fast exponentiation algorithm to evaluate each equation. Fast exponentiation generally adopts the technique of repeated squaring to reduce the number of required modulcar multiplication from linear in terms of exponent to logarithmic.

For instance, suppose $n = 17, a = 3$. We go through step 2 and figure out $r = 7$. Therefore, we want to test the following:

$$(X+3)^{17} = X^{17} + 3 \mod X^7 - 1, 17$$

Now apply repeated squaring:

$$\begin{aligned}(X+3)^{17} &= (X+3)(X+3)^{16} \mod X^7-1, 17\\ &= (X+3)((X+3)^8)^2 \mod X^7-1, 17\\ &= (X+3)(14X^6+16X^5+9X^4+8X^3+12X^2+4X+6)^2 \mod X^7-1, 17\\ &= (X+3)(15X^6+6X^5+16X^4+3X^3+9X^2+7X+13) \mod X^7-1, 17\\ &= X^{17}+3 \mod X^7-1, 17\end{aligned}$$

Since this part might not feel so natural, here we provide the general algorithm [Smi03].

Algorithm 2 Evaluation of an single equation in Step 5

```
1:  **Input:** integer $n, r, a$
2:  **Output:** all coefficients of the polynomial $(X+a)^n$ (mod $X^r - 1, n$)
3:  $f(X) = 1;\ g(X) = X + a;\ y = n$
4:  while $y \neq 0$ do
5:      if $y$ is even then
6:          $g(X) = g(X)^2$ (mod $X^r - 1, n$);
7:          $y = y/2$
8:      else
9:          $f(X) = f(X)g(X)$ (mod $X^r - 1, n$);
10:         $y = y - 1$
11: return $f(X)$
```

Therefore, the evaluation of one equation requires $O(\log n)$ modular multiplication. According to the prior subsection, such modular multiplication takes $O^{\sim}(r \cdot \log n)$. To make this clear, here r corresponds to the upper bound for degree and $\log n$ corresponds to maximum coefficient length. Given this, the evaluation of each equation takes $O^{\sim}(r \cdot \log n \cdot \log n) = O^{\sim}(r \cdot \log^2 n)$. Thus the time complexity of step 5 is $O^{\sim}(r \cdot \log^2 n \cdot \sqrt{\phi(r)} \log n) = O^{\sim}(r^{3/2} \log^3 n)$, since $\phi(r) < r$. Plugging in the upper bound for r, it becomes $O^{\sim}(\log^{21/2} n)$.

Summing up all five steps, the total time complexity for the AKS test is $O^{\sim}(\log^{21/2} n)$, which is in P.

5 Summary of importance

The AKS algorithm is of great theoretical significance, even though it doesn't offer a promising runtime in practice. Actually some randomized algorithms preceding it exhibit better runtime on real computers, approximately on the order of $O^{\sim}(\log^3 n)$ [Aar03]. Those algorithms are randomized, meaning there is a very low probability of error, or there's no chance of mistake but the algorithm might take a very long time to run [Sti08]. AKS test is the first deterministic, rigorous, polynomial-time primality testing method. It eliminates the little probability of error that mathematicians had tried for many years to make it go away.

6 Related readings

For those who are unfamiliar with primes and primality test, I recommend Aaronson's note *The Prime Facts: From Euclid to AKS* as the intro reading [Aar03]. It starts off with introducing primes and its background knowledge, and then naturally proceeds to talking about primality testing and its history, and finally provides a little scratch of the core idea of the AKS test without mentioning too much difficult math.

For commonly asked questions and relevant information about AKS test, such as its implementation, the definition of the class P and NP, prior primality testing algorithms, etc., please look at Anton Stiglic's *The PRIMES is in P little FAQ* [Sti08].

For more comprehensive discussion of the full proof, I recommend referring to Michiel Smid's thorough proof *Primality testing in polynomial time* [Smi03] and Amnon Ta-Shma's lecture note [TS].

References

[Aar03] Scott Aaronson. The Prime Facts: from Euclid to AKS. *Mathematics of Computation*, 2003.

[AKS04] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 2004.

[Ber98] D. Berstein. Detecting perfect powers in essentially linear time. *Mathematics of Computation*, 1998.

[Smi03] Michiel Smid. Primality testing in polynomial time. 2003.

[Sti08] Anton Stiglic. The PRIMES is in P little FAQ. 2008.

[TS] Amnon Ta-Shma. The AKS Algorithm 1 Primality Testing.

[vzGG13] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2013.