Research Project for Fall 2004
Advisor: Aaron Wootton

STUDY OF "PRIMES IS IN P"

J. A. GILBERT

Abstract. This paper is an in depth study of "Primes is in P." It details the proofs presented in [1], and includes a proof of the time complexity. It also explains how the algorithm was implemented in various programming languages. We finish by laying out future plans for how to improve the algorithm and further work to be investigated.

1. Introduction

The problem of whether a number is prime as well as the question of how to factor numbers have been of long standing interest to mathematicians. New discoveries in such a field would not seem to come often. However, since the invention and general use of computers, many new algorithms have been developed in order to speed up the process of determining primality and factoring large numbers. The problem has now become to find a way to determine primality or factorize that is not in exponential time, but in polynomial time. One of the biggest breakthroughs of recent time is the algorithm by Agrawal, Kayal, and Saxena, three computer science engineers at the Indian Institute of Technology Kanpur see [1]. In this paper, the three authors develop an algorithm which determines primality in polynomial time.

In this paper, we explore the algorithm in depth. We start by examining the preliminary results in detail, many of the proofs which were omitted from the original paper. We then move on to prove the time complexity which shows that the algorithm works in polynomial time. Next we demonstrate how to implement the algorithm in Matlab and our attempts with other programming languages. We finally propose future lines of research on this subject and how we plan to speed up the algorithm.

2. Preliminaries

In this section, we examine the preliminary results presented in [1], in detail. We include all proofs for completeness. As a note on notation we use $\log x$ to mean the natural logarithm, $\ln x$.

The primality test is based on the following identity for prime numbers.

Lemma 2.1. *Suppose a is coprime to p. Then, p is prime if and only if*

$$(x-a)^p \equiv (x^p - a) \mod p.$$

Proof. For $0 < i < p$ the coefficient of x^i in $(x-a)^p - (x^p - a)$ is $(-1)\begin{pmatrix} p \\ i \end{pmatrix} a^{p-i}$. If p is prime $\begin{pmatrix} p \\ i \end{pmatrix} = \frac{p!}{i!(p-i)!}$ is always divisible by p since it is the largest prime factor

of the numerator and is not a factor of the denominator. Therefore, $\left(\begin{array}{c} p \\ i \end{array} \right) \equiv 0 \mod p$.

If p is composite, let q be a prime factor of p and suppose q^k is the greatest factor of p. This implies that q^k does not divide $\left(\begin{array}{c} p \\ q \end{array} \right)$, since q^k is the greatest factor of p, and p has already been divided by q so at most q^{k-1} may divide $\left(\begin{array}{c} p \\ q \end{array} \right)$. Therefore, q^k is coprime to a^{p-q}, since q is the factor of p and a is coprime to p, so any power of q to any power of a is coprime. Therefore, the coefficient $x^q \neq 0 \mod p$. Therefore, $(x-a)^p - (x^p - a)$ is not identical to zero over $\mathbb{F}_p$.

□

Given any p input, it is possible to choose a polynomial $P(x) = x - a$ and compute whether or not the identity holds. Unfortunately, this requires time Ωp [1]. To make the algorithm run in polynomial time, both sides of the above identity are evaluated at modulo a polynomial $x^r - 1$. Thus the algorithm will consist of evaluating whether or not

$$(x-a)^p \equiv (x^p - a)(\mod x^r - 1, p).$$

From the identity above it is apparent that all primes p satisfy the above congruence for all values a and r. However, some composites, may also satisfy the identity for a few values of a and r. The identity above will take $O(r \log^2 p)$ if Fast Fourier Multiplication is used [1].

Lemma 2.2. *$(p^k - 1)|(p^d - 1)$, if and only if $k|d$.*

Proof. By performing long division it can be shown that if $(p^k - 1)|(p^d - 1)$, then $k|d$.

$$\begin{array}{r|l} p^k - 1 & p^{d-k} + p^{d-2k} + p^{d-3k} + \cdots + p^{d-nk} + 1 \\ \hline p^d & -1 \\ \underline{-p^d + p^{d-k}} & \\ p^{d-k} & \\ \underline{-p^{d-k} + p^{d-2k}} & \\ p^{d-2k} & \\ \underline{-p^{d-2k} + p^{d-3k}} & \\ \vdots & \\ p^{d-nk} - 1 & \\ \underline{-p^{d-nk} + 1} & \\ 0 & \end{array}$$

The long division must have an end since the elements are finite. So when n, the nth and final element in the long division, is reached if there is to be no remainder then $(p^k - 1)|(p^d - 1)$ must be true. However, if there is a remainder, then $(p^k - 1)$ cannot divide $(p^d - 1)$. When there is no remainder then the nth term of division will give that $p^{d-nk} = 1$, thus $d - nk = 0$, so $d = n * k$ or $\frac{d}{k} = n$. Since n is an integer $k|d$.

□

Lemma 2.3. *Let p and r be prime numbers, $p \neq r$.*

1. *The multiplicative group of any field $\mathbb{F}_{p^t}$ for $t > 0$, denoted by $\mathbb{F}^*_{p^t}$ is cyclic.*

2. Let $f(x)$ be a polynomial with integral coefficients. Then

$$f(x)^p \equiv f(x^p) \mod p$$

3. Let h(x) be any factor of $x^r - 1$. Let $m \equiv m_r \mod p$. Then

$$x^m \equiv x^{m_r} \mod h(x)$$

4. Let $o_r(p)$ be the order of p modulo r. Then in $\mathbb{F}_p$, $\frac{x^r-1}{x-1}$ factorises into irreducible polynomials each of degree $o_r(p)$.

Proof. 1. (See [6] page 34-35.)

2. Let $f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$

$$f(x)^p \equiv f(x^p)(\mod p)$$

$$f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

$$f(x^p) = a_0 + a_1x^p + a_2x^{2p} + \cdots + a_nx^{np}$$

$$(f(x))^p = (a_0 + a_1x + a_2x^2 + \cdots + a_nx^n)^p$$

The proof follows a general pattern which can be illustrated by assuming $f(x)$ is linear. Assuming this, we get

$$(a_o + a_ix)^p$$

$$= a_o^p(-1)\begin{pmatrix} p \\ 0 \end{pmatrix} + a_0a_1x(-1)\begin{pmatrix} p \\ 1 \end{pmatrix} + a_0^{p-2}a_1^2x^2(-1)\begin{pmatrix} p \\ 2 \end{pmatrix} + \cdots + a_1^px^p(-1)\begin{pmatrix} p \\ p \end{pmatrix}$$

$$= a_0^p(1) + \cdots + \frac{p!}{i!(p-i)!}a_0^{p-i}a_1^ix^i + \cdots + a_1^px^p$$

Reducing modulo p all coefficients except its first and last are divisible by p, so we get

$$(a_0 + a_1x)^p = a_0^p + a_1^px^p$$

By Fermat's Little Theorem, we get

$$a^p = a \mod p.$$

$$\text{and so } a^p - a = 0 \mod p$$
$$\text{Hence } f(x)^p \equiv f(x^p) \mod p$$

3. Let $m = kr + m_r$. Then,

$$x^r \equiv 1 \mod x^r - 1$$

$$x^{kr} \equiv 1 \mod x^r - 1$$

$$x^{m_r}x^{kr} \equiv x^{m_r} \mod x^r - 1$$

$$x^{kr+m_r} \equiv x^{m_r} \mod x^r - 1$$

$$x^m \equiv x^{m_r} \mod h(x).$$

4. Let $d = o_r(p)$ and $Q_r(x) = \frac{x^r-1}{x-1}$. Suppose that $Q_r(x)$ has an irreducible factor, $h(x)$ in $\mathbb{F}_p$ of degree k. Now $\mathbb{F}_p[x]/h(x)$ is a field of size p^k and the multiplicative subgroup of $\mathbb{F}_p[x]/h(x)$ is cyclic with a generator $g(x)$. Also, in this field we have:

$$g(x)^p \equiv g(x^p)$$

$$g(x)^{p^d} \equiv g(x^{p^d})$$

By (3) above, we have

$$g(x)^{p^d} \equiv g(x)$$

$$g(x)^{p^d-1} \equiv 1$$

Since $(p^k - 1)$ is the order of $g(x)$ and since the order of any element in a group divides the group order, we get $(p^k - 1)|(p^d - 1)$, and from Lemma 2.2 this implies $k|d$. We also have that $h(x)|(x^r - 1)$ in $\mathbb{F}_p$ and therefore in the field $\mathbb{F}_p[x]/h(x)$ we have

$$x^r \equiv 1.$$

Thus the order of x in this field must be r, since r is prime and $x \not\equiv 1$. Therefore, $r|(p^k - 1)$, or $p^k \equiv 1(\mod r)$. Thus, $d|k$. Since $k|d$ and $d|k$ then $k = d$, and so each of the irreducible polynomials must be of degree $o_r(p)$.

□

Lemma 2.4. *Let $P(n)$ denote the greatest prime divisor of n. There exists $c > 0$ and n_0 such that, for all $x \geqslant n_0$*

$$|\{p|p \text{ is prime}, p \leqslant x \text{ and } P(p-1) > x^{2/3}\}| \geqslant c\frac{x}{\log x}$$

Proof. See [4].

□

Lemma 2.5. *Let $\pi(n)$ be the number of primes $\leqslant n$. Then for $n \geqslant 1$:*

$$\frac{n}{6\log n} \leqslant \pi(n) \leqslant \frac{8n}{\log n}$$

Proof. See [2]. □

3. AKS Algorithm

Input: integer $n > 1$

1. if (n is of the form a^b, $b > 1$) output COMPOSITE;
2. r = 2;
3. while ($r < n$) {
4. if (gcd(n,r) $\neq 1$) output COMPOSITE;
5. if (r is prime)
6. let q be the largest prime factor of $r - 1$
7. if ($q \geqslant 4\sqrt{r}\log n$) and ($n^{\frac{r-1}{q}} \not\equiv 1 \mod r$)
8. break;
9. $r = r + 1$
10.}
11. for $a = 1$ to $2\sqrt{r}\log n$
12. if ($(x-a)^n \not\equiv (x^n - a)(\mod x^r - 1, n)$) output COMPOSITE;
13. output PRIME;

Theorem 3.1. *The algorithm above returns PRIME if and only if n is prime.*

In this section, we establish through a series of lemmas that the above theorem is true. We start by examining the first loop.

Lemma 3.2. *There exists positive constants c_1 and c_2 for which there is a prime r in the interval $[c_1(\log n)^6, c_2(\log n)6]$ such that $r-1$ has a prime factor $q \geqslant 4\sqrt{r}\log n$ and $q|o_r(n)$.*

Proof. Let c and $P(n)$ be the same as in Lemma 2.4. This implies that the number of prime r's, from now on called special primes, between $c_1(\log n)^6$ and $c_2(\log n)^6$ such that $P(r-1) > (c_2(\log n)^6)^{\frac{2}{3}} > r^{\frac{2}{3}}$ is (for large enough n)

$$\geqslant \text{ Number of } r\text{'s in } [1 \cdots c_s(\log n)^6] - \text{ Number of } n\text{'s in } [1 \cdots c_1(\log n)^6$$

Using Lemma 2.4 and 2.5

$$= \frac{cx_2}{\log x_2} - \frac{8x_1}{\log x_1}$$

let $x_2 = c_2(\log n)^6$ and $x_1 = c_1(\log n)^6$

$$\geqslant \frac{cc_2(\log n)^6}{\log c_2(\log n)^6} - \frac{8c_1(\log n)^6}{\log c_1(\log n)^6}$$

$$= \frac{cc_2(\log n)^6}{\log c_2 + 6\log\log n} - \frac{8c_1(\log n)^6}{6\log\log n}$$

$$\geqslant \frac{cc_2(\log n)^6}{7\log\log n} - \frac{8c_1(\log n^6)}{6\log\log n}$$

$$= \frac{(\log n)^6}{\log\log n}(\frac{cc_2}{7} - \frac{8c_1}{6})$$

Choose constants $c_1 \geqslant 4^6$ and c_2 so that the quantity $(\frac{cc_2}{7} - \frac{8c_1}{6})$ is a positive constant, which we will call c_3.

Let $x = c_2(\log n)^6$. Consider the product

$$\prod = (n-1)(n^2-1)\cdots(n^{x^{\frac{1}{3}}}-1)$$

$$n = p_1^{n_1} \cdots P_r^{n_r}$$

$$\log n = n_1 \log p_1 + n_2 \log p_2 + \cdots + n_r \log p_r$$

$$\log n \geqslant \log p_1 + \log p_2 + \cdots + \log p_r$$

$$\log p_1 > 1 \text{ for } p_1 > 2$$

If $p_1 \neq 2$ then $\log n > 1 + 1 + \cdots + 1 =$ number of primes dividing n. Therefore, $\log(n) >$ the number of primes provided that n is sufficiently large.

$$\prod = n^{1+2+\cdots+x^{\frac{1}{3}}}$$

$1 + 2 + \cdots + x^{\frac{1}{3}} \leqslant x^{\frac{2}{3}}$

$$\prod \leqslant n^{x^{\frac{2}{3}}}$$

This product has at most $x^{\frac{2}{3}} \log n$ prime factors.

$$\log n^{x^{\frac{2}{3}}} = x^{\frac{2}{3}} \log n$$

$$x^{\frac{2}{3}} \log n < \frac{c_3(\log n)^6}{\log\log n}$$

This implies that there is at least one special prime r that does not divide the product Π. Now we want to show that $q | o_r(n)$.

We know that:

$$\exists \text{special prime } r \text{ such that } r \not| \Pi$$

$$\exists q,\ q|(r-1) \text{ and } q \geqslant r^{\frac{2}{3}}$$

$$o_r(n) > x^{\frac{1}{3}} \geqslant r^{\frac{2}{3}}$$

$$o_r(n)|(r-1)$$

This implies

$$(r-1) = q(p_1 \cdots p_n)$$

$$\text{We know } q \geqslant r^{\frac{2}{3}} \text{ this implies } (p_1 \cdots p_n) \leqslant r^{\frac{1}{3}}.$$

$$\text{Therefore, } q|o_r(n).$$

This is the required prime $r-1$ that has a large prime factor $q \geqslant r^{\frac{2}{3}} \geqslant 4\sqrt{r}\log n$ because $c_1 \geqslant 4^6$ and $q|o_r(n)$. □

Since we now know that the while loop will halt, we can no show:

Lemma 3.3. *If and only if n is prime, the algorithm returns PRIME.*

Proof. The while loop cannot return COMPOSITE since $\gcd(n,\ r) = 1$ for all $r \leqslant c_2(\log n)^6$, where c_2 is from Lemma 3.2. We also know that from Lemma 2.3 (fact 2) that the for loop cannot return COMPOSITE. These facts combined mean that the algorithm cannot return COMPOSITE and must return PRIME.

Now we will examine what happens if the input n, into the algorithm, is a composite. The r found in the while loop when n is composite with $p_i, 1 \leqslant i \leqslant k$, as its prime factors. In this case $o_r(n)|\text{lcd}_i\{o_r(p_i)\}$ and this implies there exists a prime factor p of n such that $q|o_r(p)$, where q is the largest prime factor of $r-1$. For the rest of this argument, let p be such a prime factor of n.

The for loop of the algorithm uses the value of r obtained from the while loop to do polynomial computation on $l = 2\sqrt{r}\log n$ binomials: $(x-a)$ for $1 \leqslant a \leqslant l$. By Lemma 2.3 (fact 4), we have a polynomial $h(x)$ (factor of $x^r - 1$) of degree $d = o_r(p)$ irreducible in $\mathbb{F}_p$. Note that

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

implies that

$$(x-a)^n \equiv (x^n - a)(\mod h(x), p)$$

So the identities on each binomial hold in the field $\mathbb{F}_p[x]/(h(x))$. The set of l binomials form a large cyclic group in this field. □

Lemma 3.4. *In the field $\mathbb{F}_p[x]/(h(x))$, the group generated by the l binomials: $(x-a)$, $1 \leqslant a \leqslant l$ i.e.,*

$$G = \{ \prod_{1 \leqslant a \leqslant l} (x-a)^{\alpha_\alpha} | \alpha_\alpha \geqslant 0,\ \forall\ 1 \leqslant a \leqslant l\}$$

is cyclic and of size $> (\frac{d}{l})^l$.

Proof. It is clear that G is a group and since it is a subgroup of the cyclic group $(\mathbb{F}_p[x]/(h(x)))^*$, it is also cyclic.

Now consider the set

$$S = \{ \prod_{1 \leqslant a \leqslant l} (x-a)^{\alpha_\alpha} | \sum_{1 \leqslant a \leqslant l} \alpha_\alpha \leqslant d-1, \alpha_\alpha \geqslant 0, \forall 1 \leqslant a \leqslant l\}.$$

The following argument shows that all the elements of S are distinct in $\mathbb{F}_p[x]/(h(x))$. The while loop ensures that once it halts the final r is such that $r > q > 4\sqrt{r}\log n > l$. Also step 4 of the checks gcd of r and n. If any of the a's are congruent modulo p, then $p < l < r$ and this implies that step 4 of the algorithm identifies n as composite. Therefore, none of the a's are congruent modulo p. So any two elements of S are

distinct modulo p. This implies that all elements of S are distinct in the field $\mathbb{F}_p[x]/(h(x))$ since the degree of any element of S is less than d, the degree of $h(x)$.

The cardinality of the set S is:

$$\left(\begin{array}{c} l+d-1 \\ l \end{array} \right) = \frac{(l+d-1)(l+d-2)\cdots(d)}{l!}$$

$$\left(\begin{array}{c} l+d-1 \\ l \end{array} \right) > \frac{d^l}{l^l}$$

$$\left(\begin{array}{c} l+d-1 \\ l \end{array} \right) > \left(\frac{d}{l}\right)^l$$

Since S is just a subset of G we get the result.

□

Since $d \geqslant 2l$, size of G is $> 2^l = n^{2\sqrt{r}}$. Let $g(x)$ be a generator of G. Clearly, order of $g(x)$ in $\mathbb{F}_p[x]/(h(x))$ is $> n^{2\sqrt{r}}$. We now define a set related to $g(x)$ which will play an important role in the remaining arguments Let

$$I_{g(x)} = \{m | g(x)^m \equiv g(x^m) (\mod x^r - 1, p)\}.$$

Here is a nice property of $I_{g(x)}$:

Lemma 3.5. *The set $I_{g(x)}$ is closed under multiplication.*

Proof. Let $m_1, m_2 \in I_{g(x)}$. So,

$$g(x)^{m_1} \equiv g(x^{m_1}) (\mod x^r - 1, p),$$

and

$$g(x)^{m_2} \equiv g(x^{m_2}) (\mod x^r - 1, p),$$

Also we have by substituting x^{m_1} in place of x in the second congruence:

$$g(x^{m_1})^{m_2} \equiv g(x^{m_1 m_2}) (\mod x^{m_1 r} - 1, p),$$

$$g(x^{m_1})^{m_2} \equiv g(x^{m_1 m_2}) (\mod x^r - 1, p),$$

From these, we get

$$g(x)^{m_1 m_2} \equiv (g(x)^{m_1})^{m_2} (\mod x^r - 1, p),$$

$$g(x)^{m_1 m_2} \equiv g(x^{m_1})^{m_2} (\mod x^r - 1, p),$$

$$g(x)^{m_1 m_2} \equiv g(x^{m_1 m_2}) (\mod x^r - 1, p),$$

Hence $m_1 m_2 \in I_{g(x)}$. □

Now we prove a property of $I_{g(x)}$ that plays a crucial role in our proof.

Lemma 3.6. *Let the order of $g(x)$ in $\mathbb{F}_p[x]/(h(x))$ be o_g. Let m_1, $m_2 \in I_{g(x)}$. Then $m_1 \equiv m_2 \mod r$ implies that $m_1 \equiv m_2 \mod o_g$.*

Proof. Since $m_1 \equiv m_2 \mod r$, $m_2 = m_1 + kr$ for some $k \geqslant 0$. Since $m_2 \in I_{g(x)}$: (in what follows, the congruences are in the field $\mathbb{F}_p[x]/(h(x))$ unless indicated otherwise.)

$$g(x)^{m_2} \equiv g(x^{m_2}) (\mod x^r - 1, p),$$

$$g(x)^{m_2} \equiv g(x^{m_2})$$

$$g(x)^{m_1 + kr} \equiv g(x^{m_1 + kr})$$

$$g(x)^{m_1} g(x)^{kr} \equiv g(x^{m_1}) \text{[By Lemma 2.3, fact 3]}$$

$$g(x)^{m_1} g(x)^{kr} \equiv g(x)^{m_1}$$

Now $g(x) \neq 0$ implies $g(x)^{m_1} \neq 0$ and hence we can cancel $g(x)^{m_1}$ from both sides leaving us with

$$g(x)^{kr} \equiv 1.$$

Therefore,

$$kr \equiv 0 \mod o_g$$
$$m_2 \equiv m_1 \mod o_g.$$

□

The above property implies that there are "very few" ($\leqslant r$) numbers in $I_{g(x)}$ that are less than o_g. Now we are ready to prove the most important property of our algorithm.

Lemma 3.7. *If n is composite, the algorithm returns COMPOSITE.*

Proof. Suppose that the algorithm returns PRIME instead. Thus, the for loop ensures that for all $1 \leqslant a \leqslant 2\sqrt{r}\log n$,

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

Notice that $g(x)$ is just a product of powers of l binomials $(x-a), (1 \leqslant a \leqslant l)$ all of which satisfy the above equality. Therefore,

$$g(x)^n \equiv g(x^n)(\mod x^r - 1, p).$$

Therefore, $n \in I_{g(x)}$. Also, $p \in I_{g(x)}$ by Lemma 2.3, fact 2 and, trivially, $1 \in I_{g(x)}$. We will now show that the set $I_{g(x)}$ has "many" numbers less than 0_g contradicting Lemma 3.6.

Consider the set

$$E = \{n^i p^j | 0 \leqslant i, j \leqslant \sqrt{r}\}.$$

By Lemma 3.5, $E \in I_{g(x)}$. Since $|E| = (1+\sqrt{r})^2 > r$, there are two elements $n^{i_1}p^{j_1}$ and $n^{i_2}p^{j_2}$ in E with $i_1 \neq i_2$ or $j_1 \neq j_2$ such that $n^{i_1}p^{j_1} \equiv n^{i_2}p^{j_2} \mod r$ by pigeon-hole principle. But then by Lemma 3.6 we have $n^{i_1}p^{j_1} \equiv n^{i_2}p^{j_2} \mod o_g$. This implies

$$n^{i_1 - i_2} \equiv p^{j_2 - j_1} \mod o_g.$$

Since $o_g \geqslant n^{2\sqrt{r}}$ and $n^{|i_1-i_2|}, p^{|j_1-j_2|} < n^{\sqrt{r}}$, the above congruence turns into an inequality. Since p is prime, this equality implies $n = p^k$ for some $k \geqslant 1$. However, in step 1 of the algorithm, composite numbers of the form p^k for $k \geqslant 2$ are already detected. Therefore, $n = p$: a contradiction. □

This completes the proof of the theorem.

4. Time Complexity Analysis

It is straightforward to calculate the time complexity of this algorithm.

Theorem 4.1. *The asymptotic time complexity of the algorithm is $O((\log n)^{12})$.*

Proof. The first step of the algorithm takes asymptotic time: $O((\log n)^3)$. As noted during the analysis of the algorithm in the previous section, the while loop makes $O((\log n)^6)$ iterations.

Let us now measure the work done in one iteration of the while loop. The first step (gcd computation) takes $\text{poly}(\log\log r)$ time. The next two steps would take at most $\sqrt{r}\text{poly}(\log\log r)$ time in the brute-force implementation. The next three

steps would take at most poly$(\log\log r)$ steps. Therefore, the total asymptotic time taken by the while loop is $O((\log n)^6\sqrt{r}) = O((\log n)^9)$.

The for loop does modular computation over polynomials. If repeated-squaring and Fast-Fourier Multiplication is used then one iteration of this for loop take $O(\log n\, r \log n)$ steps. Therefore, the for loop takes asymptotic time $O(r^{\frac{3}{2}}(\log n)^3) = O((\log n)^{12})$.

□

In practice, however, our algorithm is likely to work much faster. The reason is that even though we only know that there are "many" primes r such that $P(r-1) > r^{\frac{2}{3}}$, a stronger property is believed to be true. In fact it is believed that for many primes $r, P(r-1) = \frac{r-1}{2}$. (Such primes are called Sophie Germain primes.)

Definition *If both r and $\frac{r-1}{2}$ are primes, then $\frac{r-1}{2}$ is a Sophie Germain prime and the r's are co-Sophie Germain primes.*

Conjecture *The number of co-Sophie Germain primes is asymptotic to $\frac{Dx}{(\log x)^2}$, where D is the twin prime constant (estimated by Wrench and others to be approximately 0.6601618...).*

If this conjecture is true, then the while loop exits with a "suitable" r of size $O((\log n)^2)$.

Lemma 4.2. *Assuming the conjecture, there exists "suitable" r in the range $64(\log n)^2$ to $c_2(\log n)^2$ for all $n > n_0$, where n_0 and c_2 are positive constants.*

Proof. First of all note that if r is prime and $q = \frac{r-1}{2}$ is a prime, then the only possible orders of n modulo r are $\{1, 2, q, 2q = r-1\}$. But the orderer of n modulo r can be 1 or 2 for at most $2\log n$ primes r. (This is because (n^2-1) can have at most $\log(n^2-1)$ prime factors.) Let us leave aside these prime factors of (n^2-1) and consider the other co-Sophie Germain primes r for which the order of n modulo r is at least $\frac{r-1}{2}$. We would now like that

$$\frac{r-1}{2} \geqslant 4\sqrt{r}\log n$$
$$\sqrt{r} \geqslant 8\log n$$
$$r \geqslant 64(\log n)^2$$

Hence we consider the range $64(\log n)^2$ to $c_2(\log n)^2$ and we show that choosing c_2 large enough, we find at least one desired r in this range. By the conjecture 5, there are $\frac{Dc_2(\log n)^2}{(\log(c_2(\log n)^2))^2}$ co-Sophie Germain primes less than $c_2(\log n)^2$. Out of these, at most $\frac{D64(\log n)^2}{(\log(64(\log n)^2))^2}$ are less than $64(\log n)^2$ (again by the conjecture). From the remaining ones, there at most $2\log n$ primes for which order of n modulo r is 1 or 2. Thus we will choose c_2 such that

$$\frac{Dc_2(\log n)^2}{(\log(c_2(\log n)^2))^2} > \frac{D64(\log n)^2}{(\log(64(\log n)^2))^2} + 2\log n$$
$$\text{or, } \frac{c_2(\log n)^2}{(\log\log n)^2} > \frac{100(\log n)^2}{(\log\log n)^2}$$
$$\text{or, } c_2 > 100 \text{ [for large enough n]}.$$

□

This immediately leads us to a heuristic time complexity of $O(r^{\frac{1}{2}}(\log n)^2)$, for the while loop, $+ O(r^{\frac{3}{2}}(\log n)^3)$, for the for loop, $= O((logn)^6)$ for our algorithm.

5. Implementation

When implementing the algorithm it was not as easy as we expected. First we attempted to implement the algorithm in Java. However, Java is very inefficient at doing mathematical computation. The main problem we encountered is that Java does not have symbolic mathematical operations built into it. Instead of writing our own symbolic engine, a lengthy project in and of itself, or implementing someone else's symbolic engine for Java, we decided it would be best to switch to a more computationally efficient language with a symbolic engine built in. After exploring the options it was decided that we would use Matlab to implement the algorithm and Maple to do the symbolic computation found in the for loop. The maple command used to calculate the identity in maple was given by G. T. Gilbert in [5]. This led to an implementation which was far faster than anything Java could have produced.

Detailed below are amortized times to determine the primality of different prime numbers using the Matlab implementation. These programs were run on a laptop with a Pentium 4 2.0 GHz processor and 512 MB of ram. It was also run in Windows, which is known to be noticeably slower in Matlab than Linux distributions. However, to keep in mind that Matlab varies by approximately one tenth a second for each time the same number is computed.

Prime Number	**Time (seconds)**
2	≈ 0.01
3	≈ 0.08
13	≈ 0.11
61	≈ 0.19
97	≈ 0.29
199	≈ 0.46
293	≈ 0.61
397	≈ 0.81
499	≈ 0.99
587	≈ 1.13
599	≈ 1.15
691	≈ 1.27
797	≈ 1.34
887	≈ 1.45
911	≈ 1.46
997	≈ 1.59
1487	≈ 3.08
1499	≈ 3.13
1999	≈ 11.97
2999	≈ 36.40
3989	≈ 55.20
4999	≈ 64.50

It is of interest to note that 587, 887, and 1487 are co-Sophie Germain primes. This suggests that these numbers should be faster to calculate than the ones around them which are not co-Sophie Germain primes. This should actually speed up the amount of time it takes to find a suitable r, however, for numbers this small the r value is actually the same for two close primes such as 587 and 599. We found it also of interest that the numbers 797, 887, and 911 all share the same r value and

have very similar run times. Further, the for loop takes the most time to calculate and this is the major limiting factor in this implementation. We programmed it so that Matlab would display the "suitable" r, this was interesting since for even a number as large as 4999, an r was found nearly instantly, but the time it took to compute the identity in the for loop took over a minute. It is also very note worthy to mention that a sieve method running in Matlab only took ≈ 0.01 to determine that 4999 was prime. This algorithms implementation of maple starts to "blow up" after the primes near 1000. The time starts to become exponentially more and it seems that $Primes \notin P$.

Implementing the algorithm in Matlab still left many things to be desired. First, this implementation is not in polynomial time. It becomes impractical to determine that prime numbers over 1000 are prime. Maple symbolic calculations are not as fast as would be desired. However, we believe this leaves room for future work which we spend the rest of the paper detailing.

6. Future Work

We would like to examine the conjectures that the three authors felt could be proven. For example, in the algorithm [1], the "for loop needs to run for $1 \leqslant a \leqslant 2\sqrt{r}\log n$." The three authors believe the upper limits of a can be improved and this would lead to greater efficiency [1]. We will also investigate the conjecture proposed that "if r does not divide n and if $(x-1)^n \equiv (x^n-1)(\mod x^r-1, n)$, then either n is prime or $n^2 \equiv 1(\mod r)^2$" [1]. If this is true, the speed of the algorithm could be greatly increased. We would like to study these so that we might improve the time complexity.

We would like to implement the algorithm with PARI and GAP and work on calculating the equality $((x-a)^n \not\equiv (x^n-a)(\mod x^r-1, n))$ and see if we can improve computational speed through methods such as repeated squaring and using Fast-Fourier Multiplication. Using these methods produce a program that would run substantially faster than the Matlab implementation. We will then make minor tweaks to the algorithm to speed up the run time. This should lead to an overall much faster program.

More importantly, we will explore improving the time complexity by examining other mathematicians' work on the subject. We would like to examine papers such as Daniel Bernstein's, in which he claims to speed up the time complexity from $O((\log n)^{12})$ [3]. We hope these efforts will lead to improving the time complexity. For all changes that we will be able to make to the algorithm, we will attempt to prove why they did work or prove why they didn't.

References

[1] Agrawal, Kayal, Saxena. *Primes is in P*. Preprint 2002.
[2] Apostol T. M. *Introduction to Analytical Number Theory*. Springer-Verlag, 1997.
[3] Bernstein, D. *Proving Primality After Agrawal-Kayal-Saxena*. Preprint 2003.
[4] Fourvy E. *Theorem de Brun-Titchmarsh; application au theoreme de Fermat*. Invent. Math., 79:383-407, 1985.
[5] Gilbert, G. T. *The Polynomial Time Algorithm for Testing Primality*. Talk 2003.
[6] Koblitz, N. *A Course in Number Theory and Cryptography*. Springer: New York, 1994.