1 Background

Prime numbers are of particular interest in cryptography, and in mathematics in general. Primality algorithms – algorithms determining whether or not a given integer has nontrivial factors – have been around since the third century B.C., when Eratosthenes discovered the notion of a prime number sieve. Unfortunately, algorithms like this do not suffice when it comes to testing large primes - the process takes too many steps and is too slow. Since then, more efficient methods have been devised that determine whether or not a given number is prime.

For example, in 1975, Miller designed a deterministic polynomial time primality algorithm, which works under the assumption of the Extended Riemann Hypothesis (ERH) [4]. In 1980, Rabin adjusted this algorithm to work without the assumption of ERH, but in turn made it probabilistic, not deterministic [6]. Other algorithms have since been conceived but suffer from at least one of the following shortcomings: the algorithms have either relied on an unproven assumption, they are probabilistic, or they do not run in polynomial time.

In 2004, Agrawal, Kayal, and Saxena (AKS) invented an algorithm that does not suffer from any of the above shortcomings [2]. Indeed their primality algorithm is deterministic, unconditional, and runs in polynomial time.

The purpose of this paper is to accompany an implementation of this algorithm into C++. The algorithm is described in Section 2, and the mathematics behind the algorithm is briefly outlined in Section 2.1. This paper has no intention of re-hashing the results of [2] so some of the results and proofs are omitted, and the proofs included are only sketched.

In section 3 a link to the C++ file is provided. The file has extensive comments and is considered an integral part of this paper. The reader is encouraged to open the file and read the comments – compiling the code and running the program is encouraged as well. Runtimes and ideas for improving the implementation are included in sections 3.1 and 3.2 respectively.

1.1 Notation

Throughout this paper, the notation of [2] is used. All logarithms are taken in base 2. For any prime number p, F_p denotes the finite field with p elements. Provided n and r are relatively prime, $o_r(n)$ refers to the multiplicative order of n modulo r, that is, the least positive integer k such that $n^k \equiv 1 \pmod{r}$. Euler's totient function is denoted φ. Given polynomials $f(x), g(x)$ and $h(x)$, when we write $f(x) = g(x) \pmod{h(x), n}$, we mean $f(x) = g(x)$ in the polynomial ring $\mathbb{Z}_n[x]/h(x)$. For any algorithm $\mathcal{B}$ that takes input k, $\mathcal{B}(k)$, refers to the output of $\mathcal{B}$ on input k.

2 The algorithm

The AKS algorithm, which we will denote $\mathcal{A}$, consists of six straightforward steps.

Input a positive integer $n > 1$.

1. If $n = a^b$ for $a, b \geq 2$ return `COMPOSITE`.

2. Find the smallest r such that $o_r(n) > \log^2 n$.

3. If $1 < \gcd(a, n) < n$ for any $a < r$ return `COMPOSITE`.

4. If $n \leq r$, return `PRIME`.

5. For $a = 1$ to $\ell = \lfloor \sqrt{\varphi(r)} \log n \rfloor$ do:
   if $(x + a)^n \neq x^n + a \pmod{x^r - 1, n}$, return `COMPOSITE`.

6. Return `PRIME`.

2.1 The mathematics of $\mathcal{A}$

The heart of $\mathcal{A}$ lies in step 5, which is a modified result of [1]:

Lemma 1. *Let $a \in \mathbb{Z}$, $n \in \mathbb{N}$ with $n > 2$ and $\gcd(a, n) = 1$. Then n is prime if and only if*

$$(x + a)^n = x^n + a \pmod{n}.$$

This is a nice characterization of prime numbers, but becomes an infeasible calculation when n gets large: the polynomial $(x + a)^n$ has n coefficients to calculate. This is why [2] introduce a polynomial modulus with a small degree exponent in step 5. Their choice of $(x - 1)^r$ must be chosen carefully so that r is small, while no composite n exists such that $(x + a)^n = x^n + a \pmod{x^r - 1, n}$. They first prove an important bound on the size of r:

Lemma 2. *There exists an $r \leq \max\{3, \lceil log^5 n \rceil\}$ such that $o_r(n) > log^2 n$.*

No matter the choice of r however, it follows directly from Lemma 1 that if n is prime, then $\mathcal{A}$ outputs prime:

Lemma 3. *If n is prime then $\mathcal{A}(n) =$ `PRIME`.*

Proof. Notice that the only possibilities for $\mathcal{A}$ to output `COMPOSITE` are in Steps 1, 3 and 5. If n is prime, then Steps 1 and 3 will certainly not output `COMPOSITE`, and by Lemma 1, Step 5 will not output composite. So $\mathcal{A}$ will output `PRIME` either in Step 4 or 6. □

To prove the converse – if $\mathcal{A}(n) =$ `PRIME` then n is prime – a more sophisticated argument is needed. Removing trivial cases, we assume that $\mathcal{A}(n) =$ `PRIME`, and we have made it to Step 6, and r is the integer returned in Step 2. Furthermore, since $o_r(n) > 1$, there exists a prime p that divides n (in fact, we are trying to prove that $n = p$) with $o_r(p) > 1$. If $p < r$ then Step 3 would detect that p is a divisor of n if $n > r$, and Step 4 would output `PRIME` otherwise. For similar reasons, $\gcd(n, r) = 1$, therefore $p, n \in \mathbb{Z}_r^*$. From this point forward, fix n, p and r.

By our assumptions, for each $0 \leq a \leq \ell$,

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

and therefore

$$(x + a)^n = x^n + a \pmod{x^r - 1, p}. \tag{1}$$

By Lemma 1, we have that

$$(x+a)^p = x^p + a \pmod{x^r - 1, p}. \tag{2}$$

Combining (1) and (2), we get

$$(x+a)^{\frac{n}{p}} = x^{\frac{n}{p}} + a \pmod{x^r - 1, p}.$$

This motivates the following definition.

Definition 1.

For polynomial $f(x)$ and integer m, we say that m is *introspective* for $f(x)$ (modulo $h(x), n$) if

$$(f(x))^m = f(x^m) \pmod{h(x), n}.$$

Since we are dealing almost exclusively with polynomials in the polynomial ring $\mathbb{Z}_n[x]/(x^r - 1)$, we drop the modulus; all uses of the definition are modulo $x^r - 1, n$.

Clearly n and $\frac{n}{p}$ are introspective for $x+a$ for $0 \leq a \leq \ell$. It is not difficult to see that given a polynomial $f(x)$, introspective numbers are closed under multiplication, and conversely, given m and the set of polynomials in which is introspective for, polynomial multiplication preserves the introspectiveness of m. That is, if m and m' are introspective for $f(x)$ then $m \cdot m'$ is introspective for $f(x)$; if m is introspective for $f(x)$ and $g(x)$ then m is introspective for $f(x) \cdot g(x)$.

It follows from these facts that every number in the set

$$I = \left\{ \left(\frac{n}{p}\right)^i \cdot p^j : i, j \geq 0 \right\}$$

is introspective for every polynomial in the set

$$P = \left\{ \prod_{a=0}^{\ell} (x+a)^{e_a} : e_a \geq 0 \right\}.$$

We now define the following two groups. Let

$$G = \{ i \pmod{r} : i \in I \}$$

and let $|G| = t$. Since $\gcd(n, r) = \gcd(p, r) = 1$, it follows that G is a subgroup of $\mathbb{Z}_r^*$. Furthermore, since G is generated by n and p, and by Lemma 2, $t > o_r(n) > \log^2 n$.

To define the second group $\mathcal{G}$, some preliminary steps are necessary. First, let $Q_r(x)$ be the r-th cyclotomic polynomial in $\mathbb{Z}_n/(x^r - 1)$ - that is, the unique polynomial $Q_r(x)$ in $\mathbb{Z}_n/(x^r - 1)$ such that $Q_r(x)$ divides $x^r - 1$, but does not divide $x^k - 1$ for any $0 < k < r$. The existence and uniqueness of such a polynomial is not within the scope or direction of this paper and is left out. Also left out of the paper is proof that $Q_r(x)$ factors into irreducible polynomials of degree $o_r(p) > 1$. Let $h(x)$ be one of these polynomials of degree $o_r(p)$. Finally, let

$$\mathcal{G} = \{ f(x) \pmod{h(x), p} : f(x) \in P \}.$$

The following two lemmas are taken as fact:

Lemma 4. $|\mathcal{G}| \geq \binom{t+\ell}{t-1}$.

Lemma 5. *If n is not a power of p then $|\mathcal{G}| \leq n^{\sqrt{t}}$.*

We are now prepared to prove the final result:

Theorem 1. *Let n be a positive integer. Then $\mathcal{A}(n) =$ `PRIME` if and only if n is prime.*

Proof. The reverse direction is proved in Lemma 3. Recall that $t > \log^2 n$, and that $\varphi(r) \geq t$ and therefore $\ell \geq \lfloor \sqrt{t} \log n \rfloor$. Given that we have made it to Step 6 to return `PRIME`, starting with Lemma 4 consider the size of $|\mathcal{G}|$:

$$\begin{aligned}
|\mathcal{G}| &\geq \binom{t+\ell}{t-1} \\
&\geq \binom{\ell + 1 + \lfloor \sqrt{t} \log n \rfloor}{t \log n} \\
&\geq \binom{2\lfloor \sqrt{t} \log n \rfloor + 1}{t \log n} \\
&\geq 2^{\lfloor \sqrt{t} \log n \rfloor + 1} \\
&\geq n^{\sqrt{t}}.
\end{aligned}$$

By Lemma 5, n is a power of p, that is $n = p^k$ for some $k \geq q$. But if $k > 1$, then Step 1 would have returned `COMPOSITE`. So $k = 1$ and $n = p^1$ is prime. □

3 Implementation

The implementation of $\mathcal{A}$ is in a C++ file called `aks.cpp` and may be found here:

goo.gl/0k6Q7P.

There are extensive comments in `aks.cpp` which describe the reasoning behind each step of the implementation. This file is integral to this paper and it is encouraged that the reader open the file and read the comments.

3.1 Accuracy and run times

A collection of 10 k-bit primes [5] for $8 \leq k \leq 40$ was used against `aks.cpp` on a 1.7 GHz 64-bit machine. The runtimes were recorded and are displayed in Figure 1, with the x-axis representing the number of bits of (x, y), and the y-axis representing the number of seconds it took to verify that (x, y) was prime.

For comparison, a 256-bit prime[1] was tested, and on Step 5, the average calculation of $(x + a)^n$ took approximately five minutes, and the limit of the for-loop, $\ell = \lfloor \sqrt{\varphi(r)} \log n \rfloor$

[1]The 256-bit prime used was

102639592829741105772054196573991675900716567808038066803341933521790711307779

and happens to be one of the factors of RSA-155, from the RSA Factoring Challenge.

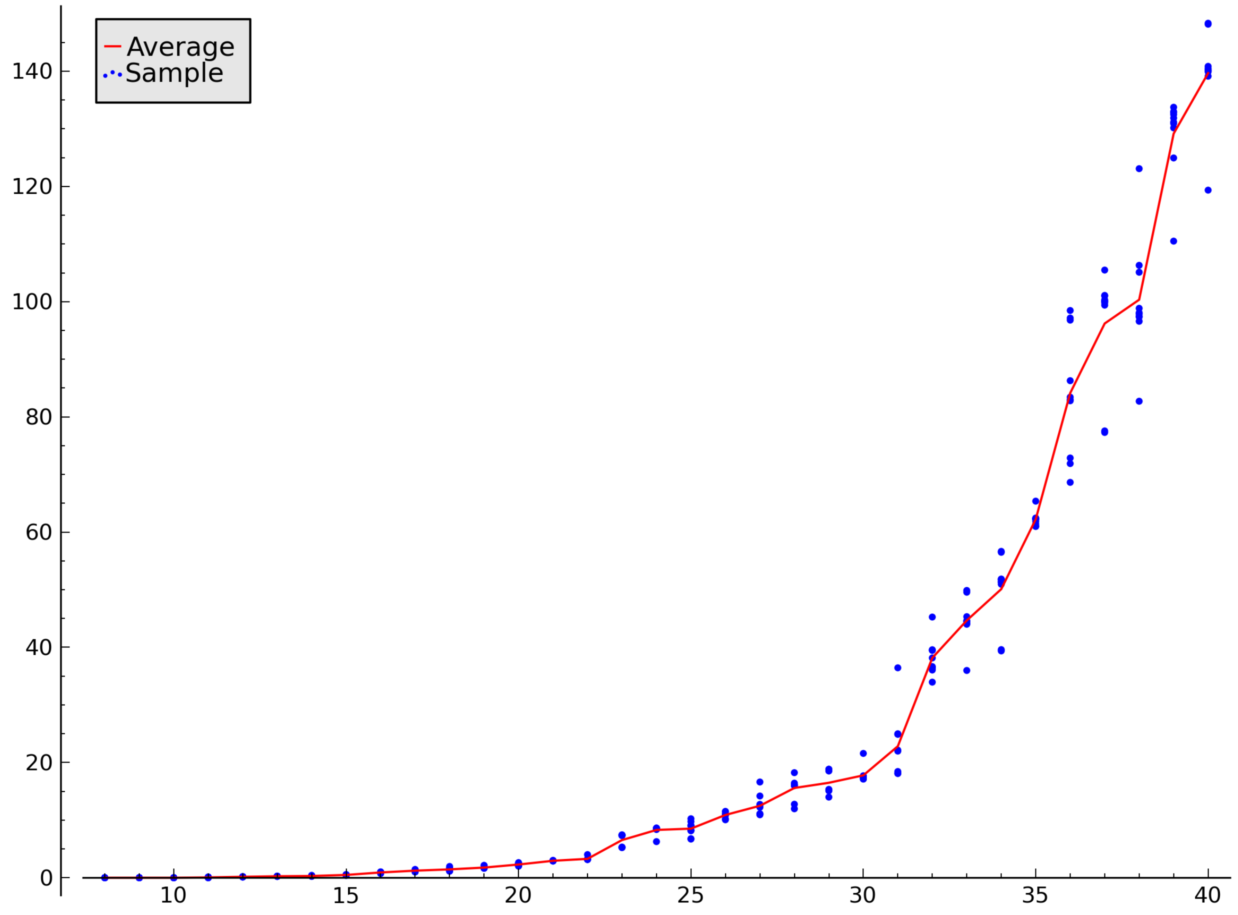

Figure 1: Runtimes (bits vs. seconds)

was equal to 36022, meaning the calculation would have taken a total of approximately 3000 hours, or 125 days to complete – far exceeding the appropriate number of days for the author to ask for an extension.

Looking at Figure 1, and considering the fact that verifying the primality of a 256-bit prime would take half a year, it appears – heuristically – that `aks.cpp` is exponential. This is most likely due to a poor implementation of Step 5. This is discussed in further detail in Section 3.2.

A selection of 100 prime numbers [3] between 30241095667 and 30241098031 were used to test composite numbers: in the collection of these 36-bit primes, subsequent pairs were multiplied together and tested against the algorithm. By construction, each of the 72-bit products had large enough factors to make it to Step 5, and they were all verified as composite. Their average runtime was 2.6204 seconds. The dramatic speed increase is due to the fact that when n is prime, Step 5 needs to verify that $(x+a)^n = x^n + a \pmod{x^r - 1, n}$ for each a up to ℓ. When n is composite, we stumble across some a that makes this equality false, so the algorithm can halt much earlier.

3.2 Shortcomings and possible improvements

Each step of `aks.cpp` is trivially efficient except possibly Euler's totient function and Step 5. Luckily, the former is called upon only to calculate $\varphi(r)$, and r is relatively small compared to n, so this does not pose an issue with runtimes.

The main issue is in Step 5 of `aks.cpp`. The for-loop in Step 5 must make exactly ℓ calculations to verify that a prime number is indeed prime. This number is quite small, and calculating it is not an issue. Rather, computing $(x + a)^n$ is the culprit: `aks.cpp` uses the `PowerMod` function from the NTL library to calculate $(x + a)^n$. This may not be the most efficient way to calculate such exponentiation. According to the documentation of NTL [7], `PowerMod` uses Fast-Fourier-Transform and the Chinese Remainder Theorem to exponentiate elements in the polynomial ring. Perhaps a better implementation could do this more efficiently. This, however, is beyond the scope of the author's capabilities as a programmer.

References

[1] Agrawal and Biswas, "Primality and identity testing via Chinese remaindering." *J. of the ACM*, 50:429443, 2003.

[2] Agrawal, Kayal, and Saxena, "Primes is in P." *Ann. Math.* 160, 781-793, 2004. `http://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf`.

[3] "Big Primes: large list of prime numbers." `http://www.bigprimes.net/`

[4] Miller, "Riemanns hypothesis and tests for primality." *J. Comput. Sys. Sci.*, 13:300317, 1976.

[5] Caldwell, "Primes just less than a power of two 8 to 100 bits (page 1 of 4)." *The Prime Pages: prime number research, records and resources.* `http://primes.utm.edu/lists/2small/0bit.html`

[6] Rabin, "Probabilistic algorithm for testing primality." *J. Number Theory*, 12:128138, 1980.

[7] Shoup, "MODULE: ZZ_pX" `http://www.shoup.net/ntl/doc/ZZ_pX.txt`