WXML Final Report: AKS Primality Test

Amos Turchet, Travis Scholl, Rohan Hiatt, Daria Mićović,
Blanca Viña Patiño, Bryan Tun Pey Quah

Winter 2017

1 Introduction

Prime numbers are fascinating objects in mathematics, fundamental to number theory and cryptography. A primality test takes an integer as input and outputs whether that number is prime or composite. Most practical applications require primality tests to be efficient. Today, the largest known prime number has over twenty million digits so proving that a number of this size is prime can be computationally expensive. In 2002, Agrawal, Kayal, and Saxena published the first deterministic primality test that also runs in polynomial time relative to the binary representation of the input. Although their algorithm represents an important breakthrough in the field of computational number theory, it is seldom used in practice. Our objective was to determine why this idealized algorithm is not practical enough compared to other primality tests. We have re-created the algorithm and optimized our initial naïve implementation by studying each step to achieve optimal complexity using Fermat's Little Theorem, modular arithmetic, and the Fast Fourier Transform for multiplicative efficiency. To confirm that probabilistic methods are still preferred, we compared execution times and accuracy of other tests to our own results.

1.1 The initial problem

Agrawal, Kayal, and Saxena published the first deterministic primality algorithm that runs in polynomial time. Prior to their discovery, most tests used are probabilistic and do not guarantee that any given prime number is in fact

prime. For example, Fermat's Little Theorem shows that for any integer a, a given number n, will satisfy the following congruence if n is prime:

$$a^n \equiv a \pmod{n}$$

However, this fails to filter pseudo-primes, called Carmichael Numbers, which satisfy the congruence even though they are composite integers. Some of the first Carmichael numbers include $561, 1105, 1729$. However, this was a property that was thought only prime numbers could satisfy, so Fermat's Little Theorem is not the most accurate method for identifying primes. Furthermore, deterministic primality tests such as trial by division simply take too long. Trial by division takes $O(2^n)$ in the worst case relative to the binary representation of the input. This time complexity is exponential, so trial by division is not an efficient primality test. After studying these two algorithms, we began looking at AKS. Even though AKS is deterministic, meaning that it correctly identifies all integers as either prime or composite, and runs in polynomial time, it is still rarely used in industry.

1.2 New directions

When testing huge crypto-size primes, polynomial time is still too slow. Mathematicians and cryptographers prefer other methods that may not be as accurate as AKS to save time when testing large numbers. The main reason we studied the AKS primality test was to understand every step of the algorithm to figure out where it takes a long time and why. The overall runtime of our algorithm is $O\tilde{}(\log^{\frac{21}{2}}(n))$ where most of the toll is taken at the step where we use polynomials with modular arithmetic (*polynom_mod*() in Figure 1).

2 Progress

2.1 Computational

The first step is to check if the number is a perfect power. This filters out Carmichael numbers that would pass the polynomial mod step that implements Fermat's. Then we find the smallest r that satisfies the conditions explained in the theoretical section. Then we test as and return composite if a and n are not coprime. If $n \leq r$ then we return prime. Once all of these

steps have been completed we go into the polynomial mod function which will be explained in the theoretical section.

```
def aks(n):
    pow = is_perf_pow2(n)
    if pow == "composite":
        return pow
    r = smallest_r(n)
    for a in range(2, min(r, (n/2) + 1)):
        if gcd(a, n) > 1:
            return "composite"
    if n <= r:
        return "prime"
    else:
        p = polynom_mod(r, n)
        return p
```

Figure 1: Sage code implementation of AKS algorithm

To study how well our version of the AKS algorithm performs, we wrote another primality test which utilizes Fermat's Little Theorem to test for primes (see Figure 2).

```
def flt(n):
    used = []
    for i in xrange(floor(log(n)) + 1):
        a = randint(1, n - 1)
        while a in used:
            a = randint(1, n - 1)
        used.append(a)
        if (a^(n) % n) != a:
            return "Composite"
    return "probably prime"
```

Figure 2: Implementation of algorithm using Fermat's Little Theorem (FLT)

In Figure 3 the blue dots are primes identified by AKS, the black dots are primes identified by FLT, the green dots are composites found by FLT, and the red dots are composites found by AKS. We can see that AKS takes longer to identify a prime number but less time to identify a composite number. Figure 4 shows Carmichael Numbers being identified as primes by FLT and as composites by AKS.

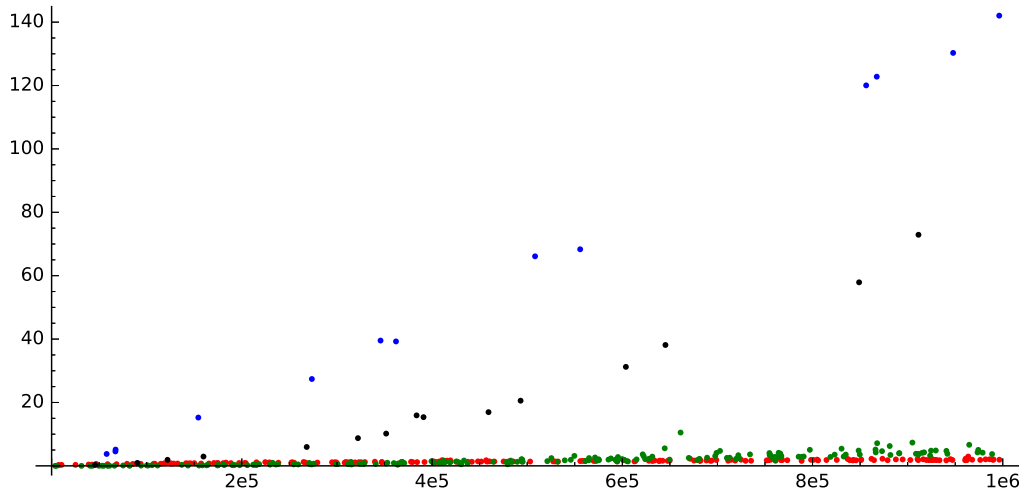

Figure 3: Graph of Number vs Time Taken for Primality Test

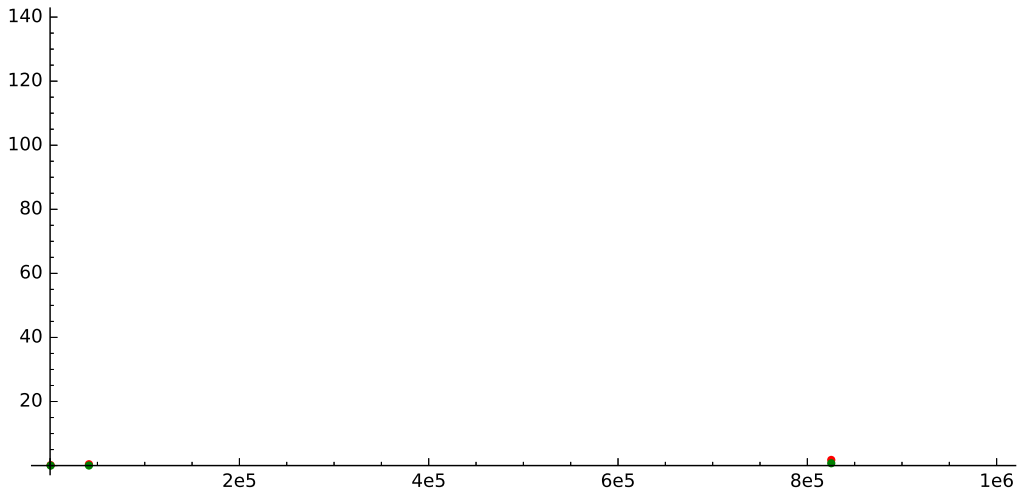

Figure 4: Graph of Number vs Time Taken for Primality Test

2.2 Theoretical

The main theorem that the AKS algorithm implements is a generalization of Fermat's Little Theorem. This generalization says that for $a \in \mathbb{Z}, n \in \mathbb{N}, n \geq 2$, where a and n are coprime then n is prime if and only if

$$(X + a)^n \equiv X^n + a (\text{mod } n)$$

This reduces the coefficients of the polynomials, however, the computation is still long. If we reduce the polynomial $(X + a)^n$ to a smaller degree then less computation on the coefficients is required. Thus the algorithm uses

$$(X + a)^n \equiv X^n + a (\text{mod } X^r - 1, n)$$

In order to efficiently use this as a test for primality, we need to set sufficient bounds on r to both reduce the complexity and to correctly categorize n. The original AKS paper sets a bound with a lemma that states that there exists an $r \leq \max(3, \lceil \log^5(n) \rceil)$ such that the $o_r(n) > \log^2(n)$ and where r and n are coprime. Here, $o_r(n)$ is the order of n modulo r. This is the final step of the algorithm. The numbers that pass this test but are not prime are filtered out in earlier steps of the algorithm. We implement this for our polinomial mod function.

3 Future directions

Initial investigations into the AKS primality algorithm opened up many interesting avenues for future pursuit of research. Our team identified multiple areas in which we could move forward and potentially inherently improve our implementation, or understand more about the algorithm and its functionality.

One of the key next steps we outline involves taking our implementation out of Sage, and using proprietary Python to code the algorithm independently. Currently, Sage has multiple functions and methods which have nebulous time complexities and operations. Coding our algorithm completely from scratch would allow for maximum control over runtime, as well as precise knowledge as to which operations utilize the most resources. More rigorous scrutiny of time complexity would be a boon for any future investigations of AKS. Our analysis would then be significantly more robust and controlled, leading into our next potential direction.
As we have noted throughout our investigations, AKS is currently not an industry standard. Our attempts to understand why this is the case have been functionally trivial at best. Thus, having complete control over runtime analysis would allow our team to proceed with more rigorous tests which compare AKS to the current industry standard primality algorithms. Having the ability to closely examine which specific aspects of AKS run better or worse than similar aspects of other algorithms could even provide insight into potential systemic improvements.

References

[1] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Ann. of Math. (2)*, 160(2):781–793, 2004.

[2] A. Granville. It is easy to determine whether a given integer is prime. *Bull. Amer. Math. Soc. (N.S.)*, 42(1):3–38, 2005.

[3] L. Rempe-Gillen and R. Waldecker. *Primality testing for beginners*, volume 70 of *Student Mathematical Library*. American Mathematical Society, Providence, RI, 2014.

[4] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 6.10)*, 2016. `http://www.sagemath.org`.