Three Primality Tests and Maple Implementation

by

Renee M. Canfield

(Under the direction of Robert Rumely)

Abstract

This paper discusses three well known primality tests: the Solovay-Strassen probabilistic test, the Miller test based on the ERH, and the AKS deterministic test. Details for the proofs of correctness are given. In addition, Maple code has been written to implement the tests and to count the number of steps executed for numbers of various sizes. Analysis of steps counted between the three tests is given along with least squares fitting of the data.

Index words: Primality Test, Miller, Monte-Carlo, AKS, ERH

Three Primality Tests and Maple Implementation

by

Renee M. Canfield

B.S., Kent State University, 2006

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Arts

Athens, Georgia

2008

Three Primality Tests and Maple Implementation

by

Renee M. Canfield

Approved:

Major Professor: Robert Rumely

Committee: Leonard Chastkofsky
Dino Lorenzini

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2008

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The interest in primality testing has grown rapidly in the past two decades. This is due largely to the introduction of public-key cryptography which is used for encryption of electronic correspondence. The security of this type of cyrptography relies on the difficulty involved in factoring very large numbers which in turn requires knowledge of whether these large numbers are prime or composite to begin with.

There are two types of primality tests: deterministic and probabilistic. Deterministic tests determine with absolute certainty whether a number is prime while the latter can possibly identify a composite number as prime, but not vice versa. If a number passes a probabilistic primality test, it is only referred to as probably prime. If it is actually composite, then it is said to be a pseudoprime. The most common pseudoprime is a Fermat pseudoprime which satisfies Fermat's Little Theorem.

The search for a good primality test may very well be one of the oldest issues in mathematics. One of the simplest and well known is the Seive of Eratosthenes. Eratosthenes, a Greek mathematician who lived circa 200 B.C., developed a primality test based on the fact that if a number n is composite, then all of its factors must be $\leq \sqrt{n}$. First make a list of all integers $2, 3, \ldots, m$ where $m \leq \sqrt{n}$. Then circle 2 and cross off all the multiples of two on the list. Then circle 3 and cross off its multiples. Continue this process, each time advancing to the least integer that is not crossed off, circling that integer, and crossing off its multiples. Then test to see if any of the circled numbers divide n. If the list of circled numbers is exhausted and no divisor is found, then n must be prime. This algorithm is fairly straightforward and easy to implement, but is by no means efficient. If we were to use it on a number with only 20 digits, we would need to first find all the primes up to 10^{10},which is about 450 million numbers. At the rate of finding one prime per second, we would be working for a little over 14 years, even before dividing them into our 20 digit number (McGregor-Dorsey [8]).

A property that almost gives an efficient test is Fermat's Little Theorem: for any prime p and any $a,\ p \nmid a$, we have $a^{p-1} \equiv 1 \pmod{p}$. So given a pair (a, n), we can check this equivalence using repeated squaring. What keeps this from being a correct primality test is that many composite numbers n called *Carmichael numbers* satisfy the Fermat congruence. Nevertheless, Fermat's Little Theorem still became the basis for many efficient primality tests.

The first test mentioned in this paper was developed by Solovay and Strassen in 1974. It is a randomized (hence the name Monte-Carlo) polynomial-time algorithm using the property that for a prime number n, $\left(\frac{a}{n}\right) \equiv a^{\frac{n-1}{2}} \pmod{n}$ for every a where $\left(\frac{a}{n}\right)$ is the Jacobi symbol.

The second test described in this paper was developed in 1975 by Gary Miller. It uses a property based on Fermat's Little Theorem to obtain a deterministic polynomial-time algorithm using the *Extended Riemann Hypothesis* (ERH). Soon afterwards, his test was modified by Rabin to yield an unconditional but randomized polynomial-time algorithm.

In 1983, Adleman, Pomerance and Rumely achieved a breakthrough by creating a deterministic algorithm for primality that runs in $(\log n)^{\mathcal{O}(\log \log \log n)}$ time, whereas all the previous tests of this type ran in exponential time. And in 1986, Goldwasser and Kilian proposed a randomized algorithm based on elliptic curves running in expected polynomial-time on almost all inputs. Adleman and Huang modified their algorithm to obtain a similar algorithm that runs in expected polynomial-time on all inputs.

The overall goal in finding a desirable primality test leads to an unconditional deterministic polynomial-time algorithm, the final test discussed this paper. Agrawal, Kayal and Saxena, three mathematicians from India, created an algorithm in 2002 that runs in $\mathcal{O}\tilde{\ }(\log^{\frac{21}{2}} n)$ time. Their test relies on the fact that n is prime iff $(X + a)^n \equiv X^n + a \pmod{n}$. To keep this efficient, they reduced the number of coefficients to compute on the left side of the congruence by reducing both sides modulo a polynomial of the form $X^r - 1$ for an appropriately chosen r. Some composite $n's$ may satisfy the equivalence for a few values of a and r. Thus they also showed for the well chosen r, if the equivalence is satisfied for an appropriate number of $a's$, then n must be a prime power. Adding a binary search for prime powers, we conclude that n must be prime. Because the number of $a's$ and the value of r are both bounded by a polynomial in $\log n$, they gave us a deterministic polynomial-time algorithm for testing primality.

This paper is a synopsis of three different papers: *A Fast Monte-Carlo Test for Primality* by Solovay and Strassen [12], *Riemann's Hypothesis and Tests for Primality* by Gary Miller [9] and *Primes is in P* by Agrawal, Kayal and Saxena [2]. In that order, I read each paper and filled in missing details to the proofs presented in the papers.

Along with the work done reading the papers and understanding proofs, I began to implement the primality tests in Maple. I wrote my own code following the written algorithms in the papers. It was sometimes difficult to write nested loops because the details cannot be found in the papers themselves. I studied multiple resources to get familiar with creating code. Once the code was working, i.e. correctly declared whether a number was prime or composite, I started to break down the code even further. I wrote multiple subroutines, sometimes borrowing suggestions from sources like Dietzfelbinger [6]. To expand upon the analysis of the tests, I used Maple to count the steps executed among various sizes and types of inputs for n. Breaking down subroutines added to the accuracy of this step counting. Some computations were left to the Maple but most were broken down. The code and explanations of step counting can be found in the Appendices A and B.

Overall, the Miller test was hardest to implement in Maple because of the nested loops. The Monte-Carlo test was the easiest to code, and I added some additional code to try to catch Carmichael numbers before the congruence Solovay and Strassen suggested as the basis of their test. Other than that, I adhered to the algorithms given. A larger computer with more memory would have been helpful for my calculations because even a number of size 10^7 sometimes took up to 15 hours on my home computer in the AKS test. This is a note to anyone who might try calculations on their own with my code.

As a student with no experience in statistics, I added some amateur least squares fitting analysis to my data recorded to see if I could find any linear relationships. Much to my delight, the relationship explained in Section 5.3 between the number tested for primality n and the steps counted in the algorithms was there. It would be interesting to see if a quadratic relationship is present or perhaps some other nonlinear model fit if someone were to continue my work. In order to do this with my code, the problem with my AKS test of the numbers being too large in context would have to be corrected. The analysis is ready to work with larger numbers in the Monte-Carlo and Miller tests.

Chapter 2

A Fast Monte-Carlo Test for Primality

2.1 Introduction

Let n be an odd integer. Our first test, called a Monte-Carlo test because of the random sampling of the variable a from the set $\{2, \ldots, n-1\}$, is based on the modular equivalence of the residue $e := a^{\frac{n-1}{2}} \pmod{n}$ where $-1 \leq e \leq n-2$ and the Jacobi symbol $j := (\frac{a}{n}) \pmod{n}$ for a and n relatively prime (Solovay, Strassen [12]). Euler proved that if n is an odd prime and $a \in \mathbb{Z}$ then $e \equiv j \pmod{n}$. For a given a, there is $\geq 1/2$ chance that a is a witness to the compositeness of n and $< 1/2$ chance that n will falsely pass the test as a composite number posing as a prime. So if the congruence holds for $\lfloor \log_2 n \rfloor$ choices of a, then we can reasonably assume n is probably prime. The chance of n falsely passing the test is $< 1/n$ because the probability of the algorithm failing is 2^{-k}, where k is the number of $a's$ tested. If at any time we find a nontrivial $gcd(a, n)$ or $e \neq j \pmod{n}$ then n is composite. This test was the easiest of all three tests to implement in Maple mostly because of its comparative length. The only difficulty was making sure the random a did not duplicate itself for the smaller $n's$ tested. This was remedied using the intersection and union of sets. The cost of this procedure is $\mathcal{O}(\log^3 n)$ binary operations or $6 \log n$ muliprecision operations per value of a.

2.2 Notation.

In this paper we assume the length of n is $\log_2 n$ and we denote this by $\log n$ omitting the subscript.

Definition 2.2.1. We say *an algorithm tests primality in $\mathcal{O}(f(n))$ steps* if there exists a deterministic Turing machine (assuming a bit model for arithmetic) which implements this algorithm, and this machine correctly indicates whether n is prime or composite in less than $C \cdot f(n)$ steps, for some constant C.

We know the Jacobi symbol is a generalization of the Legendre symbol $\left(\frac{a}{p}\right)$ for $p \geq 3$ prime and any integer a. This test by Solovay and Strassen uses the fact that $a^{\frac{p-1}{2}} \equiv \left(\frac{a}{p}\right) \pmod{p}$. Euler proved this in the lemma below by showing $a^{\frac{p-1}{2}}$ equals 1 if a is a quadratic residue modulo p and -1 if a is a nonresidue modulo p matching the definition for the Legendre symbol $\left(\frac{a}{p}\right)$.

Lemma 2.2.2. *(Wojciechowski [15]) Let p be an odd prime number and a an integer such that $gcd(a,p) = 1$. Then:*

$$a^{\frac{p-1}{2}} = \begin{cases} 1, & \text{if } a \text{ is a quadratic residue modulo } p, \\ -1, & \text{if } a \text{ is a nonresidue modulo } p. \end{cases} \tag{2.1}$$

Proof. Let $x = a^{\frac{p-1}{2}}$. Then $x^2 \equiv a^{p-1} \equiv 1 \pmod{p}$ by Fermat's Little Theorem, so $x = \pm 1$. Suppose a is a quadratic residue so there exists a b such that $b^2 \equiv a \pmod{p}$. Then we have

$$x \equiv a^{\frac{p-1}{2}} \equiv (b^2)^{\frac{p-1}{2}} \equiv b^{p-1} \equiv 1 \pmod{p}$$

again using Fermat's Little Theorem at the end.

Now suppose a is a nonresidue modulo p. Since there are at most $\frac{p-1}{2}$ roots of the equivalence $z^{\frac{p-1}{2}} \pmod{p}$ and there are $\frac{p-1}{2}$ quadratic residues modulo p (because p is an odd prime), the only roots of the equivalence are the quadratic residues modulo p. Since a is not one of those, and $a^{\frac{p-1}{2}} \equiv \pm 1 \pmod{p}$, x must be equal to -1 modulo p. □

2.3 Error probability.

We now investigate the correctness of this algorithm (Solovay, Strassen [12]).

Lemma 2.3.1. *If n is composite at most $\frac{1}{2}$ of the numbers from 1 to $n-1$ will lead to the procedure incorrectly concluding n is prime.*

Proof. If n is prime the procedure will reach a correct decision so assume n is composite. Let

$$G = \{a + (n) : a \in \mathbb{Z} \;\&\; gcd(a,n) = 1 \;\&\; a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right) \pmod{n}\}$$

be a subgroup of $\mathbb{Z}_n^*$.

If $G \neq \mathbb{Z}_n^*$ then because the order of a subgroup divides the order of the group, the order of G will be at most $\frac{n-1}{2}$ so at most $\frac{1}{2}$ of the numbers from 1 to $n-1$ will lead to the procedure concluding n is prime.

Let us assume that the congruence holds and

$$a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right) \pmod{n} \tag{2.2}$$

for all $a \in \mathbb{Z}$ relatively prime to n. If $n = p^e$ where p is prime, then from (2.2) we get

$$a^{p^e - 1} \equiv 1 \pmod{p^e}$$

as long as a is not divisible by p. Because $\mathbb{Z}_{p^e}^*$ is cyclic of order $\phi(n) = p^{e-1}(p-1)$ we get

$$p^{e-1}(p-1) | p^e - 1$$

which implies $e \leq 1$. This cannot happen because n is composite so n is not a power of a prime and must look like $n = rs$ with $gcd(r,s) = 1$. Let us first suppose that n is square free. Equation (2.2) implies that

$$a^{\frac{n-1}{2}} \equiv \pm 1 \pmod{n} \tag{2.3}$$

for a such that $gcd(a,n) = 1$. We will prove that in fact

$$a^{\frac{n-1}{2}} \equiv 1 \pmod{n} \tag{2.4}$$

for a relatively prime to n.

Assume the opposite and there is an a such that $a^{\frac{n-1}{2}} \equiv -1 \pmod{n}$. Since $gcd(r,s) = 1$ the Chinese remainder theorem says we can find a b such that $b \equiv 1 \pmod{r}$ and $b \equiv a \pmod{s}$. Raising both sides of the congruences to the power $\frac{n-1}{2}$ we get

$$b^{\frac{n-1}{2}} \equiv 1 \pmod{r}, \quad b^{\frac{n-1}{2}} \equiv -1 \pmod{s}.$$

This contradicts (2.3) again by the Chinese remainder theorem. Thus (2.4) must be true and because we assumed (2.2), these two equations imply that

$$\left(\frac{a}{n}\right) \equiv 1$$

for all a relatively prime to n. This is impossible because n is a square free composite.

Now suppose that n is not square free and say $n = p^e q$ where p is an odd prime, $e > 1$, and q is relatively prime to p. It follows from (2.2) that $a^{n-1} \equiv 1 \pmod{n}$ for all a such that $gcd(a, n) = 1$. By the Chinese remainder theorem, $a^{n-1} \equiv 1 \pmod{p^e}$ for all a such that $gcd(a, p) = 1$. Then these two congruences imply that $p^{e-1}(p-1)|n-1$ while $e > 1$ because the order of $\mathbb{Z}_{p^e}^*$ is $\phi(p^e)$. But, $p|n$ and $p^{e-1}(p-1)|n-1$ cannot both be true. Thus, (2.2) is not true for all $a \in \mathbb{Z}$ relatively prime to n so $|G| \leq \frac{n-1}{2}$ (Solovay, Strassen [13]). □

2.4 Running Time

For this test, we not only talk about bit operations for running time but also multiprecision operations, meaning an arithmetic operation or a divison with remainder of two numbers $< n^2$ (Knuth [7]). We first must compute $gcd(a, n)$ using the Euclidean algorithm. To do this efficiently, we write $d := a$ and n in binary. From here, we run the Euclidean algorithm to find the $gcd(d, n)$. We want to first write $n = q_1 d + d_1$, but instead of doing a division to find q_1, d_1 we perform the subtraction $n - 2^{c_1} d$ where multiplying d by 2^{c_1} allows us to 'line up' our subtraction. We continue to subtract $2^{c_j} d, j \geq 1$ and eventually we get $d_1 < d$ and $q_1 = 2^{c_1} + 2^{c_2} + \ldots + 2^{c_l}$ for some l. Once we have $n = q_1 d + d_1$, our next step is to write $d = q_2 d_1 + d_2$. We repeat the previous step until we find $d_2 < d_1$ and $q_2 =$ sum of powers of 2. Continuing this process which involves subsequent subtractions, at each step we have $d_k = q_{k+2} d_{k+1} + d_{k+2}$ where $d_{k+2} < d_{k+1}$. The process terminates when we get $d_k = 0$ somewhere and then $d_{k-1} = gcd(d, n)$. At each subtraction we reduce the length of our d_k by at least 1 so at most there are $\log n$ subtractions. And each subtraction involves subtracting two numbers of length $\log n$ so overall this procedure takes time $\mathcal{O}(\log^2 n)$. See (Knuth [7]) for an explanation of the gcd computation in $1.5 \log n$ multiprecision operations.

Computing e can be done by $1.25 \log n$ multiplications each followed by a reduction mod n so altogether $2.25 \log n$ multiprecision operations. On the other hand with bit operations, there are potentially $\mathcal{O}(\log n)$ steps in the powering algorithm to compute $a^{(n-1)/2} \pmod{n}$. Each step requires multiplying two numbers of length $\log n$ and dividing by n to get the remainder mod n. Both multiplying and dividing can be done in $\mathcal{O}(\log^2 n)$ steps. Then accumulating the partial products in the powering algorithm, there are $\mathcal{O}(\log n)$ steps multiplying two numbers of $\mathcal{O}(\log n)$ which is $\mathcal{O}(\log^2 n)$ steps. Overall, the computation takes time $\mathcal{O}(\log^3 n)$.

We compute j using the law of reciprocity for Jacobi symbols which is about a hard as a *gcd* computation (Dietzfelbinger [6]). Consider the following algorithm for the Jacobi symbol computation:

```
Let a ∈ ℤ and n ≥ 3 and odd integer.

0 Let x := a mod n, y := n, s := 1.

1  While x ≥ 2 do
2    While x ≡ 0 mod 4 do x := x/4; end do;
3    If x ≡ 0 mod 2 then
4      If y mod 8 ∈ 3,5 then s := −s; end if;
5      x := x/2;
6    end if;
7    If x = 1 then break; end if;
8    If x mod 4 ≡ y mod 4 ≡ 3 then s := −s; end if;
9    (x, y) := (y mod x, x);
10   end do;
11 end do;
12 return s · x;
```

Figure 2.1: Jacobi Symbol

Dividing a number x given in binary by 2 or by 4 amounts to dropping one or two trailing $0's$. Determining the remainder of x and y modulo 4 or modulo 8 amounts to looking at the last two or three bits of y. So the only costly operations we find in this algorithm are the divisions with remainder in lines 0 and 9. This we know has running time $\mathcal{O}(log^2 n)$. This makes the computation of the Jacobi symbol comparable to the Euclidean Algorithm (Dietzfelbinger [6]).

Thus, altogether we get a total number of $6 \log n$ multiprecision operations or $\mathcal{O}(log^3 n)$ steps in binary operations per a.

Chapter 3

Riemann's Hypothesis and Tests for Primality

3.1 Introduction

The second primality test is due to Gary Miller. Unconditionally, it has been proved to run in $\mathcal{O}(n^{.134})$ steps. Assuming the *ERH(Extended Riemann Hypothesis)*, the test runs faster at $\mathcal{O}(\log^4 n \log\log\log n)$ steps, i.e. in polynomial time. The test relies on the existence of a small quadratic nonresidue and is based on *Fermat's Little Theorem*. We want to use the converse of this famous theorem which can be difficult because a quadratic nonresidue may not be readily available to use as a witness to the compositeness of n. Another problem we encounter is the existence of Carmichael numbers which satisfy Fermat's congruence but are actually composite numbers.

Programming the simplified version of the algorithm by Miller (Figure 3.1) was not too difficult. But the largest loop in the modification of the algorithm by Miller (Figure 3.2) was particularly hard to work with and was by far the hardest to implement in Maple of the primality tests. This is due to the last few composite statements which are present to find nontrivial square roots of 1 modulo n. The goal of this section will be to prove the following two theorems (Miller [9]):

Theorem 3.1.1. *There exists an algorithm which tests primality in* $\mathcal{O}(n^{.134})$ *steps.*

Assuming the *ERH* leads us to the second theorem:

Theorem 3.1.2. *(ERH) There exists an algorithm which tests primality in* $\mathcal{O}(\log^4 n \log\log\log n)$ *steps.*

The difficulty in proving the two theorems is showing there exists a "small" quadratic nonresidue. The proof of Theorem 3.1.1 uses a result of Burgess, which in turn depends upon Weil's proof of the Riemann Hypothesis over finite fields. The proof of Theorem 3.1.2 uses Ankeny's bound for the size of the first quadratic nonresidue, assuming the *Extended Riemann Hypothesis.*

3.2 Notation and Definitions

We assume that the n we test for primality is always odd because we can easily test for divisibility by 2. We let p, q vary over odd primes. The exact power of 2 dividing n will be denoted by $\#_2(n)$, i.e. $\#_2(n) = max\{K : 2^K|n\}$.

Definition 3.2.1. Let $n = p_1^{v_1} \cdots p_m^{v_m}$ be the prime factorization of the odd number n. We then use the following three functions throughout the rest of this chapter to prove the two theorems:

$$(i)\ \phi(n) = p_1^{v_1-1}(p_1-1)\cdots p_m^{v_m-1}(p_m-1)\ (Euler's\ \phi - function),$$

$$(ii)\ \lambda(n) = lcm\{p_1^{v_1-1}(p_1-1), \ldots, p_m^{v_m-1}(p_m-1)\}\ (The\ Carmichael\ \lambda - function),$$

$$(iii)\ \lambda'(n) = lcm\{p_1 - 1, \ldots, p_m - 1\}.$$

Definition 3.2.2. For p prime we can choose a generator of the cyclic group $\mathbb{Z}_p^*$, say b. Then for $a \neq 0 \bmod p$ we define the index of $a \bmod p$ to be $ind_p(a) = min\{m : b^m \equiv a \pmod p\}$, noting this value is dependent upon our generator. We also say a is a q^{th} residue mod p if there exists b with $b^q \equiv a \pmod p$.

3.3 Outline of the Proofs

Recall that Fermat proved for $n = p$, a prime, and $gcd(a, p) = 1$, the following congruence holds:

$$a^{p-1} \equiv 1 \pmod p.$$

If we could find an $a,\ 1 < a < n$, so $a^{n-1} \not\equiv 1 \pmod n$, then n would have to be composite. As described in the introduction, such an a need not exist (because of the existence of Carmichael numbers), and even if such an a exists it may be very large. We remedy this using the definitions in (3.2.1).

Theorem 3.3.1. *(Carmichael [5]) For a given integer n, Fermat's Congruence $a^{n-1} \equiv 1 \bmod n$ holds for all a with $gcd(a, n) = 1$ if and only if $\lambda(n)|n - 1$.*

For example, the composite number $561 = 3 \cdot 11 \cdot 17$ meets the conditions of Theorem (3.3.1) because $\lambda(n) = lcm\{2, 10, 16\} = 80|560$. Then the $gcd(a, 561) = 1$ implies $a^{560} \equiv 1 \pmod{561}$ for all $a \in \mathbb{N}$ coprime to 561. In order to find a rigorous primality test, we will need to test a stronger condition than Fermat's congruence. If n is composite, we want to quickly find a witness for its compositeness. Instead of using Theorem (3.3.1) we are going to group composite numbers into two sets according to whether $\lambda'(n) \nmid n-1$ or $\lambda'(n)|n-1$ (Miller [9]).

> Let f be a computable function on the natural numbers. For input $n > 1$:
>
> (1) Check if n is a perfect power, i.e. $n = m^s$ where $s \geq 2$.
> If n is a perfect power, output "composite" and halt.
>
> (2) Carry out steps (i)-(iii) for each $a \leq f(n)$.
> If at any stage (i),(ii), or (iii) holds output "composite" and halt:
> (i) $a|n$,
> (ii) $a^{n-1} \neq 1 \bmod n$,
> (iii) $gcd((a^{\frac{n-1}{2^k}} \bmod n) - 1, n) \neq 1, n$ for some $k, 1 \leq k \leq \#_2(n-1)$.
>
> (3) Output "prime" and halt.

Figure 3.1: Definition of the Miller Algorithm for Primality Testing

Note. Miller's algorithm in Figure 3.1 is a simplified version of the algorithm needed for Theorem (3.1.2). This version gives an algorithm for testing primality in $\mathcal{O}(\log^5 n \log^2(\log n))$ steps assuming *ERH*. Before proving the Theorems (3.1.1) and (3.1.2) we develop the theory needed to define f and show there is an $a \leq f(n)$ which works.

3.4 Composite Numbers n satisfying $\lambda'(n) \nmid n-1$

Lemma 3.4.1. *If $\lambda'(n) \nmid n-1$, then there exist primes p, q such that:*

(1) $p|n,\ p-1 \nmid n-1,\ q^m|p-1,\ q^m \nmid n-1$ *for some integer* $m \geq 1$;

(2) *if a is any q^{th} nonresidue* $\bmod p$ *then* $a^{n-1} \neq 1 \bmod n$.

Proof. Let $q_1, \ldots, q_m$ be the distinct prime divisors of n. Since $\lambda'(n) = lcm\{q_1-1, \ldots, q_m-1\} \nmid n-1$ by assumption, we must have $q_i - 1 \nmid n-1$ for some i. Set $p = q_i$, giving $p|n$ and $p-1 \nmid n-1$ as in (1). Since $p-1 \nmid n-1$, there exists a prime q and an integer $m \geq 1$ such that $q^m | p-1$ and $q^m \nmid n-1$. This proves condition (1).

Suppose condition (2) is false and $a^{n-1} \equiv 1 \pmod{n}$. Let p be as above. Since $p|n$,

$$a^{n-1} \equiv 1 \pmod{p}. \tag{3.1}$$

Let b be a generator mod p; then by (3.1) we have $b^{(ind_p(a))(n-1)} \equiv 1 \pmod{p}$. Since $b^m \equiv 1 \pmod{p}$ implies $p-1|m$ we have

$$p-1|(ind_p(a))(n-1). \tag{3.2}$$

Now a is a q^{th} nonresidue mod p, so $q \nmid ind_p(a)$. Thus

$$q \nmid ind_p(a) \text{ and } q^m | p-1. \tag{3.3}$$

Applying (3.3) to (3.2) gives $q^m | n-1$, which is a contradiction to condition (1). □

Definition 3.4.2. Given a prime p and a prime q such that $q \nmid p-1$, let $N(p,q)$ be the least a such that a is a q^{th} nonresidue modulo p. Necessarily $N(p,q)$ is prime.

Proof. Suppose $a = N(p,q)$ is not prime and factors as $a = p_1 \cdots p_r$. Then if each of the $p_i, 1 \leq i \leq r$ are q^{th} residues mod p with $b_i^q \equiv p_i \bmod p$ then $(b_i \cdots b_r)^q \equiv (p_1 \cdots p_r) \equiv a \bmod p$ so n is also a q^{th} residue modulo p. Taking the contrapositive, the fact that each $p_i < a$ means that if a is a q^{th} nonresidue modulo p, then there must be some prime factor $p_j, 1 \leq j \leq r$ such that p_j is a q^{th} nonresidue modulo p which is smaller than a. So $N(p,q)$ must be prime. □

Theorem 3.4.3. *(Ankeny [3])(ERH)* $N(p,q) = \mathcal{O}(\log^2 p)$.

Using Ankeny's Theorem (3.4.3) and Lemma (3.4.1) we know that if $\lambda'(n) \nmid n-1$ then there exists an $a \leq \mathcal{O}(\log^2 n)$ such that $a^{n-1} \neq 1 \pmod{n}$.

3.5 Composite Numbers n satisfying $\lambda'(n)|n-1$

Definition 3.5.1. (Miller [9]) Let $q_1, \dots, q_m$ be the distinct prime divisors of n. By the definition of $\lambda'(n)$ we know that $\#_2(\lambda'(n)) = max(\#_2(q_1 - 1), \dots, \#_2(q_m - 1))$. We classify n as "Type A" or "Type B" according to the following conditions:

$$\text{Type A}: \quad \text{if for some } 1 \le j \le m,\ \#_2(\lambda'(n)) > \#_2(q_j - 1),$$

$$\text{Type B}: \quad \text{if } \#_2(\lambda'(n)) = \#_2(q_1 - 1) = \cdots = \#_2(q_m - 1).$$

To motivate the next few lemmas, consider a composite number $n = pq$, where p, q are primes, and suppose we have a number m so

$$m \equiv 1 \bmod q \quad \text{and} \quad m \equiv -1 \bmod p. \tag{3.4}$$

The first congruence implies $q|m-1$ and the second $m \not\equiv 1 \pmod n$. This gives us $gcd(m-1, n) = q$ so if we could compute this m in (3.4) efficiently, we would quickly know a divisor of n. The next three lemmas develop a strategy to finding such an m.

Lemma 3.5.2. *Let n be an odd composite number of type A, and let the primes p, q be such that $p|n$ and $q|n$, with $\#_2(\lambda'(n)) = \#_2(p-1) > \#_2(q-1)$. Assume further that $0 < a < n$ satisfies $\left(\frac{a}{p}\right) = -1$, where $\left(\frac{a}{p}\right)$ is the Jacobi symbol. Then either a has a nontrivial GCD with n or $(a^{\frac{\lambda'(n)}{2}} \bmod n) - 1$ has a nontrivial GCD with n.*

Proof. Suppose a has a trivial GCD with n. Because $1 < a < n$ we must have $gcd(a, n) = 1$. Since $q - 1|\lambda'(n)$ and $\#_2(q-1) < \#_2(\lambda'(n))$, we know $q - 1| \left(\frac{\lambda'(n)}{2}\right)$. Thus,

$$a^{\frac{\lambda'(n)}{2}} \equiv 1 \bmod q \tag{3.5}$$

by Fermat's Little Theorem.

Since $p - 1|\lambda'(n)$, again by Fermat we have $(a^{\frac{\lambda'(n)}{2}})^2 \equiv 1 \bmod p$ so $a^{\frac{\lambda'(n)}{2}} \equiv \pm 1 \bmod p$. Suppose $a^{\frac{\lambda'(n)}{2}} \equiv 1 \bmod p$. Then $p - 1|(ind_p a)(\frac{\lambda'(n)}{2})$ which implies that $ind_p a$ is even because $\#_2(\lambda'(n)) = \#_2(p-1)$. However, if $\left(\frac{a}{p}\right) = -1$ and g is a generator of $\mathbb{Z}_p^*$ with $g^k \equiv a \bmod p$, then considering Jacobi symbols we get $\left(\frac{a}{p}\right) = \left(\frac{g^k}{p}\right) = \left(\frac{g}{p}\right)^k = (-1)^k$. Note $\left(\frac{g}{p}\right) = -1$ or otherwise all of $\{1, \dots, p-1\}$ would be quadratic residues mod p when only half of them are. This argument implies $ind_p a$ is odd. This is an obvious contradiction so it must be true that

$$a^{\frac{\lambda'(n)}{2}} \equiv -1 \bmod p. \tag{3.6}$$

Combining (3.5) and (3.6) we get $gcd((a^{\frac{\lambda'(n)}{2}} \bmod n)-1, n) \neq 1, n$, so we must have a nontrivial divisor of n. □

Lemma 3.5.3. *If $p|n$, $\lambda'(n)|m$ and $k = \#_2\left[\frac{m}{\lambda'(n)}\right] + 1$, then $a^{\frac{\lambda'(n)}{2}} \equiv a^{\frac{m}{2^k}} \bmod p$.*

Proof. Assuming $a^{\lambda'(n)} \equiv 1 \bmod p$, we have $a^{\frac{\lambda'(n)}{2}} \equiv \pm 1 \bmod p$. Consider the two cases separately:

1. If $a^{\frac{\lambda'(n)}{2}} \equiv 1 \bmod p$, then $\lambda'(n)|m$ implies $\lambda'(n) \cdot c = m$ for some c. Then

$$\frac{m}{2^k} = \frac{\lambda'(n) \cdot c}{2^k} = \frac{\lambda'(n) \cdot c}{2^{\#_2\left[\frac{m}{\lambda'(n)}\right]+1}} = \frac{\lambda'(n) \cdot c}{2 \cdot 2^{\#_2\left[\frac{m}{\lambda'(n)}\right]}}$$

so $\left(\frac{\lambda'(n)}{2}\right) \mid \left(\frac{m}{2^k}\right)$ giving us $a^{\frac{m}{2^k}} \equiv 1 \bmod p$.

2. If instead $a^{\frac{\lambda'(n)}{2}} \equiv -1 \bmod p$ note that:

$$a^{\frac{m}{2^k}} \equiv (a^{\frac{\lambda'(n)}{2}})^{\frac{m}{\lambda'(n)2^{k-1}}} \equiv (-1)^{\frac{m}{\lambda'(n)2^{k-1}}} \bmod p.$$

Since $k-1 = \#_2\left[\frac{m}{\lambda'(n)}\right]$, $\frac{m}{\lambda'(n)2^{k-1}}$ is odd. Hence, $a^{\frac{m}{2^k}} \equiv -1 \equiv a^{\frac{\lambda'(n)}{2}} \bmod p$.

□

From Lemmas (3.5.2) and (3.5.3) we see that if n is a type A composite number, $\lambda'(n)|n-1$ and $a = N(p, 2)$, then either a or $gcd((a^{\frac{n-1}{2}} \bmod n) - 1, n)$ is a nontrivial divisor of n. For type B composite numbers we need more information.

Lemma 3.5.4. *Let n be an odd composite number with at least two distinct prime divisors, say p and q. Further suppose n is type B and $1 < a < n$ satisfies $\left(\frac{a}{pq}\right) = -1$, where $\left(\frac{a}{pq}\right)$ is the Jacobi symbol. Then, either a has a nontrivial GCD with n or $(a^{\frac{\lambda'(n)}{2}} \bmod n) - 1$ has a nontrivial GCD with n.*

Proof. As in the proof of Lemma (3.5.2) we assume that a has a trivial GCD with n, thus $gcd(a, n) = 1$. WLOG, assume $\left(\frac{a}{p}\right) = -1$ and $\left(\frac{a}{q}\right) = 1$. Using arguments similar to those in (3.5.2), we can show $a^{\frac{\lambda'(n)}{2}} \equiv -1 \bmod p$ and $a^{\frac{\lambda'(n)}{2}} \equiv 1 \bmod q$. The rest of the argument follows from the proof of Lemma 3.5.2. □

Definition 3.5.5. Let p and q be distinct primes. Define N(pq) to be the least a for which $\left(\frac{a}{pq}\right) \neq 1$, where $\left(\frac{a}{pq}\right)$ is the Jacobi symbol. Again $N(pq)$ is prime.

Theorem 3.5.6. *(Ankeny [3])(ERH)* $N(pq) = \mathcal{O}(\log^2(pq))$.

Proof of Theorem 3.1.2. (Miller [9]) Here we refer to the simplified version of the algorithm by Miller in Figure 3.1. By Ankeny's Theorems (3.4.3) and (3.5.6) which are dependent upon the ERH, there is a number $c \geq 1$ such that for all pairs of distinct primes p, q

$$N(p,q) \leq c(\log^2 p) \text{ and } N(pq) \leq c(\log^2(pq)).$$

Consider A_f where $f(n) = c(\log^2 n)$.

Analysis of Running Time of the Miller algorithm in Figure 3.1:

(1) The algorithm first checks to see if n is a perfect power. If $n = b^k$, then the least b could be is 2 so the biggest k occurs when $b = 2$. Then $k \leq \lfloor \log n \rfloor \cong \log n$. Thus there are $\mathcal{O}(\log n)$ exponents to consider. For each of these exponents $s = 1, 2, \ldots, \lfloor \log(n) \rfloor$, we do a binary search to find if there is a base b for which $b^s = n$. There will be $\log n$ steps in each such search. To compute b^s we use repeated squaring and multiply two binary numbers of $s \leq \log n$ digits which can be performed in $\mathcal{O}(\log^2 n)$ steps. Thus, this first step takes $\mathcal{O}(\log^4 n)$ steps.

(2) The algorithm next checks (i),(ii), and (iii) for $f(n)$ different values of a.

Check(i) involves division of two numbers of binary length $\mathcal{O}(\log n)$. This division can be carried out by a sequence of shifts and binary subtractions. As explained in the running time of the gcd in Chapter 2 there are at most $\mathcal{O}(\log n)$ shifts and $\mathcal{O}(\log n)$ subtractions of bits for each digit in the quotient. At the end we compare the remainder with 0 to see if they are equal or not. Overall, this check takes $\mathcal{O}(\log^2 n)$ steps.

Check(ii) involves verifying Fermat's Congruence. There are potentially $\mathcal{O}(\log n)$ steps in the powering algorithm to compute $a^{n-1} \pmod{n}$. Each step requires multiplying two numbers of length $\log n$ and dividing by n to get the remainder mod n. Both multiplying and dividing can be done in $\mathcal{O}(\log^2 n)$ steps and this procedure we denote $M(|n|)$ where $|n| = \log n$. Then accumulating the partial products in the powering algorithm, there are $\mathcal{O}(\log n)$ steps multiplying two numbers of $\mathcal{O}(\log n)$ which is again $M(|n|)$. Comparing with 1 mod n takes merely $\mathcal{O}(\log n)$ time. Thus, in all the check takes $\mathcal{O}(\log n \cdot M(|n|))$ steps.

Check(iii) again uses the powering algorithm to see if we can find a number which has a nontrivial *gcd* with n, by computing $(a^{\frac{n-1}{2^k}} \bmod n) - 1$ for some k such that $1 \leq k \leq \#_2(n-1)$. In particular $k \leq \log n$. It is necessary to do the computation for at most $\log n$ different values of k. As in check (ii) we know the computation of $a^{\frac{n-1}{2^k}} \bmod n$ takes $\mathcal{O}(\log n \cdot M(|n|))$ steps. Subtracting 1 from this value adds a negligible $\mathcal{O}(\log n)$.

All that remains is the computation of the greatest common divisor. Overall this procedure takes time $\mathcal{O}(\log^2 n)$ as described in Chapter 2. So far, we have $\mathcal{O}((((\log n) \cdot M(|n|)) + (\log^2 n)) \cdot (\log n))$. Now because multiplication takes at least $\log n$ steps, the check takes at most $\mathcal{O}((\log^2 n) \cdot M(|n|))$ steps.

So the Miller algorithm in Figure 3.1 runs in $\mathcal{O}((\log^4 n) \cdot M(|n|))$ steps (assuming the ERH) because check (iii) dominates the running time of this algorithm and we must perform the step 2(iii) in the algorithm for potentially $f(n) = \mathcal{O}(\log^2 n)$ number of $a's$. If we use the Schonhage-Strassen algorithm ([11]) for multiplying binary numbers, $M(|n|) = \mathcal{O}(\log n \;\; \log\log n \;\; \log\log\log n)$ so we get $\mathcal{O}(\log^5 n \;\; \log\log n \;\; \log\log\log n)$ steps.

Correctness of the Miller algorithm:

If n is prime, then Miller algorithm will declare n is prime, so we only need to show that it recognizes composite n. If n is composite, then one of the following three conditions holds:

(1) n is a prime power,

(2) $\lambda'(n) \nmid n - 1$,

(3) $\lambda'(n) | n - 1$ and n is not a prime power.

Case 1. If n is a prime power, then it is clearly a perfect power and the algorithm in Figure 3.1 will indicate n is composite in step 1 of the algorithm.

Case 2. If $\lambda'(n) \nmid n-1$, then by Lemma 3.4.1 there exist primes p and q such that if $a = N(p,q)$, then $a^{n-1} \neq 1 \bmod n$. We only need to note that $N(p,q) \leq f(n)$, which follows by Theorem 3.4.3 and our choice of f.

Case 3. If $\lambda'(n)|n-1$ and n is not a prime power:

(A) Suppose n is a type A composite number. Then by Lemmas 3.5.2 and 3.5.3 we can choose p and k, $(k \leq \#_2(n-1))$ such that if $a = N(p,2)$ then either $a|n$ or $gcd((a^{\frac{n-1}{2^k}} \bmod n) - 1, n) \neq 1, n$. Since $N(p,q) \leq f(n)$, n will be declared composite by either step 2(i) or step 2(ii).

(B) On the other hand, suppose n is a type B composite number. Then by Lemmas 3.5.4 and 3.5.3 and n not being a perfect power, we can choose p, q, and $k \geq \#_2(n-1)$ such that if $a = N(pq)$ then either $a|n$ or $gcd((a^{\frac{n-1}{2^k}} \bmod n) - 1, n) \neq 1, n$. Since $N(pq) \leq f(n)$ by Theorem 3.5.6, the algorithm will indicate n is composite. □

In order to prove Theorem 3.1.1 we use the following result due to Burgess:

Theorem 3.5.7. *(Burgess [4])*

$$N(p,q) = \mathcal{O}(p^{\frac{1}{4\sqrt{e}}+\epsilon}) \quad \textit{for any } \epsilon > 0,$$
$$N(pq) = \mathcal{O}((pq)^{\frac{1}{4\sqrt{e}}+\epsilon}) \quad \textit{for any } \epsilon > 0.$$

Proof of Theorem 3.1.1. (Miller [9]) Put $l = 4(2.71)^{\frac{1}{2}} < 4\sqrt{e}$, noting that $l \cong 6.58483$. By the above theorem, we can choose a number $c \geq 1$ such that for all pairs of distinct primes p, q,

$$N(p,q) \leq c \cdot p^{\frac{1}{l}} \quad \text{and} \quad N(pq) \leq c \cdot (pq)^{\frac{1}{l}}.$$

Consider the simplified version of the Miller algorithm, again from Figure 3.1, where $f(n) = \lceil cn^{\frac{1}{l+1}} \rceil \leq \lceil cn^{.133} \rceil$. We use $l+1$ in the exponent's denominator because we want to prove our algorithm tests primality in $\mathcal{O}(n^{.134})$ steps and $\frac{1}{l} > .134$ whereas $\frac{1}{l+1} < .134$. Using the size of $f(n)$ and looking back at the proof of Theorem 3.1.2, we see that the Miller algorithm runs in $\mathcal{O}(n^{.133} \cdot \log n \ \log\log n \ \log\log\log n)$ steps. We can absorb everything but the $n^{.133}$ into $n^{.001}$ by increasing the implied constant so we indeed have this algorithm running in $\mathcal{O}(n^{.134})$ steps. Hence we only need to show that it tests primality.

As before, if n is prime, the algorithm declares it prime. So we assume n is composite. Then n must fit into one of the following three cases:

Case 1. n is a prime power.
This case follows, as it did in the proof of Theorem 3.1.2, by step 1 of the algorithm.

Case 2. n has a divisor $\leq f(n)$.
This case was also explored in the previous proof and n is declared composite in step 2(i) of the algorithm.

Case 3. $\lambda'(n) \nmid n-1$ and n has no divisor $\leq f(n)$.
By Lemma 3.4.1 there are primes p, q so that if $a = N(p,q)$ then $a^{n-1} \neq 1 \bmod n$ so we just need to make sure that $a = N(p,q) \leq f(n)$ and that a was indeed tested in step 2. We have

$$a \leq \lceil cp^{\frac{1}{l}} \rceil \tag{3.7}$$

from the theorem above involving the size of $N(p,q)$. If $n = p \cdot a$ were true for some a and p with $p > \frac{n}{f(n)}$, then $\frac{n}{a} > \frac{n}{f(n)}$, hence $a \leq f(n)$. Thus there is an a with $1 < a \leq f(n)$ for which $a|n$ which has been ruled out by Step 2(i) of the algorithm. It follows that for each prime dividing n, we have

$$p \leq \frac{n}{f(n)}, \quad i.e., \quad p \leq \lceil \left(\frac{1}{c}\right) n^{\frac{l}{l+1}} \rceil. \tag{3.8}$$

Substituting (3.8) into (3.7), we have

$$a \leq \lceil n^{\frac{1}{l+1}} \rceil \leq f(n), \quad \text{since } c \geq 1.$$

(Subcase 3A) Suppose n is a type A composite number. As in Case 3A of the proof of Theorem 3.1.2, it is necessary to show $a = N(p,2) \leq f(n)$ where $p|n$. Because equations (3.7) and (3.8) hold, we get the result $a \leq f(n)$ by the work above.

(Subcase 3B) Assume n is a type B composite number. Since n is not a prime power it has at least two distinct prime divisors, say p, q. Again, we must show that $N(pq) \leq f(n)$ which follows if we can show $pq \leq \frac{n}{f(n)}$.

Claim 3.5.8. *(Carmichael [5]) $n \neq pq$.*

Proof. Suppose $n = pq$ where $p < q$. Then $pq - 1 = ((p-1)+1)((q-1)+1) - 1 = (p-1)(q-1) + (q-1) + (p-1)$. And $q - 1 | pq - 1$ since $\lambda'(n) | n - 1$. This implies that $q - 1 | p - 1$. Hence $q \leq p$ must be true contradicting $p < q$. □

Thus, $n = pqr$ where $r \neq 1$. Since $r|n$, we have $r \geq f(n)$ because we have already tested whether $a|n$ for all $a \leq f(n)$. Hence $pq = \frac{n}{r} \leq \frac{n}{f(n)}$ and we have

$$N(pq) \leq c(pq)^{\frac{1}{l}} \leq c\left(\frac{n}{f(n)}\right)^{\frac{1}{l}} = c\left(\frac{n}{\lceil cn^{\frac{1}{l+1}} \rceil}\right)^{\frac{1}{l}} = \left(n^{\frac{l}{l+1}}\right)^{\frac{1}{l}} = n^{\frac{1}{l+1}} = f(n).$$

□

3.6 Modification to the Miller Algorithm

In Figure 3.2, a modified version of the Miller algorithm is given. It speeds up the process by only testing prime numbers $\leq f(n)$ instead of all numbers $\leq f(n)$ in step 2. Why is this correct? In step 2(i) if $a|n$, then for some prime dividing a, we also have $p|n$. If we find that $a^{n-1} \neq 1 \bmod n$ in step 2(ii), the Chinese remainder theorem proves there must be some $p|a$ such that $p^{n-1} \neq 1 \bmod n$.

Lastly in step 2(iii), we see that if we find a nontrivial divisor of n, then for some prime q , we have $q|n$ and $q|a^{\frac{n-1}{2^k}} - 1$. Recall that Lemma (3.5.2) says if n is type A and $pq|n$, where $\#_2(p-1) > \#_2(q-1)$, then if $0 < a < n$ is such that $\left(\frac{a}{p}\right) = -1$, (where $\left(\frac{a}{p}\right)$ is the Jacobi symbol), either $gcd(a, n) \neq 1, n$ or $gcd((a^{\frac{\lambda'(n)}{2}} \bmod n) - 1, n) \neq 1, n$. Suppose $a(\leq f(n))$ factors as $a = p_1 \cdots p_m$ where p_j are prime for $j \in \{1, \ldots, m\}$. Then $\left(\frac{a}{p}\right) = \left(\frac{p_1}{p}\right) \cdots \left(\frac{p_m}{p}\right) = -1$ implies

Amend the Miller algorithm as follows:

(1) If n is a perfect power, output "composite" and halt.

(2) Compute $p_1, \ldots, p_m$ where p_i is the i^{th} prime number and m is so that $p_m \leq f(n) < p_{m+1}$. Compute Q, S so that $n-1 = Q2^S$ and Q is odd.
Let $i = 1$ and proceed to (ii). Denote p_i by a throughout.

(i) If $i < m$ set i to $i+1$. If $i = m$ then output "prime" and halt.
(ii) If $a|n$ then output "composite" and halt.

Compute $a^Q \bmod n, a^{Q2} \bmod n, \ldots, a^{Q2^S} = a^{n-1} \bmod n$.
(iii) If $a^{Q2^S} \neq 1 \bmod n$ then output "composite" and halt.
(iv) If $a^Q \equiv 1 \bmod n$ go to (i).

Set $J = max(J : a^{Q2^J} \neq 1 \bmod n)$.
(v) If $a^{Q2^J} \equiv -1 \bmod n$ go to (i).
(vi) Output "composite" and halt.

Figure 3.2: Modification of the Miller Algorithm for Primality Testing

$\left(\frac{p_i}{p}\right) = -1$ for some $i, 1 \leq i \leq m$. Thus, we can use this p_i instead of a.

If n is instead of type B, then Lemma (3.5.4) says for distinct prime divisors p, q with $\#_2(p-1) = \#_2(q-1)$ and for $0 < a < n$, supposing $\left(\frac{a}{pq}\right) = -1$, then either $gcd(a,n) \neq 1, n$ or $gcd((a^{\frac{\lambda'(n)}{2}} \bmod n) - 1, n) \neq 1, n$. Again we assume a factors into primes as $a = p_1 \cdots p_m$. Then $\left(\frac{a}{pq}\right) = \left(\frac{p_1}{pq}\right) \cdots \left(\frac{p_m}{pq}\right) = -1$ implies $\left(\frac{p_i}{pq}\right) = -1$ for some $i, 1 \leq i \leq m$ as long as $p_i \neq p, q$. Then we can use this p_i instead of a. If $p_i = p$ or $p_i = q$ then we have found a divisor of n by dividing p_i into n.

Since the number of primes $\leq f(n)$ is $\mathcal{O}\left(\frac{f(n)}{log f(n)}\right)$ by the prime number theorem, we get an upper bound of $\mathcal{O}(\log^5 n \log\log\log n)$ steps for Theorem 3.1.2.

As before in the proof of Theorem 3.1.2, which used the simplified version of the Miller algorithm in Figure 3.1, the running time of the modified Miller algorithm in Figure 3.2 is dominated by step 2. Assume f is as before and now the modified algorithm computes the following:

(1) the first m primes, which takes $\mathcal{O}((f(n))^3)$ by the naïve sieve method. This method can be executed with an array $A\,[2\ldots f(n)]$, initializing each value $A[k]=1$. Then we can use a loop from $k=2$ to $k=\sqrt{f(n)}$. In each run of the loop we let $i=2k$ and have an inside loop: while $i\leq f(n)$ do $A[i]:=0, i:=i+k$. The first loop is naïvely bounded by $\sqrt{f(n)}$ binary steps and the inside loop is naïvely bounded by $f(n)$ steps, so we get $\mathcal{O}((f(n))^3)$ steps.

(2) $a^{n-1} \pmod n$ where a varies over the first m primes. We do this by repeated squaring beginning with $a^Q \pmod n$. There are $\log n$ squarings and each multiplication takes $\mathcal{O}(M|n|)$ steps as discussed before. Thus because there are $\mathcal{O}\left(\frac{f(n)}{\log(f(n))}\right)$ primes $\leq f(n)$ the running time of the modified Miller algorithm is $\mathcal{O}(m\ \log n\ M(|n|)) = \mathcal{O}(\log^4 n \log\log\log n)$.

To show the modified Miller algorithm tests primality we only need to reconsider Case 3:

Case 3. $\lambda'(n)|n-1$ and n is not a prime power.

(A) Suppose n is a type A composite number with $\#_2(\lambda'(n)) = \#_2(p-1) > \#_2(q-1)$ and $p,q|n$. Let $a=N(p,2)$ so a is prime. We need to show either step (ii),(iii), or (vi) outputs "composite" for this particular a in this modification to the Miller algorithm. Suppose $a\nmid n$ and $a^{n-1}\equiv 1 \bmod n$, i.e. our n has passed through the first two determinations of compositeness in steps (ii) and (iii). Let us see in this case that the algorithm reaches step (vi).

Suppose $a^Q\equiv 1 \bmod n$. Then $a^Q\equiv 1 \bmod p$ since $p|n$. Now p is odd and $\left(\frac{a}{p}\right)=-1$, where $\left(\frac{a}{p}\right)$ is the Jacobi symbol, so we have

$$1=\left(\frac{1}{p}\right)=\left(\frac{a^Q}{p}\right)=\left(\frac{a}{p}\right)^Q=(-1)^Q$$

which implies that Q must be even. This is an obvious contradiction to Q being odd. Thus $a^Q\not\equiv 1 \bmod n$ so the modified Miller algorithm will reach step (v). By Lemmas (3.5.2) and (3.5.3), we know there exists a k such that $a^{Q2^k}\equiv 1 \bmod q$ and $a^{Q2^k}\equiv -1 \bmod p$. Suppose $a^{Q2^J}\equiv -1 \bmod n$ so $a^{Q2^J}\equiv -1 \bmod p$ and mod q, where J is defined in the modified algorithm. Then $a^{Q2^k}\equiv a^{Q2^J}\equiv -1 \bmod p$ implies $k=J$. But we have $a^{Q2^k}\equiv 1 \bmod q$ and $a^{Q2^J}\equiv -1 \bmod q$

implies $k > J$ on the other hand. This is a contradiction to the maximality of J so $a^{Q2^J} \neq -1 \bmod n$. Thus the algorithm now reaches step (vi).

(B) Suppose n is a type B composite number. The proof follows the argument of Case A.

Chapter 4

Primes is in P

4.1 Introduction

In the third test, we look at an unconditional deterministic polynomial-time algorithm for primality testing due to Agrawal, Kayal and Saxena. Given an odd integer n, we can determine if it is prime based upon the fact that $(X + a)^n \equiv (X^n + a) \pmod{n}$ if and only if n is prime. Using this congruence as a primality test is rather inefficient if we search for $a's$ that make this congruence fail, because as n gets larger we must compute n coefficients on the left-hand side. The beauty of this algorithm is that the authors found a way around this computation by instead computing $(X + a)^n \equiv (X^n + a) \pmod{X^r - 1, n}$ for an appropriately chosen r. We then show it is sufficient to test $\lfloor 2\sqrt{r} \cdot \log n \rfloor + 1$ many $a's$ to conclude n is a prime power. Lastly, we show that n is not a prime power so it must be prime. The only difficulty that I found in programming this algorithm was finding the crucial r so the order of n modulo r is greater than $4 \log^2 n$. To cut down on the cost of this procedure we use repeated squaring to compute the left-hand side and the cost amounts to $\mathcal{O}\tilde{}(\log^{\frac{21}{2}} n)$. This notation is defined in Section 4.3.

4.2 The Idea

This test is based on the following lemma which generalizes Fermat's Little Theorem.

Lemma 4.2.1. *(Agrawal, Kayal and Saxena [2]) Let $a \in \mathbb{Z}$, $n \in \mathbb{N}$, $n \geq 2$, and $gcd(a, n) = 1$. Then n is prime if and only if*

$$(X + a)^n \equiv (X^n + a) \bmod n.$$

Proof. For $0 < i < n$, the coefficient of X^i in $((X + a)^n - (X^n + a))$ is $\binom{n}{i} a^{n-i}$.

Suppose n is prime. Then the binomial coefficients are all divisible by n so $\binom{n}{i} \equiv 0 \pmod{n}$ for

$1 \leq i \leq n-1$, while $a^n \equiv a \bmod n$ by Fermat's Little Theorem, and the congruence holds. Now suppose n is composite and q is one prime factor where $q^k \| n$. Note that the binomial coefficient $\binom{n}{q} = \frac{n!}{q!(n-q)!} = \frac{n(n-1)\cdots(n-q+1)}{1\cdots q}$. We know $q^k \| n$ in the numerator but q is prime and does not divide any of the other numbers in the numerator. Also $q \| q!$ so the order of q in $\binom{n}{q}$ is $k-1$ and thus $q^k \nmid \binom{n}{q}$. Now the $gcd(a,n) = 1$ so $gcd(a,q) = 1$ and $q \nmid a^{n-q}$. Hence the coefficient of X^q is $\binom{n}{q} a^{n-q} \neq 0 \pmod{q^k}$ and thus $\binom{n}{q} a^{n-q} \neq 0 \pmod{n}$. This means that $((X+a)^n - (X^n + a)) \neq 0$ in $\mathbb{Z}_n$. □

4.3 Notation and Preliminaries

The Class **P** refers to all problems solvable in polynomial time, i.e. the class of sets accepted by deterministic polynomial-time Turing machines. Thus, the title *Primes is in P* of the AKS paper says we can determine primality using the AKS test in polynomial time. The Class **NP** refers to all problems that are verifiable in polynomial time given a non-deterministic algorithm.

$\mathbb{Z}_n$ is the ring of integers modulo n. $\mathbb{F}_p$ denotes the finite field with p elements, where p is a prime. Remember that if p is prime, and $h(X)$ is a polynomial of degree d and irreducible in $\mathbb{F}_p$, then $\mathbb{F}_p[X]/(h(X))$ is a finite field of order p^d. The notation $f(X) = g(X) \pmod{h(X), n}$ represents the equation $f(X) = g(X)$ in the ring $\mathbb{Z}_n[X]/(h(X))$.

We will use the symbol $\mathcal{O}\tilde{}\,(t(n))$ for $\mathcal{O}(t(n) \cdot poly(\log t(n)))$, where $t(n)$ is any function on n. As an example, $\mathcal{O}\tilde{}\,(\log^k n) = \mathcal{O}(\log^k n \cdot poly(\log\log n)) = \mathcal{O}(\log^{k+\varepsilon} n)$ for any $\varepsilon > 0$. We continue to use log for base 2 logarithm, and *ln* for natural logarithm.

Given $r \in \mathbb{N}$, $a \in \mathbb{Z}$ with $gcd(a,r) = 1$, the *the order of a modulo r* is the smallest number k such that $a^k \equiv 1 \pmod{r}$, denoted $o_r(a)$. We let $\phi(r)$ denote *Euler's totient function* and notice that $o_r(a) | \phi(r)$ for any a, $gcd(a,r) = 1$.

Lemma 4.3.1. *Let LCM(m) denote the lcm of the first m numbers. For $m \geq 7$:*

$$LCM(m) \geq 2^m.$$

See (Radhakrishnan, Telikepalli and Vinay [10]) for a proof.

Input: integer $n > 1$.

1. If ($n = a^b$ for $a \in \mathcal{N}$ and $b > 1$), output COMPOSITE.
2. Find the smallest r such that $o_r(n) > 4\log^2 n$.
3. If $1 < gcd(a, n) < n$ for some $a \leq r$, output COMPOSITE.
4. If $n \leq r$, output PRIME.
5. For $a = 1$ to $\lfloor 2\sqrt{r}\log n \rfloor + 1$ do
   if $((X + a)^n \neq X^n + a (\text{mod} X^r - 1, n))$, output COMPOSITE.
6. Output PRIME.

Figure 4.1: AKS Algorithm for Primality Testing

4.4 The Algorithm and Proof of its Correctness

Theorem 4.4.1. *The algorithm above returns PRIME if and only if n is prime.*

In order to prove this theorem we need a sequence of lemmas; the first one is trivial.

Lemma 4.4.2. *If n is prime, the algorithm returns PRIME.*

Proof. If n is prime, then steps 1 and 3 can never return COMPOSITE. By Lemma 4.2.1, the `for` loop cannot return COMPOSITE either. Thus the algorithm will determine n is PRIME in either step 4 or 6. □

The converse of Lemma 4.4.2 requires more work. If the algorithm returns PRIME in step 4 then n must be prime since step 3 would have otherwise found a nontrivial factor of n. So we only need to consider the case when the algorithm declares n PRIME in step 6. We assume this from now on.

The algorithm has two main steps, 2 and 5. We first prove the existence and bound the number r in

$$(X + a)^n \equiv (X^n + a) \pmod{X^r - 1, n}. \tag{4.1}$$

Lemma 4.4.3. *(Tou, Alexander [14]) There exists an $r \leq \lceil 16\log^5 n \rceil$ such that $o_r(n) > 4\log^2 n$ for $n \geq 2$.*

Proof. Let $r_1, r_2, \ldots, r_t$ be all the numbers such that $r_i \leq \lceil 16 \log^5 n \rceil$ and $o_{r_i}(n) \leq 4 \log^2 n$. We need to prove that $\{r_1, r_2, \ldots, r_t\} \neq \{1, 2, \ldots, \lceil 16 \log^5 n \rceil\}$. For each r_i, by assumption

$$n^j \equiv 1 \pmod{r_i}, \text{ for some } j \leq 4 \log^2 n.$$

This implies that $r_i | n^j - 1$ for some $j \leq 4 \log^2 n$. Then we have

$$\begin{aligned} r_i | \prod_{j=1}^{\lfloor 4 \log^2 n \rfloor} (n^j - 1) &< \prod_{j=1}^{\lfloor 4 \log^2 n \rfloor} (n^j) \\ &= n^{\{\sum_{j=1}^{\lfloor 4 \log^2 n \rfloor} (j)\}} \\ &= n^{\{\frac{\lfloor 4 \log^2 n \rfloor \left(\lfloor 4 \log^2 n \rfloor + 1 \right)}{2}\}} \\ &< n^{\{\frac{\lfloor 4 \log^2 n \rfloor \left(\lfloor 4 \log^2 n \rfloor + \lfloor 4 \log^2 n \rfloor \right)}{2}\}} \\ &= n^{\lfloor 4 \log^2 n \rfloor^2} \\ &\leq n^{16 \log^4 n} \\ &= 2^{16 \log^5 n} \ (as\ n = 2^{\log n}). \end{aligned}$$

Now suppose by contradiction that the $r_i's$ are *all* the numbers $\leq \lceil 16 \log^5 n \rceil$, i.e. $r_1 = 1, r_2 = 2, \ldots, r_t = \lceil 16 \log^5 n \rceil$. Then $1, 2, \ldots, \lceil 16 \log^5 n \rceil$ all divide a number strictly smaller than $2^{16 \log^5 n}$. But Lemma 4.3.1 says the least common multiple of the first $\lceil 16 \log^5 n \rceil$ numbers is at least $2^{\lceil 16 \log^5 n \rceil}$. This is a contradiction. Therefore, there exists a number $r \leq \lceil 16 \log^5 n \rceil$ such that $o_r(n) > 4 \log^2 n$. $\square$

Let us assume that n is a composite number and the algorithm outputs PRIME in step 6. We will show this leads to a contradiction, thus proving the other direction of Theorem 4.4.1. Let $l = \lfloor 2\sqrt{r} \log n \rfloor + 1$. Because n passes all the congruences in step 5, we know that

$$for\ a = 1, 2, \ldots, l, \ (X + a)^n \equiv X^n + a \pmod{X^r - 1, n}. \tag{4.2}$$

In the above identity we may replace n in the modulus by any divisor of n. Let p be one such prime divisor. Then instead we have

$$for\ a = 1, 2, \ldots, l, \ (X + a)^n \equiv X^n + a \pmod{X^r - 1, p}. \tag{4.3}$$

Since p is prime, we always have

$$for\ a = 1, 2, \ldots, l,\ (X + a)^p \equiv X^p + a \pmod{X^r - 1, p}. \tag{4.4}$$

In (4.3) and (4.4) the numbers n and p satisfy similar identities (Radhakrishnan, Telikepalli and Vinay [10]). The AKS creators name these *introspective numbers.* To continue our proof of correctness, we will show that introspective numbers are multiplicative.

Claim 4.4.4. *(Agrawal,Kayal and Saxena [2]) Suppose*

$$(X + a)^{m_1} \equiv X^{m_1} + a \pmod{X^r - 1, p}$$

$$(X + a)^{m_2} \equiv X^{m_2} + a \pmod{X^r - 1, p}.$$

Then, $(X + a)^{m_1 m_2} \equiv X^{m_1 m_2} + a \pmod{X^r - 1, p}$.

Proof. (Radhakrishnan, Telikepalli and Vinay [10]) The second assumption says that $(X + a)^{m_2} - (X^{m_2} + a) = (X^r - 1)g(X) \pmod{p}$, for some polynomial $g(X)$. Substituting X^{m_1} for X in this identity, we get

$$(X^{m_1} + a)^{m_2} - (X^{m_1 m_2} + a) = (X^{m_1 r} - 1)g(X^{m_1}) \pmod{p}.$$

Since $(X^r - 1)|(X^{m_1 r} - 1)$, this gives us

$$(X^{m_1} + a)^{m_2} \equiv (X^{m_1 m_2} + a) \pmod{X^r - 1, p}.$$

Using this and the first assumption in the claim, we obtain

$$(X + a)^{m_1 m_2} \equiv (X^{m_1} + a)^{m_2} \equiv (X^{m_1 m_2} + a) \pmod{X^r - 1, p}.$$

□

Now starting with (4.3) and (4.4) and using the claim we just proved, we see that for each m of the form $p^i n^j$ $(i, j \geq 0)$ we have

$$(X + a)^m \equiv X^m + a \pmod{X^r - 1, p},\ for\ a = 1, 2, \ldots, l.$$

(The case $i, j = 0$ corresponding to $m = 1$ is trivially true.)

Consider the list $L = (p^i n^j : 0 \leq i, j \leq \lfloor\sqrt{t}\rfloor)$ where t is the order of the subgroup G of $\mathbb{Z}_r^*$, generated by p and n taken modulo r. All the elements have size at most $n^{2\sqrt{t}}$. Each element in L taken modulo r lands in the subgroup G, but $|L| = (\lfloor\sqrt{t}\rfloor + 1)^2 > t = |G|$. So we must have two numbers that are congruent modulo r; call them $m_1 = p^{i_1} n^{j_1}$ *and* $m_2 = p^{i_2} n^{j_2} = m_1 + kr$, where we assume $m_1 < m_2$ and $(i_1, j_1) \neq (i_2, j_2)$. From here on, we concentrate on these two numbers congruent modulo r. Note we have $(X+a)^{m_2} \equiv X^{m_1+kr} + a \equiv X^{m_1} + a \equiv (X+a)^{m_1} \pmod{X^r - 1, p}$ since $X^r \equiv 1 \pmod{X^r - 1}$. Thus,

$$for\ a = 1, 2, \ldots, l,\ (X + a)^{m_1} \equiv (X + a)^{m_2} \pmod{X^r - 1, p}. \tag{4.5}$$

Claim 4.4.5. *(Agrawal, Kayal and Saxena [2])* $m_1 = m_2$.

Assuming this claim, we see that $p^{i_1} n^{j_1} = p^{i_2} n^{j_2}$. Since we assumed that $(i_1, j_1) \neq (i_2, j_2)$ and p is prime, n must be a power of p. That is, $n = p^s$ for some s. If $s \geq 2$, the algorithm would have output COMPOSITE in step 1 contradicting our assumption that it declared n to be PRIME. Then $s = 1$ is the only option contradicting our other assumption that n is composite. Thus if the algorithm outputs PRIME, then n is prime proving the algorithm is accurate. We now prove Claim 4.4.5.

Proof of Claim 4.4.5. (Radhakrishnan, Telikepalli and Vinay [10]) This proof uses the elementary fact that *in a field, a non-zero polynomial of degree d has at most d roots.* Consider the polynomial $b(Z) = Z^{m_1} - Z^{m_2}$. If we can show $b(Z)$ has more roots than its degree $d = max\{m_1, m_2\}$ in some field, then $b(Z) \equiv 0$ and thus $m_1 = m_2$.

Moving to a Field: To start, we must move from the ring $\mathbb{F}_p[X]/(X^r - 1)$ to a field. Let ω be a primitive r^{th} root of unity. By (4.5) we can write

$$for\ a = 1, 2, \ldots, l,\ (\omega + a)^{m_1} = (\omega + a)^{m_2} \tag{4.6}$$

in the field $\mathbb{F}_p(\omega)$, making $(\omega + a)$ a root of $b(Z)$. Note that if e_1, e_2 are roots of $b(Z)$, then $b(e_1 e_2) = (e_1 e_2)^{m_1} - (e_1 e_2)^{m_2} = e_1^{m_1} e_2^{m_1} - e_1^{m_2} e_2^{m_2} = 0$ because $e_1^{m_1} = e_1^{m_2}$ and $e_2^{m_1} = e_2^{m_2}$. So $e_1 e_2$ is also a root. This implies that each element of the form $\prod_{a=1}^{l} (\omega + a)^{\alpha_a},\ \alpha_a \geq 0$ is a root of $b(Z)$ as well.

Here $t = |G| \le r-1$ because G is a subgroup of $\mathbb{Z}_r^*$. Put $l' := \lfloor 2\sqrt{t}\log n \rfloor + 1 \le l$, and consider the set

$$S = \{\prod_{a=1}^{l'} (\omega + a)^{\alpha_a} | \alpha_a \in \{0,1\}\}.$$

Each element of S is a root of $b(Z)$. We claim that S has $2^{l'}$ elements, implying $b(Z)$ has at least $2^{l'}$ roots in the field $\mathbb{F}_p(\omega)$. If this is so, then since $m_1, m_2 \le n^{2\lfloor\sqrt{t}\rfloor}$ while $2^{l'} > n^{2\lfloor\sqrt{t}\rfloor}$, $b(Z)$ would have more roots than its degree proving $b(Z) = Z^{m_1} - Z^{m_2} \equiv 0$ and $m_1 = m_2$.

Roots of b(Z): Each element in S is found by substituting ω for X in a polynomial of the form $\prod_{a=1}^{l'}(X+a)^{\alpha_a} \in \mathbb{F}_p[X]$. In step 3 of the algorithm, which our n is assumed to have passed, we have seen n has no small divisors so neither does p. Thus $l' < t \le r-1 < r < p$ so each $X + a,\ a = 1, \ldots, l'$ is distinct in $\mathbb{F}_p[X]$. Since the elements of $\mathbb{F}_p[X]$ factor uniquely into irreducible factors, we get different polynomials from different products. We want to show that different products $g(X) = \prod_{a=1}^{l'}(X+a)^{\alpha_a}$ yield different $g(\omega) = \prod_{a=1}^{l'}(\omega+a)^{\alpha_a}$ in $\mathbb{F}_p(\omega)$. By Claim 4.4.4, $g(X)^m = g(X^m) \pmod{X^r - 1, p}$ for each $g(X)$ of the form above, and $m = p^i n^j$. Hence $g(\omega)^m = g(\omega^m)$ in $\mathbb{F}_p(\omega)$ for each such m.

If $g_1(X)$ and $g_2(X)$ have the specified form and $g_1(\omega) = g_2(\omega)$, then $g_1(\omega^m) = g_2(\omega^m)$. Thus, each ω^m ($m = n^i p^j,\ i, j \ge 0$) is a root of $g_1(X) - g_2(X)$ in $\mathbb{F}_p[X]$. The number of distinct values ω^m is the same as the number of distinct residues modulo r generated by $n^i p^j$, because ω is a primitive r^{th} root of unity. This means that $g_1(X) - g_2(X)$ has at least t roots in $\mathbb{F}_p(\omega)$, while the degree of each polynomial is at most l'. Because $t \ge o_r(n) > 4\log^2 n$, we have $l' < t$, so $g_1(X) - g_2(X) \equiv 0$ must be true in $\mathbb{F}_p[X]$. Thus, we get the distinct elements in $\mathbb{F}_p(\omega)$ we were searching for upon substituting ω for X. In summary, we have proved S has $2^{l'}$ distinct elements which are distinct roots of $b(Z)$ in $\mathbb{F}_p(\omega)$. So $b(Z) \equiv 0$ and $m_1 = m_2$. □

4.5 Running Time Analysis

(Agrawal, Kayal and Saxena [2]) To calculate the time complexity of this algorithm, we use the fact that addition, multiplication, and division operations between two m bit numbers can be performed in time $\mathcal{O}\tilde{}(m)$. These operations on two degree d polynomials with coefficients at most m bits can be done in time $\mathcal{O}\tilde{}(d \cdot m)$ steps. Recall that the symbol $\mathcal{O}\tilde{}(t(n))$ stands for $\mathcal{O}(t(n) \cdot poly(\log t(n)))$. Here $t(n)$ is any function on n and $\mathcal{O}(t(n))$ means there exists a Turing machine which correctly indicates whether n is prime or composite in less than $C \cdot t(n)$ steps, for some constant C.

Theorem 4.5.1. *The asymptotic time complexity of the algorithm is* $\mathcal{O}\tilde{}(\log^{\frac{21}{2}} n)$.

Proof. The first step in this algorithm determines whether or not n is a prime power. As in the Miller running time analysis, there are $\mathcal{O}(\log n)$ possible exponents to consider and $\log n$ steps in each binary search for each exponent. To compute b^s we use repeated squaring and multiply two binary numbers of $s \leq \log n$ digits which we said would be performed in $\mathcal{O}\tilde{}(\log n)$ time. Thus, step 1 of the algorithm takes $\mathcal{O}\tilde{}(\log^3 n)$ time.

In step 2, we look for the least number r for which $o_r(n) > 4\log^2 n$. This is done by trying successive values of r and testing if $n^k \neq 1 \pmod r$ for every $k \leq 4\log^2 n$. For a particular r, this involves at most $\mathcal{O}(\log^2 n)$ multiplications modulo r and so will take time $\mathcal{O}\tilde{}(\log^2 n \log r)$. By Lemma 4.4.3, we know we only have to test $\mathcal{O}(\log^5 n)$ different $r's$ so the total time complexity for step 2 is $\mathcal{O}\tilde{}(\log^7 n)$.

Step 3 involves the computation of greatest common divisors of r numbers. Each gcd computation takes time $\mathcal{O}(\log^2 n)$ (as explained in Chapter 2) so altogether this step takes time $\mathcal{O}(r\log^2 n) = \mathcal{O}(\log^7 n)$. Step 4 only requires comparing two numbers of approximate size $\log n$ so it takes time $\mathcal{O}(\log n)$.

In step 5, we must verify $\lfloor 2\sqrt{r}\log n \rfloor + 1$ equations. Each equation requires $\mathcal{O}(\log n)$ multiplications of degree r polynomials with coefficients of size $\mathcal{O}(\log n)$. So each equation is verified in time $\mathcal{O}\tilde{}(r\log^2 n)$ steps by above. Thus, the time of step 5 is $\mathcal{O}\tilde{}(r\sqrt{r}\log^3 n) = \mathcal{O}\tilde{}(r^{\frac{3}{2}}\log^3 n) = \mathcal{O}\tilde{}(\log^{\frac{21}{2}} n)$. This time is dominant compared to the others so it becomes the time complexity of the algorithm. $\square$

Chapter 5

Analysis of Maple Calculations

5.1 Key to Thesis Calculations in Spreadsheets

```
*c=composite

Monte-Carlo test:
c1= nontrivial gcd with n
c2= nontrivial square root of 1 mod n
c3= if e<>j mod n
Probably prime= n is probably prime after a reasonable number of a's tested

Gary Miller test:
c1= n is a perfect power
c2= nontrivial divisor of n
c3= prime[a] fails Fermat test
c4= prime[a]^Q (mod n) gives a nontrivial square root of 1 mod n
c5= d is a nontrivial square root of 1 mod n
c6= prime[a]^(Q*2^(S-1)) is a nontrivial square root of 1 mod n
prime= after all primes have been tested in the while loop, n must be prime

AKS test:
c1= n is a perfect power
c2= nontrivial gcd with n
c3= nontrivial gcd with n
c4= (X+a)^n<>X^n+a mod n
prime1= if r>=n
prime2= the equality holds in c4 for all a from 1 to L
```

5.2 Observations from Maple Data

I recorded data from my Maple procedures in excel spreadsheets and they can be found in Appendix C. There are multiple outputs and step counts for the Monte-Carlo test by Solovay and Strassen due to the randomness of the selection of the number a. Four different types of numbers n were tested: Prime, Carmichael, Composite and Perfect Power. So when I refer to composite below, it does not include Carmichael numbers or Perfect Powers unless indicated. The key to the outputs are above for reference.

5.2.1 Monte-Carlo test by Solovay and Strassen

As the size of our number tested increased, we were less likely to find a nontrivial gcd with n because the factors are larger and $2 \ldots n-1$ is a larger range of numbers from $a[b]$ to be chosen from. When we reach composite2 in this test, we are attempting to find Carmichael numbers because prime numbers have no nontrivial square roots of 1. As our n gets larger though, it seemed to be less likely that we would catch our Carmichael numbers here. But overall, most of our composite and Carmichael numbers return composite2.

In this test, we have no specific output for a perfect power as we do in the other two tests. Our perfect powers tested never outputted anything but composite1 or composite2, i.e. a nontrivial gcd with n or a nontrivial square root of 1 respectively. For a $k - digit$ perfect power, the size of the steps taken to output composite2 is between $(k-1) \cdot 10^2$ and $(k+1) \cdot 10^2$.

All but one composite number strictly outputted composite2 in this test. This may be due to the size of the factors because there are only two of them. The one number, 5287, which output composite1 had a factor 17 which is comparably small. This test was efficient with most computations finishing in less than one minute. The problem is, of course, that we can only conclude our number is probably prime when we want to be sure it is. This leads us to the next test.

5.2.2 Modified Gary Miller test

Each composite number outputted composite3 because they failed the Fermat conguence, all with the prime number 2 as the base a. This was expected because the prime factors of the composites are all ≥ 17 so we do not quickly find a divisor. Also, Carmichaels falsely pass the Fermat congruence, but ordinary composite numbers usually fail right away.

The number of steps taken by the prime numbers were significantly larger than that of the composites because of the large nested loops at the end of the test. There is a smaller gap between the steps executed for the primes and the Carmichael numbers, although the primes still dominate.

Carmichael numbers never output composite3 because pseudoprimes pass the Fermat test. We never even found a small divisor for these numbers because even though the prime factors are small for small n, we find a nontrivial square root of 1 modulo n with base prime equal to 2 or 3. We tested numbers up to size 10^{11} (12 digits) and surprisingly only 4 of the Carmichals had to pass to the second prime 3 to discover a nontrivial square root of 1 modulo n. With this said, this code is extremely efficient as all these results were found within seconds.

5.2.3 AKS test by Agrawal, Kayal and Saxena

Overall, this test had the most steps executed for our numbers tested. This is undoubtably due to the large loop our input may enter to see if $(X + a)^n \equiv X^n + a \pmod{X^r - 1, n}$ for $a = 1, \ldots, L$ where L was in the hundreds. Only among the Carmichael numbers did we see more steps taken in the Miller test than AKS for some smaller values of n. The steps for perfect powers were only 10 less than that of the Miller test due to some additional initialization steps before the perfect power loop.

Once I reached a number of size 10^9 (10 digits), I only have two results from my AKS test for Carmichael numbers because of an error I was unable to fix. The error was: *in (quo/poly) integer too large in context*. The test was hung up inside of the large loop mentioned in the above paragraph. For the Carmichael numbers calculated, they always outputted composite2 with a nontrivial divisor of n found less than or equal to the number r such that $o_r(n) > \log^2 n$ and $r \leq \lceil \log^5 n \rceil$. So because Carmichael numbers have at least 3 prime factors, they must be small enough so $gcd(p, n) = p$ and thus output composite2.

At about size 10^6 (7 digits), we can be confident our composite n will enter the large loop in the AKS test. Because we had to stop testing n of size up to 10^8 we found that each of our numbers entering this loop had $a = 1$ being a witness to compositenss and were quickly returned as composite4.

For prime numbers n of the size we have tested, $r \geq n$ is unlikely so in order to declare n prime, it must reach prime2 after the large loop. So AKS gives us the largest number of steps taken by our prime numbers. Again, we ran into the problem of testing these numbers at about size 10^9 because of the previous error.

One last remark is that the loop involving the local variable b in the AKS test was unnecessary. I only realized this after my results indicated none of my numbers output composite3. I reviewed my code and noticed this was the same test used to output composite2.

This test becomes inefficient due to the large loop size at the end and the potential errors. We accept a slower time here than the other two tests though because it is deterministic in polynomial time. This is in contrast to the probabilistic Monte-Carlo test and the Miller test which relies on the ERH to get the desirable polynomial running time.

5.3 Least Squares Fitting to Data

In addition to the observations of the data from the various tests, I also attempted to find a relationship between the numbers tested for primality and the steps counted. I assumed there was a relationship between the number of steps y_i and a constant times a power of the size of our number tested, $\log n_i = x_i$. Then $y_i = B \cdot x_i^A$ so $\log y_i = A(\log x_i) + (\log B)$ or $\log y_i = A(\log(\log n_i)) + (\log B)$. Thus, on the x-axis I plotted $c := \log(\log n_i)$ and on the y-axis, $d := \log y_i$. I decided to partition the numbers for this analysis, not according to size, but according to the type of number: Prime, Carmichael, Composite or Perfect Power. For each type of number, I found a linear relationship between $\log(\log n_i)$ and $\log y_i$ where y_i was the number of steps in MC, Miller or AKS test. Thus, each different type of number has at least three results for the least squares fitting, possibly more than one corresponding to the different outputs in the Monte-Carlo test. The code, plots and the line that best fit can be found in Appendix D. After doing least squares fitting in Maple, I found the linear relationship I was looking for.

In order to transfer excel data directly to Maple, save the excel spreadsheet as a text file. Then import the data into a Maple spreadsheet. From there, copy and paste any columns of data needed into an execution group in Maple. It outputs it as a MATRIX so change it to Vector. Then inside the code, the Vector is converted to a list by deleting the brackets so it can be used inside of the least squares commands in Maple. This seems to be the easiest way to eliminate retyping the data in Maple.

Below are tables with the data from the least squares fitting bringing the constants A and B together from the different tests and types of numbers. We want to compare these results with the numbers given in the theory.

A	Monte-Carlo	Miller	AKS
Prime	1.981111085	2.582225342	3.800596559
Carmichael	.7370043274(c2)	2.154580094	6.346981687
	1.356391454(c3)		
Composite	.9449063256(c2)	2.161482390	7.435119388
Perfect Power	.7217244200(c1)	1.031988969	1.037511555
	.9802826885(c2)		

Table 5.1: Exponent A Values

B	Monte-Carlo	Miller	AKS
Prime	106.5709497	251.8478759	199.3310786
Carmichael	160.3028603(c2)	143.8936999	.005449041142
	29.05648189(c3)		
Composite	54.58766607(c2)	140.0118675	.006039838378
Perfect Power	10.05210029(c1)	119.7964082	117.3495868
	49.78768807(c2)		

Table 5.2: Coefficient B Values

The exponent A given in the theory was about 4 for the Monte-Carlo test once we test $\log n$ different $a's$, 4 for the Miller test and $\frac{21}{2}$ for the AKS test. The exponents A observed above are all less than what we expected; in some cases, much smaller. This may be due to the fact that our step counting was not dependent upon the size of our inputs or has underestimated what we believe Maple is actually computing.

For each test, I looked back at the most costly computation to see how the exponents of $\log n$ differ from the given ones in the theory as a result of my counting conventions. In the Monte-Carlo test, the modular exponentiation was most costly. There are potentially $\mathcal{O}(\log n)$ steps in the powering algorithm to compute $a^{\frac{n-1}{2}} \bmod n$. Each step requires multiplying two numbers and dividing by n to get a remainder modulo n. While counting steps in my code, I considered mulitplying, dividing and reducing a number modulo n as single steps. So overall, this computation only takes $\mathcal{O}(\log n)$ steps. When we complete this for $\log n$ values of a, we get a running time of $\mathcal{O}(\log^2 n)$ steps instead of the given $\mathcal{O}(\log^4 n)$ in the theory.

In the Miller test, we go back to the second step and find another modular exponentiation, this time computing $a^{n-1} \bmod n$ where a varies over the first m primes. Again we used repeated squaring and there are $\mathcal{O}(\log n)$ of these each counted as 1 step. So because there are $\mathcal{O}\left(\frac{f(n)}{\log f(n)}\right)$ primes less than or equal to $f(n)$, we get a running time of $\mathcal{O}(\log^3 n)$. This is in contrast with the exponent of 4 in the theory.

Finally, in the AKS test we look to the largest loop for our complexity. We must verify $\lfloor 2\sqrt{r}\log n \rfloor + 1$ equations. Each equation requires $\mathcal{O}(\log n)$ multiplications of degree r polynomials. In contrast to the theory, my code does not take into account the size of the coefficients and the PApoly depends on $\log n$ and not r when we use the powmod command. This would give us a running time of $\mathcal{O}\tilde{}\,(L \cdot \log n \cdot \log n) = \mathcal{O}\tilde{}\,(\log^{\frac{11}{2}} n)$. This is smaller than the $\frac{21}{2}$ exponent in the theory.

The three new exponents are closer to the actual results we obtained. The largest A found in the Monte-Carlo and Miller tests corresponds to the prime numbers. We expect these to be the most accurate because the primes must run through the largest loops before outputting prime. For the AKS test though, composites give the largest A. The r values for the primes and composites are close, but the Carmichaels give much smaller values of r. This affects the size of the largest loop in the test which only the primes will have to run completely through.

I went back to my code to find a reason for the exponents observed. What I found is that even though the primes have the most steps executed, the majority of those steps actually occur before the large complex loop. The step counts for the composite numbers are larger before entering this loop than those of the prime numbers and the primes do not accumulate as many steps as expected inside the loop. What I believe throws the data off is the lack of a subroutine written for the powmod command in Maple. I think there are not enough steps accounted for in the complexity of this Maple command. Had there been an original procedure written for this, I believe the prime number step counts would have been much higher and the exponent would have surpassed that of the composites and Carmichaels. I am still unsure as to why the Carmichael exponent is so much larger than the prime exponent.

As for the value of B, we get the largest values from the Miller and AKS tests with prime numbers. We expect this. Only in the Monte-Carlo test, the Carmichael numbers give a slightly larger coefficient B than the primes. This I believe is due to the choice of random $a's$. If none of them show n is composite, we have to keep cycling through the test and the coefficient B builds up. We must keep in mind that if a number does not enter the most complex part of these tests, we expect the A and B to be smaller than the theoretical values.

Bibliography

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

[2] M. Agrawal, N. Kayal and N. Saxena, *PRIMES is in P Preprint*, **http://www.iitk.ac.in/news/primality.ps**, August 8, 2002.

[3] N.C.Ankeny, *The least quadratic non-residue*, Ann. of Math. **55** (1952), 65-72.

[4] D.A. Burgess, *The distribution of quadratic residues and non-residues*, Mathematika **4** (1957), 106-112.

[5] R.D. Carmichael, *On composite numbers p which satisfy the Fermat congruence* $a^{p-1} \equiv 1$, Amer. Math. Monthly **19** (1912), 22-27.

[6] Martin Dietzfelbinger, *Primality Testing in Polynomial Time: From Randomized Algorithms to "Primes is in P"*, Springer-Verlag, Germany, 2004.

[7] D.E. Knuth, *The Art of Computer Programming, Vol.2: Seminumerical Algorithms*, Addison-Wesley, Reading, Mass., 1969.

[8] Z.S. McGregor-Dorsey, *Methods of Primality Testing*, MIT Undergraduate Journal of Mathematics **1** (1999), 133-141.

[9] G.M. Miller, *Riemann's Hypothesis and Test for Primality*, J. Comput. Syst. Scie. **13** (1976), 300-317.

[10] J. Radhakrishnan, K. Telikepalli and V. Vinay, *News from India: Primes is in P*, **http://www.tcs.tifr.res.in/ jaikumar/Papers/AKS-revised.pdf**, 2002.

[11] Schönhage and V. Strassen, *Schnelle Multiplication Grosser Zahlen*, Computing **7** (1971), 281-292.

[12] R. Solovay and V. Strassen, *A Fast Monte-Carlo Test for Primality*, SIAM J. Comput., **6** (1977), 84-85.

[13] R. Solovay and V. Strassen, *Erratum: A Fast Monte-Carlo Test for Primality*, SIAM J. Comput., **7** (1978), 118.

[14] Chi-Seng Tou and T. Alexander, *AKS Algorithm*, **http://padic.mathstat.uottawa.ca/ MAT3166/reports/AKS.pdf**, 2005.

[15] Jason Wojciechowski, *The Extended Riemann Hypothesis and its Application to Computing*, **http://wonka.hampshire.edu/ jason/math/comp2/final_paper.pdf**, 2003.

Appendix A

Maple Procedures for the Three Primality Tests

Note. There are some step counting conventions I used that affect the results I obtained. I broke a lot of procedures down and wrote my own, but some things I left to Maple and have consequently been added as a single count to the total. I did not write a procedure for $k() = rand(2..n-1)$ so this was counted as one step. Any return or print steps were not needed for the computation so I left them out of the count. Trunc or floor involves some extra bit operations, but were only counted as one step for my analysis. Computing a logarithm, placing a value in a set (as in the Monte-Carlo test) and creating an array were all counted as single steps. Also, there was no subroutine written for the Maple powmod command.

Lines beginning with # denote comments in Maple. In the Monte-Carlo test, I only tested at most $\log_2(n)$ choices of a. Also, in the Miller test the size of $f(n)$ was not explicit, so I have used $10 \cdot \log^2(n)$ in my calculations. Any subroutines used in these procedures can be found in Appendix B.

A.1 Monte-Carlo test by Solovay and Strassen

```
with(numtheory):
MC:=proc(n)
local a,b,e,j,s,t,k;
global count;

#a is the number inbetween 2 and n-1 we are testing to see if a&^(n-1)/2
#and the jacobi(a/n) are equal mod n;b is the index of a; e=a[b]&^(n-1)/2 mod n;
#j=jacobi(a[b],n); s is the set of all a[b] up until the most recent;
#t is the set of just the last a[b];
#k=rand(2..(n-1));count counts the steps executed in the algorithm

a[0]:=1;
```

```
b:=1;
count:=2;

#here we take account of the simple cases n=1 and n=2;
#otherwise we set a random value to a[1]
if n=1 then
   count:=count+1;
   print(count);
   return "neither";
elif n=2 then
   count:=count+2;
   print(count);
   return "prime";
else k:=rand(2..(n-1));
   a[b]:=k();
   count:=count+5;
end if;

#we initialize our set s simply as a[1] and t as 1 because we know
#a[b] can never equal one because it is not a number in k=rand(2..(n-1));
#this will help us run through the if-then below when we check
#the intersection of s and t
s:={a[1]};
t:={1};
count:=count+4;

#we don't want to test n different numbers for a so we test
#evalf(trunc(log[2](n)) different a's dependent upon n

while b<=evalf(trunc(log[2](n))) do
   count:=count+3;

#if we find a nontrivial gcd, then obviously n has a
#nontrivial factor and is composite
   if gr(a[b],n)<>1 then
      count:=count+1;
      print(count);
      return "composite1";

#test for compositeness of a^((n-1)/2)<>+-1 because a prime number
#has no nontrivial perfect squares
   elif (PA(a[b],(n-1)/2,n))<>1 and (PA(a[b],(n-1)/2,n))<>(n-1) then
      count:=count+4;
      print(count);
      return "composite2";

#if not and s intersect t is empty then we have found a new a[b] and
#so we see if e=j mod n or not
```

```
#we do this so we're not testing the same a[b] over and over again for smaller n
   elif gr(a[b],n)=1 and (s intersect t = {}) then
      count:=count+4;
      count:=count+5;
      e:=PA(a[b],(n-1)/2,n);
      count:=count+1;
      j:=jac(a[b],n);
      count:=count+1;

#if e<>j mod n then n must be composite because for prime n,
#this is an equivalence
         if (e mod n)<>(j mod n) then
            count:=count+3;
            print(count);
            return "composite3";
         end if;
         count:=count+3;

#we move onto the next b, we randomly generate a[b],
#add the last t we had to the set s,
#and then change t to be the new value of a[b]
      b:=b+1; a[b]:=k();
      s:=s union t;
      t:={a[b]};
      count:=count+8;

#if s intersect t is not empty then we want a new value for a[b] so as
#not to repeat any previous calculations for smaller n
#again we set t={a[b]} so we can check the intersection with the new value a[b]
   elif gr(a[b],n)=1 and (s intersect t <> {}) then
      count:=count+5;
      count:=count+9;
      a[b]:=k();
      t:={a[b]};
      count:=count+4;
   end if;
end do;
count:=count+3;

#if we don't find n to be composite then we can only say probably
#prime because we've only tested a select number of a's
print(count);
print ("probably prime");
end proc;
```

A.2 Modified Gary Miller test

```
ModMiller:=proc(n)
local j,k,UB,LB,i,r,f,S,N,Q,x,a,b,m,prime,numprime;
global count;
#j=power of a prime we are testing to see if n is a perfect power of,
#k=prime that n might be a perfect power of; UB=upper bound on the intervals
#we are cutting in half for our binary search for a perfect power, LB=lower bound
#i,r=indexes in the array of n; f(=f(n))=upper bound on the a's that could be
#witnesses to the compositeness of n; S=the max number of 2's that divide n-1;
#Q=(n-1)/2^(S); N=n-1,prime[]=lists of primes; a=index of m; b=index of prime
#x=the various powers of S in the exponent of our primes[a] we're testing in the
#largest part of the test; numprime=number of primes we find <=f

count:=0;
#we need to make sure that f<n
if (trunc(10*((log(n))^2)))>=n then
   count:=count+5;
   f:=n-1;
   count:=count+2;
else count:=count+5;
   f:=trunc(10*((log(n))^2));
   count:=count+5;
end if;

#first step is to test if n is a perfect power; if so we output composite
for j from 2 to floor(evalf(log[2](n))) do
   LB:=1;
   UB:=n;
   count:=count+2;
   while (UB-LB)>1 do
      k:=floor((LB+UB)/2);
      count:=count+6;
         if (PAnomod(k,j))>n then
            count:=count+1;
            UB:=k;
            count:=count+1;
         elif (PAnomod(k,j))<n then
            count:=count+2;
            LB:=k;
            count:=count+1;
         else
            count:=count+2;
            count:=count+(j-1);
            print(count);
            return "composite1";
         end if;
```

```
   end do;
   count:=count+2;
end do;
count:=count+(floor(evalf(log[2](n)))-1);

#if n is not a perfect power, then we start the largest part of the test
#we begin by using the Sieve of Eratosthenes to compute all the primes <=f(=f(n))
#whenever m[b]=0, then b is a prime
m:=array(2..f);
count:=count+2;

for r from 2 to f do
m[r]:=0;
count:=count+1;
end do;
count:=count+(f-1);

#this assigns the number its smallest prime divisor or 0 if it is prime itself
r:=2;
count:=count+1;
while r^2<=f do
   count:=count+2;
   if m[r]=0 then
      i:=r^2;
      count:=count+2;
      while i<=f do
         count:=count+1;
         if m[i]=0
            then m[i]:=r; count:=count+1;
         end if;
         count:=count+1;
         i:=i+r;
      end do;
      count:=count+1;
   end if;
   count:=count+1;
   r:=r+1;
   count:=count+2;
end do;
count:=count+2;

#find S,Q such that n-1=Q*2^(S)
S:=0;
N:=n-1;
count:=count+3;
while (N mod 2)=0 do
   count:=count+2;
```

```
   N:=N/2;
   S:=S+1;
   count:=count+4;
end do;
count:=count+2;

Q:=(n-1)/(PAnomod(2,S));
count:=count+3;

#from all m[b] above, we extract just the prime numbers (when m[b]=0)
#to get a table of primes <=f(n)
numprime:=0;
count:=count+1;
for b from 2 to f do
   if m[b]=0 then
      prime[numprime]:=b;
      numprime:=numprime+1;
      count:=count+3;
   end if;
   count:=count+1;
end do;
count:=count+(f-1);

#here we start the largest part of the test once we have
#found all the primes<=f(=f(n))
 a:=0;
 count:=count+1;
 while a<=numprime do
 count:=count+1;
#a starts at 0 and numprime starts at 1,
#so once a=numprime we have already tested all the primes <=f
    if a=numprime then count:=count+1; return "prime";
#this tests if prime[a] divides n
    elif (n mod prime[a])=0 then count:=count+3;
       return "composite2";
#this is the Fermat test
    elif (PA(prime[a],n-1,n))<>1 then count:=count+4;
       return "composite3";
#here we compute prime[a]^Q mod n and start looking for
#nontrivial square roots of 1 mod n
    else d:=PA(prime[a],Q,n); count:=count+5;
#if S=1, then 1<=x<=S-1=0 doesn't make sense,
#so we make extra steps for a number where S=1
#here, if d=1 or d=n-1 then so do all the squares afterward so we
#just go to our next prime[a]
       if S=1 and (d=1 or d=n-1) then a:=a+1; count:=count+8;
#if not, then because Q*2=n-1 and prime[a]=1 so if d<>1 and d<>n-1,
#then d is a nontrivial square root of 1 and n is composite
```

```
       elif S=1 and (d<>1 and d<>n-1) then count:=count+12;
          return "composite4";
#again, here we find no information so we move onto our next prime[a]
       elif S>=2 and (d=1 or d=n-1) then a:=a+1; count:=count+20;
#because S>=2 we begin to compute d,d^2,d^4...
#looking for nontrivial square roots of 1 mod n
       elif S>=2 then x:=1; count:=count+20;
          while 1<=x and x<=S-1 do count:=count+4;
             d:=PA(d,2,n); count:=count+1;
#no information is obtained
#so we break this while loop and move onto our next prime[a]
             if d=n-1 then a:=a+1; count:=count+4; break;
#the last number was not n-1 or 1 so d=1 gives us a nontrivial square root of 1
             elif d=1 then count:=count+3; return "composite5";
#once we reach x=S-1 and d<>1 and d<>n-1,
#we know prime[a] passes the Fermat test so d is a nontrivial square root of 1
             elif x=S-1 then count:=count+5; return "composite6";
             else x:=x+1; count:=count+7;
             end if;
          end do;
          count:=count+4;
       end if;
    end if;
 end do;
 count:=count+1;
 end proc;
```

A.3 AKS Algorithm by Agrawal, Kayal and Saxena

```
with(numtheory):
AKS:=proc(n)
local j,LB,UB,k,r,L,i,a,b,c;
global count;
#j is our power in the perfect power test with k our base; LB=lower
#bound on the intervals; UB=upper bound;
#r is the smallest positive integer such that the order of n mod r
#is larger than log[2](n)^2 and i is the exponent tested in the loop
#we use b such that 2<=b<=r to see if we can find a nontrivial divisor of n;
#a is the constant coefficient tested in the main loop between 1 and r;
#c=n^i mod r;

count:=0;
#first step is to test if n is a perfect power; if so we output composite
for j from 2 to floor(evalf(log[2](n))) do
   LB:=1;
   UB:=n;
```

```
   count:=count+2;
   while (UB-LB)>1 do
      k:=floor((LB+UB)/2);
      count:=count+6;
         if (PAnomod(k,j))>n then
            count:=count+1;
            UB:=k;
            count:=count+1;
         elif (PAnomod(k,j))<n then
            count:=count+2;
            LB:=k;
            count:=count+1;
         else
            count:=count+2;
            count:=count+(j-1);
            print(count);
            return "composite1";
         end if;

   end do;
   count:=count+2;
end do;
count:=count+(floor(evalf(log[2](n)))-1);

#now we want to find the smallest r such that the order of n mod r is
#larger than log[2](n)^2;
#r is as above and i is the power we raise n to to see if n^i=1 mod r;
#Because i<=trunc((log[2](n))^2), if we cycle through all the i's
#then we have found such an r

r:=2;
i:=1;
count:=count+2;
while i<=trunc((log[2](n))^2) do
   count:=count+4;

#nontrivial gcd with n gives us a divisor of n so it is composite
   if gr(r,n)<>1 and gr(r,n)<>n then
      count:=count+3;
      print(count);
      return "composite2";
   elif r>=n then
      count:=count+4;
      print(count);
      return "prime1";
   else
      count:=count+4;
      if (PA(n,i,r) mod r)=1 then
```

```
        count:=count+2;
        r:=r+1;
        i:=1;
        count:=count+3;
     elif (PA(n,i,r) mod r) <>1 then
        count:=count+4;
        i:=i+1;
        count:=count+2;
     end if;
  end if;
end do;
count:=count+4;

#here we look for a nontrivial divisor of n
#After reviewing the calculations, this loop I found to be
#redundant of the test for composite2 so it may be left out.
for b from 2 to r do
  if gr(b,n)<>1 and gr(b,n)<>n then
     count:=count+3;
     count:=count+(b-1);
     print(count);
     return "composite3";
  end if;
  count:=count+3;
end do;
count:=count+(r-1);

L:=trunc(sqrt(phi(r))*log(n));
print(L);
count:=count+5;

#this is the longest part of the AKS test;
#we can determine compositeness of n here because if n is prime
#then (X+a)^n=X^n+a (mod(X^r-1),n) for all a
for a from 1 to L do
  if (PApoly(X+a,n,r,n))<>(PApoly(X^n+a,1,r,n)) then
     count:=count+1;
     count:=count+a;
     print(count,a);
     return "composite4";
  end if;
  count:=count+1;
end do;
count:=count+L;
#if we don't find compositeness of n in the loop above, our n must be prime
print(count);
print(prime2);
end proc;
```

Appendix B

Maple Procedures used as Subroutines in Primality Tests

B.1 Powering Algorithm

```
PA:=proc(a,k,n)
#computes a^k mod n
local pow, prod, b, i;
global count;
pow:=a;
i:=k;
prod:=1;
count:=count+3;

while i>0 do
   b:=i-2*floor(i/2);
   count:=count+6;
      if b=1 then
         prod:=prod*pow mod n;
         count:=count+3;
      end if;
      count:=count+1;
   pow:=(pow)^2 mod n;
   i:=floor(i/2);
   count:=count+6;
end do;
count:=count+1;
prod;
end proc;
```

B.2 Powering Algorithm with no Modulus

```
PAnomod:=proc(a,k)
#computes a^k with no modulus
local pow, prod, b, i;
global count;
pow:=a;
i:=k;
```

```
prod:=1;
count:=count+3;

while i>0 do
   b:=i-2*floor(i/2);
   count:=count+6;
      if b=1 then
         prod:=prod*pow;
         count:=count+2;
      end if;
      count:=count+1;
   pow:=(pow)^2;
   i:=floor(i/2);
   count:=count+5;
end do;
count:=count+1;
prod
end proc;
```

B.3 Powering Algorithm for Polynomials using powmod

```
PApoly:=proc(f,k,r,n)
#computes f(x)^k mod(x^r-1,n)
local pow, prod, b, i;
global count;
pow:=f;
i:=k;
prod:=1;
count:=count+3;

while i>0 do
   b:=i-2*floor(i/2);
   count:=count+6;
      if b=1 then
         prod:=(powmod(prod*pow,1,X^r-1,X)) mod n;
         count:=count+5;
      end if;
      count:=count+1;
   pow:=(powmod(pow,2,X^r-1,X)) mod n;
   count:=count+5;
   i:=floor(i/2);
   count:=count+3;
end do;
count:=count+1;
prod
end proc;
```

B.4 Greatest Common Divisor

```
gr:=proc(n,m)
#computes gcd(n,m)
local a,b;
global count;

if abs(n)>=abs(m)
   then count:=count+3; a:=abs(n); b:=abs(m);
   count:=count+4;
else count:=count+3; b:=abs(n); a:=abs(m); count:=count+4;
end if;

while b>0 do
   count:=count+1;
   (a,b):=(b,a mod b);
   count:=count+3;
end do;
count:=count+1;

a
end proc;
```

B.5 Jacobi symbol

```
jac:=proc(a,n)
#computes the Jacobi symbol (a/n)
local b,c,s;
global count;

b:=a mod n;
c:=n;
s:=1;
count:=count+4;

while b>=2 do
   count:=count+1;

   while (b mod 4)=0 do
      count:=count+2;
      b:=b/4;
      count:=count+2;
   end do;
   count:=count+2;

   if (b mod 2)=0 then
```

```
      if (c mod 8)=3 or (c mod 8)=5 then
         s:=-s;
         count:=count+2;
      end if;
      count:=count+5;
   b:=b/2;
   count:=count+2;
   end if;
   count:=count+2;

   count:=count+1;
   if b=1 then break;
   end if;

   if (b mod 4)=3 and (c mod 4)=3 then
      s:=-s; count:=count+2;
   end if;
   count:=count+5;
   (b,c):=(c mod b, b);
   count:=count+3;

end do;
count:=count+1;

s*b
end proc;
```

Appendix C

Maple Calculations with Three Primality Tests

n	Test	Digits	Step Count	Output
4603	MC	4	7630;7059;7830;7613;7668	probably prime
	Miller		64331	prime
	AKS		678972	prime2
8147	MC	4	7976;8043;8122;8488;7763	probably prime
	Miller		71696	prime
	AKS		858483	prime2
59239	MC	5	11714;10546;11716;12325;12980	probably prime
	Miller		119763	prime
	AKS		2005525	prime2
72949	MC	5	12162;13071;13422;13451;13284	probably prime
	Miller		133217	prime
	AKS		1850955	prime2
486133	MC	6	16214;16702;17250;16509;17749	probably prime
	Miller		195180	prime
	AKS		3480552	prime2
920393	MC	6	18586;17699;18018;19717;16907	probably prime
	Miller		219360	prime
	AKS		4061589	prime2
2313827	MC	7	22153;22271;21290;22452;21766	probably prime
	Miller		254347	prime
	AKS		5240332	prime2
4203187	MC	7	23596;23055;24445;25197;23695	probably prime
	Miller		274614	prime
	AKS		5752296	prime2

Table C.1: Prime Number Calculations I

n	Test	Digits	Step Count	Output
9373031	MC	7	26877;26241;26561;25985;26089	probably prime
	Miller		318828	prime
	AKS		7111292	prime2
22823263	MC	8	26894;29654;29055;28802;29619	probably prime
	Miller		367211	prime
	AKS		8594656	prime2
72823249	MC	8	35340;33309;35644;33684;35363	probably prime
	Miller		468626	prime
	AKS		10627276	prime2
82823263	MC	8	32868;34113;32890;33915;35644	probably prime
	Miller		451922	prime
	AKS		15079889	prime2
96896249	MC	8	33296;32823;32939;35898;34409	probably prime
	Miller		471191	prime
	AKS		11671429	prime2
124910491	MC	9	34144;34577;34420;35230;35887	probably prime
	Miller		475961	prime
	AKS		13619385	prime2
191454331	MC	9	35458;37763;37362;36844;37633	probably prime
	Miller		504592	prime
	AKS		15542547	prime2
346884011	MC	9	39202;39291;39333;39537;39905	probably prime
	Miller		541166	prime
	AKS		16659513	prime2
592559147	MC	9	42697;40420;40982;42919;39133	probably prime
	Miller		583524	prime
	AKS		20680662	prime2

Table C.2: Prime Number Calculations II

n	Test	Digits	Step Count	Output
2147483647	MC	10	49191;47690;50934;49582;49883	probably prime
	Miller		726428	prime
6781252727	MC	10	52104;51673;50245;51658;50127	probably prime
	Miller		778190	prime
8371854217	MC	10	50349;50431;50436;51086;53282	probably prime
	Miller		808268	prime
9586739249	MC	10	49731;51595;55257;55650;54779	probably prime
	Miller		852989	prime
13346629577	MC	11	54844;56055;54869;52436;52471	probably prime
	Miller		870905	prime
20574592273	MC	11	56425;60602;57698;53404;57569	probably prime
	Miller		934420	prime
33247518979	MC	11	57574;57732;57055;58263;58561	probably prime
	Miller		929573	prime
62710792723	MC	11	61557;62655;62148;59082;62429	probably prime
	Miller		985628	prime
120234043603	MC	12	64997;62593;64716;65470;63904	probably prime
	Miller		1075864	prime
242113332463	MC	12	66900;65437;68113;67239;69616	probably prime
	Miller		1145493	prime
598267465151	MC	12	75585;74468;73301;77310;73263	probably prime
	Miller		1271340	prime
999991632797	MC	12	77420;75567;74111;72225;76158	probably prime
	Miller		1329191	prime

Table C.3: Prime Number Calculations III

n	Test	Digits	Factors	Step Count	Output
2821	MC	4	7, 13, 31	368;483;47;1816;549	c2;c1;c1;c2;c3
	Miller			12454	c6; prime=2
	AKS			7047	c2
6601	MC	4	7, 23, 41	562;578;59;406;414	c3;c1;c1;c2;c2
	Miller			15713	c5; prime=2
	AKS			9044	c2
29341	MC	5	13, 37, 61	43;1337;1311;482;482	c1;c2;c2;c2;c2
	Miller			22215	c5; prime=3
	AKS			13678	c2
46657	MC	5	13, 37, 97	588;596;729;1349;698	c3;c3;c3;c3;c3
	Miller			23699	c5; prime=2
	AKS			15961	c2
334153	MC	6	19, 43, 409	825;592;2159;596;596	c3;c2;c2;c2;c2
	Miller			34621	c5; prime=2
	AKS			23473	c2
512461	MC	6	13, 61, 271	1340;1356;620;721;786	c2;c2;c2;c3;c3
	Miller			36437	c6; prime=2
	AKS			35755	c2
6733693	MC	7	109, 163, 379	918;969;778;766;1779	c3;c3;c2;c2;c2
	Miller			54604	c6; prime=2
	AKS			214662	c2
53711113	MC	8	157, 313, 1093	1078;2165;816;796;2832	c3;c3;c2;c2;c2
	Miller			72104	c5; prime=3
	AKS			629268	c2
84350561	MC	8	107, 743, 1061	2070;5674;1098;2170;1081	c3;c3;c3;c3;c3
	Miller			75598	c5; prime=2
	AKS			227235	c2
96895441	MC	8	109, 433, 2053	2997;3389;828;1974;1171	c2;c1;c2;c2;c3
	Miller			75825	c5; prime=2
	AKS			274960	c2

Table C.4: Carmichael Number Calculations I

n	Test	Digits	Factors	Step Count	Output
114910489	MC	9	127, 659, 1373	1089;1169;1218;2228;2391	c3;c3;c3;c3;c1
	Miller			77234	c5; prime=2
	AKS			311869	c2
171454321	MC	9	163, 811, 1297	2061;3228;1077;75;878	c2;c2;c3;c1;c2
	Miller			81752	c5; prime=2
	AKS			624792	c2
221884001	MC	9	131, 521, 3251	1042;3476;4485;2345;3620	c3;c3;c3;c3;c3
	Miller			83849	c5; prime=2
	AKS			374094	c2
492559141	MC	9	367, 733, 1831	928;2154;1232;1275;1178	c2;c2;c3;c3;c3
	Miller			91191	c6; prime=2
	AKS			2736045	c2
863984881	MC	9	307, 613, 4591	1280;3800;1254;1275;1297	c1;c3;c3;c3;c3
	Miller			96890	c5; prime=2
	AKS			1754577	c2
1260332137	MC	10	163, 487, 15877	5128;3800;1300;1241;1265	c3;c3;c3;c3;c3
	Miller			100920	c5; prime=2
	AKS			680245	c2
5781222721	MC	10	1033, 1549, 3613	1314;1330;1358;1361;1404	c3;c3;c3;c3;c3
	Miller			116947	c5; prime=2
	AKS			23366636	c2
8251854001	MC	10	1301, 1951, 3251	4089;2727;1397;2787;1372	c3;c3;c3;c3;c3
	Miller			120332	c5; prime=2
8652633601	MC	10	1249, 2081, 3329	1336;2788;2792;5390;2764	c3;c3;c3;c3;c3
	Miller			123306	c5; prime=2
9086767201	MC	10	1201, 1801, 4201	1368;1314;3958;2725;4012	c3;c3;c3;c3;c3
	Miller			123137	c5; prime=2

Table C.5: Carmichael Number Calculations II

n	Test	Digits	Factors	Step Count	Output
11346205609	MC	11	1237, 2473, 3709	1438;2378;1050;2468;1384	c3;c2;c2;c2;c3
	Miller			125323	c5; prime=2
12456671569	MC	11	1013, 3037, 4049	1392;1064;1080;1341;1064	c3;c2;c2;c3;c2
	Miller			125543	c5; prime=2
14313548881	MC	11	1061, 3181, 4241	2481;1423;1084;2524;2446	c2;c3;c2;c2;c2
	Miller			127735	c5; prime=2
16157879263	MC	11	1667, 2143, 4523	1064;1104;1104;1076;1080	c2;c2;c2;c2;c2
	Miller			128381	c4; prime=2
23224518901	MC	11	1901, 3301, 3701	1078;1074;2523;3024;1090	c2;c2;c2;c2;c2
	Miller			134549	c6; prime=3
40999665001	MC	11	1021, 3001, 13381	1507;2580;2673;1108;1520	c3;c2;c2;c2;c3
	Miller			142177	c5; prime=3
56718791641	MC	11	1237, 2473, 18541	1550;1134;2649;1380;2845	c3;c2;c2;c3;c3
	Miller			144512	c5; prime=2
73543985857	MC	11	1453, 4357, 11617	2889;2853;2931;4652;1469	c3;c3;c3;c3;c3
	Miller			150214	c5; prime=3
100264053529	MC	12	2557, 5113, 7669	2699;4070;1439;6046;2665	c2;c2;c3;c3;c2
	Miller			151681	c5; prime=2
136368172081	MC	12	1657, 3313, 24841	4559;1142;3028;3036;1146	c3;c2;c3;c3;c2
	Miller			154444	c5; prime=2
172113632461	MC	12	2791, 5023, 12277	3142;1204;1184;1160;2765	c3;c2;c2;c2;c2
	Miller			159577	c6; prime=2
342267565249	MC	12	3019, 7043, 16097	3205;3278;3128;1630;1605	c3;c3;c3;c3;c3
	Miller			168992	c5; prime=2
635681188801	MC	12	1009, 20161, 31249	1220;1608;1220;1216;1212	c2;c3;c2;c2;c2
	Miller			178092	c5; prime=2
846891632791	MC	12	1667, 4999, 101627	1230;1234;2894;1270;1246	c2;c2;c2;c2;c2
	Miller			182262	c4; prime=2

Table C.6: Carmichael Number Calculations III

n	Test	Digits	Factors	Step Count	Output
5287	MC	4	7, 311	410;43;406;410;398	c2;c1;c2;c2;c2
	Miller			14601	c3; prime=2
	AKS			12424	c2
9211	MC	4	61, 151	454;450;454;458;454	c2;c2;c2;c2;c2
	Miller			16787	c3; prime=2
	AKS			77827	c2
37327	MC	5	163, 229	528;500;504;516;516	c2;c2;c2;c2;c2
	Miller			22887	c3; prime=2
	AKS			682745	c2
61133	MC	5	133, 541	540;516;508;532;524	c2;c2;c2;c2;c2
	Miller			24476	c3; prime=2
	AKS			309116	c2
226679	MC	6	419, 541	596;600;576;572;604	c2;c2;c2;c2;c2
	Miller			31601	c3; prime=2
	AKS			3197817	c4; a=1
604033	MC	6	137, 4409	624;632;624;620;628	c2;c2;c2;c2;c2
	Miller			38232	c3; prime=2
	AKS			466649	c2
3029053	MC	7	1321, 2293	698;702;678;706;690	c2;c2;c2;c2;c2
	Miller			47957	c3; prime=2
	AKS			6017736	c4; a=1
5510053	MC	7	1543, 3571	736;724;704;728;704	c2;c2;c2;c2;c2
	Miller			52676	c3; prime=2
	AKS			6256482	c4; a=1
7023449	MC	7	1997, 3517	738;722;702;734;726	c2;c2;c2;c2;c2
	Miller			54420	c3; prime=2
	AKS			6471654	c4; a=1

Table C.7: Composite Number Calculations I

n	Test	Digits	Factors	Step Count	Output
10000043	MC	8	4787, 2089	750;754;738;738;754	c2;c2;c2;c2;c2
	Miller			57434	c3; prime=2
	AKS			7799329	c4; a=1
67029583	MC	8	20269, 3307	838;854;854;862;838	c2;c2;c2;c2;c2
	Miller			71624	c3; prime=2
	AKS			13899043	c4; a=1
90028349	MC	8	1013, 88873	878;862;862;878;894	c2;c2;c2;c2;c2
	Miller			75252	c3; prime=2
	AKS			13892277	c4; a=1
115710557	MC	9	5039, 22963	884;840;868;872;864	c2;c2;c2;c2;c2
	Miller			77059	c3; prime=2
	AKS			14670758	c4; a=1
281894243	MC	9	116341, 2423	918;934;898;890;922	c2;c2;c2;c2;c2
	Miller			86520	c3; prime=2
	AKS			17080632	c4; a=1
723001537	MC	9	161999, 4463	914;930;918;930;922	c2;c2;c2;c2;c2
	Miller			94709	c3; prime=2
	AKS			23327888	c4; a=1
900924397	MC	9	91159, 9883	940;960;968;960;976	c2;c2;c2;c2;c2
	Miller			97025	c3; prime=2
	AKS			19824094	c4; a=1

Table C.8: Composite Number Calculations II

n	Test	Digits	Factors	Step Count	Output
1000	MC	4	10	47;43;324;39;31	c1;c1;c2;c1;c1
	Miller			978	c1
	AKS			968	c1
6859	MC	4	19	432;43;416;436;420	c2;c1;c2;c2;c2
	Miller			1256	c1
	AKS			1246	c1
12167	MC	5	23	460;464;59;456;464	c2;c2;c1;c2;c2
	Miller			1329	c1
	AKS			1319	c1
85184	MC	5	44	536;59;532;59;71	c2;c1;c2;c1;c1
	Miller			1384	c1
	AKS			1374	c1
157464	MC	6	54	568;568;47;55;75	c2;c2;c1;c1;c1
	Miller			1599	c1
	AKS			1589	c1
830584	MC	6	94	656;63;644;652;79	c2;c1;c2;c2;c1
	Miller			1992	c1
	AKS			1982	c1
1000000	MC	7	100	616;71;59;636;51	c2;c1;c1;c2;c1
	Miller			896	c1
	AKS			886	c1
8120601	MC	7	201	63;744;772;83;724	c1;c2;c2;c1;c2
	Miller			2060	c1
	AKS			2050	c1
32768000	MC	8	320	840;75;856;828;832	c2;c1;c2;c2;c2
	Miller			2347	c1
	AKS			2337	c1
79507000	MC	8	430	870;866;55;850;67	c2;c2;c1;c2;c1
	Miller			2633	c1
	AKS			2623	c1

Table C.9: Perfect Power Number Calculations I

n	Test	Digits	Factors	Step Count	Output
341532099	MC	9	699	936;111;952;896;924	c2;c1;c2;c2;c2
	Miller			2572	c1
	AKS			2562	c1
611960049	MC	9	849	103;940;912;960;95	c1;c2;c2;c2;c1
	Miller			2763	c1
	AKS			2753	c1
3716672149	MC	10	1549	1024;1012;1020;1044;1036	c2;c2;c2;c2;c2
	Miller			2987	c1
	AKS			2977	c1
7483530816	MC	10	1956	1072;95;1072;83;87	c2;c1;c2;c1;c1
	Miller			3004	c1
	AKS			2994	c1
32768000000	MC	11	3200	1140;99;119;99;1156	c2;c1;c1;c1;c2
	Miller			3206	c1
	AKS			3196	c1
84027672000	MC	11	4380	99;99;1162;115;95	c1;c1;c2;c1;c1
	Miller			3463	c1
	AKS			3453	c1
494725990429	MC	12	7909	1226;91;1270;1246;1230	c2;c1;c2;c2;c2
	Miller			3693	c1
	AKS			3683	c1
710687513024	MC	12	8924	1282;1278;127;1258;1270	c2;c2;c1;c2;c2
	Miller			3934	c1
	AKS			3924	c1

Table C.10: Perfect Power Number Calculations II

Appendix D

Maple Least Squares Fitting

D.1 Maple code for least squares fitting

```
LSQ:=proc(x,y)
 local X,Y,data,dataplot,lsqcurve,f;
 X:=convert(x,list);
 Y:=convert(y,list);
 data:=X,Y;
 dataplot:=scatterplot(data,symbol=cross,symbolsize=14,color=black):
 f:=fit[leastsquare[[c,d],d=a*c+b,{a,b}]]([data]);
 lsqcurve:=plot(rhs(f),c=0..3.5,color=gold):
 print(f);
 display(lsqcurve,dataplot);
 end proc:
```

D.2 Prime Numbers Least Squares Plots and Lines

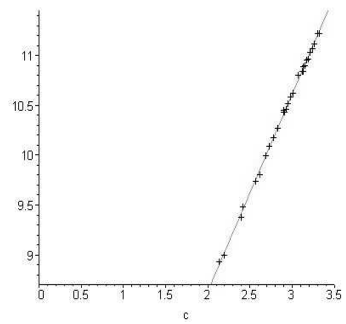

Figure D.1: Monte-Carlo Least Squares with Primes

$$d = 1.981111085 * c + 4.668810958$$

$$A = 1.981111085$$

$$B = e^{4.668810958} \approx 106.5709497$$

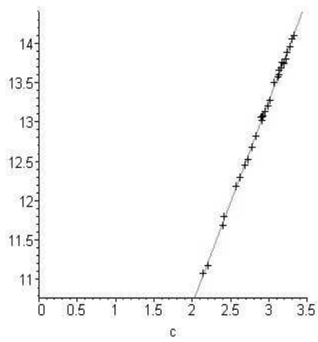

Figure D.2: Miller Least Squares with Primes

$$
\begin{aligned}
d &= 2.582225342 * c + 5.528825238 \\
A &= 2.582225342 \\
B &= e^{5.528825238} \approx 251.8478759
\end{aligned}
$$

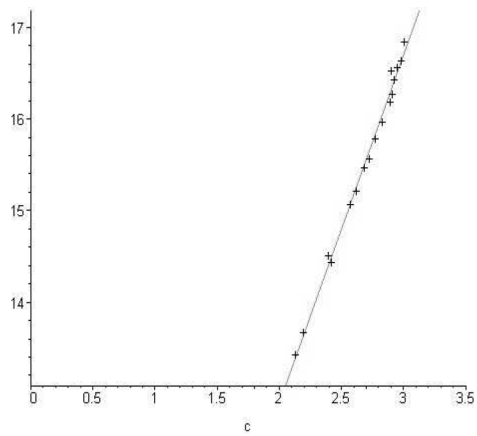

Figure D.3: AKS Least Squares with Primes

$$
\begin{aligned}
d &= 3.800596559 * c + 5.294967154 \\
A &= 3.800596559 \\
B &= e^{5.294967154} \approx 199.3310786
\end{aligned}
$$

D.3 Carmichael Numbers Least Squares Plots and Lines

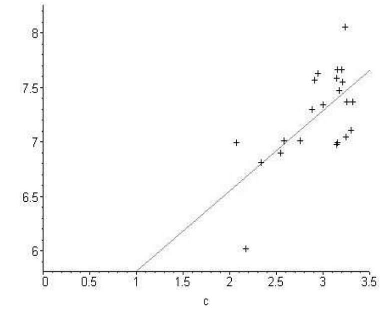

Figure D.4: Monte-Carlo composite2 outputted Least Squares with Carmichaels

$$
\begin{aligned}
d &= .7370043274 * c + 5.077064903 \\
A &= .7370043274 \\
B &= e^{5.077064903} \approx 160.3028603
\end{aligned}
$$

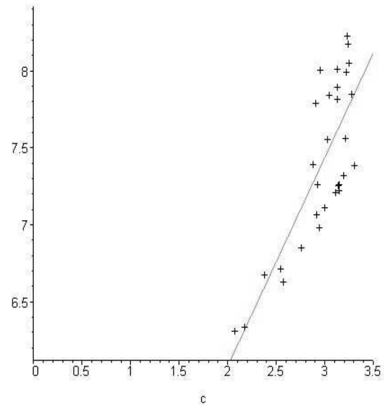

Figure D.5: Monte-Carlo composite3 outputted Least Squares with Carmichaels

$$
\begin{aligned}
d &= 1.356391454 * c + 3.369241587 \\
A &= 1.356391454 \\
B &= e^{3.369241587} \approx 29.05648189
\end{aligned}
$$

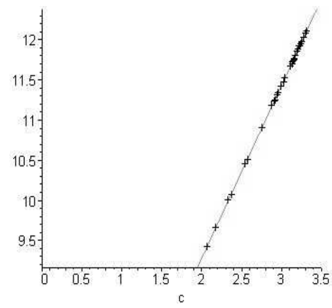

Figure D.6: Miller Least Squares with Carmichaels

$$d = 2.154580094 * c + 4.969074832$$
$$A = 2.154580094$$
$$B = e^{4.969074832} \approx 143.8936999$$

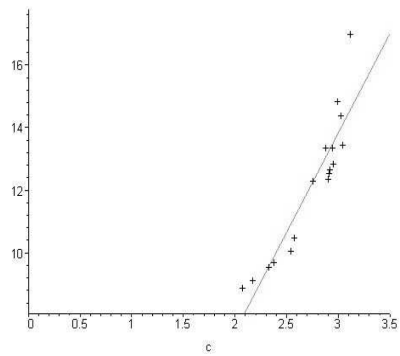

Figure D.7: AKS Least Squares with Carmichaels

$$d = 6.346981687 * c - 5.212315623$$
$$A = 6.346981687$$
$$B = e^{-5.212315623} \approx .005449041142$$

D.4 Composite Numbers Least Squares Plots and Lines

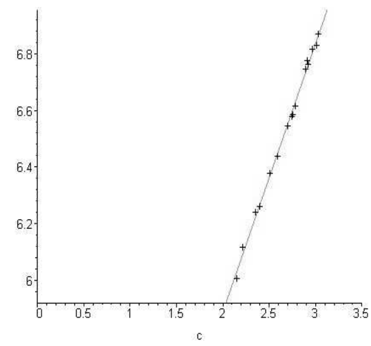

Figure D.8: Monte-Carlo Least Squares with Composites

$$\begin{aligned} d &= .9449063256 * c + 3.999807961 \\ A &= .9449063256 \\ B &= e^{3.999807961} \approx 54.58766607 \end{aligned}$$

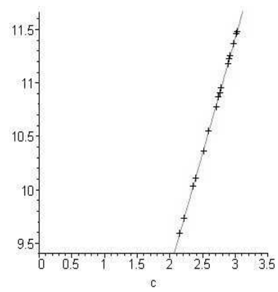

Figure D.9: Miller Least Squares with Composites

$$\begin{aligned} d &= 2.161482390 * c + 4.941727187 \\ A &= 2.161482390 \\ B &= e^{4.941727187} \approx 140.0118675 \end{aligned}$$

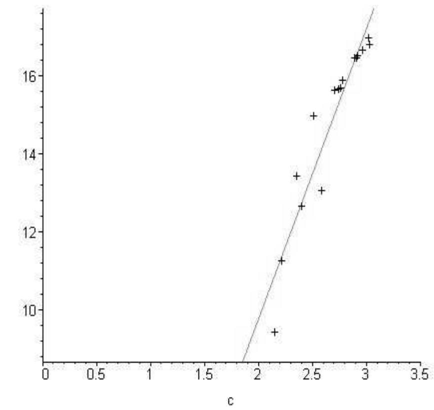

Figure D.10: AKS Least Squares with Composites

$$\begin{aligned} d &= 7.435119388 * c - 5.109378026 \\ A &= 7.435119388 \\ B &= e^{-5.109378026} \approx .006039838378 \end{aligned}$$

D.5 Perfect Power Numbers Least Squares Plots and Lines

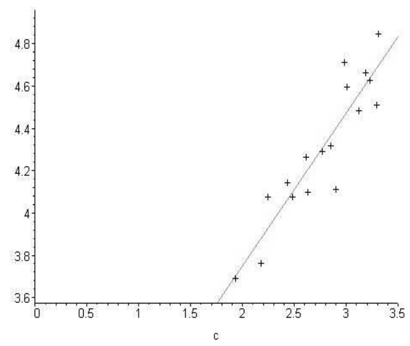

Figure D.11: Monte-Carlo composite1 outputted Least Squares with Perfect Powers

$$
\begin{aligned}
d &= .7217244200 * c + 2.307781597 \\
A &= .7217244200 \\
B &= e^{2.307781597} \approx 10.05210029
\end{aligned}
$$

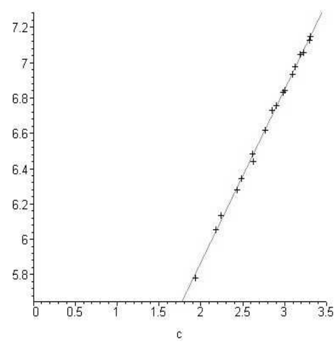

Figure D.12: Monte-Carlo composite2 outputted Least Squares with Perfect Powers

$$
\begin{aligned}
d &= .9802826885 * c + 3.907767726 \\
A &= .9802826885 \\
B &= e^{3.907767726} \approx 49.78768807
\end{aligned}
$$

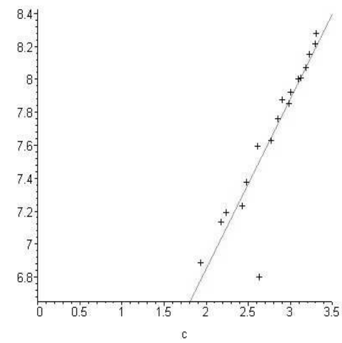

Figure D.13: Miller Least Squares with Perfect Powers

$$
\begin{aligned}
d &= 1.031988969 * c + 4.785793704 \\
A &= 1.031988969 \\
B &= e^{4.785793704} \approx 119.7964082
\end{aligned}
$$

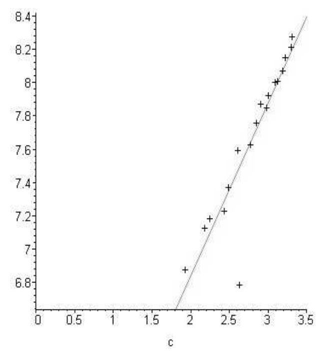

Figure D.14: AKS Least Squares with Perfect Powers

$$
\begin{aligned}
d &= 1.037511555 * c + 4.765157401 \\
A &= 1.037511555 \\
B &= e^{4.765157401} \approx 117.3495868
\end{aligned}
$$