Primality Testing and Sub-Exponential Factorization

David Emerson
Advisor: Howard Straubing
Boston College Computer Science Senior Thesis

May, 2009

Abstract

This paper discusses the problems of primality testing and large number factorization. The first section is dedicated to a discussion of primality testing algorithms and their importance in real world applications. Over the course of the discussion the structure of the primality algorithms are developed rigorously and demonstrated with examples. This section culminates in the presentation and proof of the modern deterministic polynomial-time Agrawal-Kayal-Saxena algorithm for deciding whether a given n is prime. The second section is dedicated to the process of factorization of large composite numbers. While primality and factorization are mathematically tied in principle they are very different computationally. This fact is explored and current high powered factorization methods and the mathematical structures on which they are built are examined.

1 Introduction

Factorization and primality testing are important concepts in mathematics. From a purely academic motivation it is an intriguing question to ask how we are to determine whether a number is prime or not. The next logical question to ask is, if the number is composite, can we calculate its factors. The two questions are invariably related. If we can factor a number into its pieces then it is obviously not prime, if we can't then we know that it is prime. The definition of primality is very much derived from factorability.

As we progress through the known and developed primality tests and factorization algorithms it will begin to become clear that while primality and factorization are intertwined they occupy two very different levels of computational difficulty. The study and development of tests for primality has been focused on fast and deterministic algorithms. However, even early on in the development of factorization algorithms the idea of very fast and deterministic algorithms was relegated to the background as it became more and more accepted that these algorithms may not exist to solve factorization. A problem is in **P** or is said to have polynomial time complexity if it can reach a decision with a O bounding measured in polynomials of n and $\ln n$. For a long time the decision problem of whether a number was prime or not was thought to be in **P** but it was not until 2002 with the release of the AKS algorithm and its proof that this fact was completely demonstrated. Polynomial time algorithms are much more efficient than the more difficult problems of which **P** is a subset known as **NP** problems. As for the problem of factorization it is believed that no general polynomial timed algorithm exists to fully factor a given composite number. This is the reason that, as we develop the more sophisticated factorization algorithms we slip away from fully deterministic algorithms to more efficient probabilistic ones. The computational complexity of these two problems is drastically different. Primality is a relatively efficient problem when compared to that of factorization. We will see how different these two processes are as we move all in our development of both branches of algorithms.

While primality and factorization are studied in many branches of mathematics much of the recent study has been spurred by advances in cryptography and number theory. It is this application that has motivated the recent interest in efficient primality and factorization algorithms . Primality and factorization theory have had their greatest impact on modern encryption methods and the public key systems. In early encryption coding, the encoding and decoding were done by one key. This key was the encoder that allowed the sender and receiver to hide their messages when communicating. But by using one key at least two people held the code at one time. Even

worse, the unsolved problem of how to securely transmit the key to a remote party exposed the system to vulnerabilities. If the key was intercepted then the messages sent thereafter would no longer be secure. There was no way to protect the key transmission because in order to encrypt anything both parties needed to confer first. The modern public key system is responsible for solving this problem and maintaining the security of messages passed between two or more persons. The basic principles of the public key system, also known as asymmetric encryption, were conceived independently in 1976 by Diffie and Hellman of Stanford University and by Merkle at the University of California. See [Diffie and Hellman 1976].

In the new crypto-system, the encryption and decryption keys are unique and therefore do not need to be traded before messages can be sent. The method is called a public key system because while it has two unique encryption and decryption keys one of them is made public. Which key is made public depends on the type of message passing that one would like to do. The first application of the system is a way to have people *send* you encoded messages. If this is the goal then you publish the encryption key for people to encode their messages with and you keep the decryption key yourself. In this way you will be able to receive encoded messages that no one else can read but you. The second application of the key system is to turn around the first concept to obtain a "signature". That is, if we publish the decryption key but keep the encryption key secret we will be able to send messages stamped with our, hopefully, unique encryption signature. As an example, say that you are trying to send a message to a bank. You know who you would like to send the message to so you publish the decryption key and give it to them. Thereafter when the bank receives messages from someone they believe is you it will attempt to decrypt the message with the key that you gave them. If the message makes sense then it must have been encrypted with your unique key and therefore the bank can "safely" assume that you created the order. While the second method allows messages you send to be read by anyone, it allows you to have sole control over what content is put out under your name. In this way you obtain a unique signature with which to sign your messages.

When it was first conceived, it was believed that the public key crypto-system would completely outdate the old symmetric key systems. However, in today's encryption most algorithms still use the symmetric key system for their encryption. The public key crypto-system is too slow for most exchanges. What the public key crypto system is most often responsible for is the transmission of the symmetric key to the remote party. This key system solved the largest problem for the symmetric system, transmission of the shared key.

The scheme that is most widely recognized is by Rivest, Shamir, and Adleman. It was developed at MIT in 1977 and is referred to as the RSA algorithm, see [Rivest Shamir and Adleman 1978]. It is widely used as the basis for modern encryption. The scheme is based on a theorem of Euler that will be proven later in this paper.

The algorithm works as follows: Let p and q be distinct large primes and $n = p \cdot q$. Assuming we have two integers, d for decryption and e for encryption such that

$$d \cdot e \equiv 1 \pmod{\phi(n)}. \tag{1}$$

The symbol $\phi(n)$ denotes the Euler function of n. That is, the number of integers less than n that are relatively prime to n. The integers n and e are published. The primes p, q and integer d are kept secret.

Let M be the message that we would like to encrypt and send. We need M to be a positive integer less than n and relatively prime to n. If we keep M less than the p or q used to create n then we will be guaranteed that $(M, n) = 1$. However, the sender will not have access to p or q in practice for reasons that will become clear soon. So it is enough to have $M < n$ because the probability that p or q divides M, if it is less than n, is extremely small. If we encode our message in an alpha-numeric scheme that is recognizable to the party that we are sending the message to, then our M can be encoded and sent.

The sender encrypts the message by computing, E, the encrypted message

$$E = M^e \bmod n.$$

The message is decoded simply by applying d

$$E^d \bmod n.$$

The following theorem by Euler, that will be used to demonstrate that $E^d \bmod n$ is equal to M, is proven later. If n and b are positive and relatively prime this implies that

$$b^{\phi(n)} \equiv 1 \pmod{n}. \tag{2}$$

By the above relation (2) and equation (1)

$$\begin{aligned} E^d \equiv (M^e)^d \equiv M^{e \cdot d} &\equiv M^{(k \cdot \phi(n))+1} \pmod{n} \\ &\equiv (M^{(k \cdot \phi(n))}) \cdot M^1 \pmod{n} \\ &\equiv 1 \cdot M \pmod{n} \\ &\equiv M \pmod{n}. \end{aligned}$$

We know that M and $E^d \bmod n$ are both strictly less than n and so they must be equal.

To choose our e and d all we need to know is $\phi(n)$. If we choose e relatively prime to $\phi(n)$ then we are guaranteed a d that satisfies equation (1) by Euclid's algorithm. If we know the factorization of n, that is $n = p \cdot q$ where p, q are distinct, then it is a property of the Euler function that

$$\phi(n) = (p-1)(q-1).$$

There is no easier way of computing $\phi(n)$ without the factorization of n. Therefore, the way to find d is to know, or find, the factorization of n. After creating our encryption keys we merely throw out the p, q, and $\phi(n)$ used in their creation.

It is easy to see how important primality and factorization are to the creation and security of this algorithm. In order to create our n we need to be able to generate and test very large prime numbers efficiently and effectively. In this paper we examine the number theory and implementation behind some of the modern approaches to primality testing. The feasibility of the RSA system rests on the fact that it is possible to efficiently generate large primes. Its security is based on the conjecture that there is no efficient algorithm for factoring a large integer into its prime decomposition. As we stated above, we can compute $\phi(n)$ if we can calculate the factorization of n. Thereby, we can compute d. Because we display n publicly with e, if n could be factored quickly then our decryption key would not be secure for very long. It should be noted that this is why p and q must be large primes, among other things, since taking out small factors is much easier than large ones. In the second section of this paper modern factorization implementations and their mathematical foundations are explored.

2 Primality

Recognizing when a large number is prime or composite is a fundamental task in number theory and sits at the heart of many mathematical systems like cryptography. During the encryption process it is important to be able to generate large prime numbers to create "hard" to factor, large composite numbers. In factorization, essentially the opposite process, primality testing is fundamental to modern techniques like the Multiple Polynomial Quadratic Sieve. These algorithms require the ability to generate long lists of large prime numbers in a relatively short time.

For small numbers, it is generally easy to intuit an answer to the primality question by merely looking at the number. We all do the same thing. If the

number is even we know that it is not prime. If it is odd we consider the number and ask ourselves, does 3 go into that number? What about 5? This process is essentially trial division. We go through a small amount of prime numbers in our head and ask ourselves it the number in question is divisible by them. It is easy to see the problem with this sort of method if we would like to prove primality for any n let alone the large n needed to create our encryption keys. Trial division *is* a deterministic algorithm for primality testing. Unfortunately, it has an extremely inefficient runtime. If we want to prove that n is prime we must trial divide n by every prime less than $\sqrt{n}$. If none of them divides n then $n$is prime. As n gets larger so does the number of primes that we will need to test. Indeed, The Prime Number Theorem states that

$$\lim_{x\to\infty} \frac{\pi(x)}{x/\ln(x)} = 1 \ .$$

Therefore $\pi(x) \sim \frac{x}{\ln(x)}$. So, the number of primes we must test grows significantly as n does. If we were to try to factor a composite n of 100 digits using trial division we would have to perform approximately $\sqrt{10^{100}}/\ln\sqrt{10^{100}} \approx 8.68588964 \times 10^{47}$ trial divisions to fully determine whether n were prime or composite. This leads us to consider the properties of prime numbers in order to come up with a test that will prove primality more efficiently than just through trial division.

We begin our discussion of primality testing with an observation first made by Fermat while studying Mersenne primes.

Theorem 1. *If p is an odd prime then $2^{p-1} \equiv 1 \pmod{p}$.*

We will not give the proof of this immediately, but instead will prove a more general result below. This theorem illuminates a particular property of prime numbers. Consider using this theorem as a primality test. The algorithm could be defined as follows.

Algorithm 1. *Given an n*

1. *Calculate $k = 2^{n-1} \bmod n$*
2. *if(k is equal to 1)*
   return *prime*
   else **return** *composite*

This algorithm will run quickly even for very large n since we can perform repeated squaring and reduce mod n after each squaring. Theorem 1 must hold if n is prime and therefore this algorithm will give no false negatives. On the other hand, it is not sufficient to prove that n is a prime. Numbers

such as $341 = 11 \cdot 31$ satisfy this condition and are clearly composite. So this algorithm can err by classifying an integer as prime when it is actually composite.

A more general case of the above theorem holds. Indeed, we consider the theorem due to Euler that we proposed earlier

Theorem 2 (Euler)**.** *Let n and $b \in \mathbb{Z}$ and $(n, b) = 1$ then $b^{\phi(n)} \equiv 1 \pmod{n}$.*

Proof. Let $t = \phi(n)$ and let $a_1, a_2, \ldots, a_t$ be those integers which are less than n and $(a_i, n) = 1$. Let $r_1, r_2, \ldots, r_t$ be the residues of those $a_i \cdot b$. That is

$$b \cdot a_i \equiv r_i \pmod{n}.$$

It should be noted that each r is pairwise distinct. Indeed, if they were not distinct mod n we would have

$$b \cdot a_i \equiv b \cdot a_j \pmod{n}.$$

But since $(n, b) = 1$ we can divide by b

$$a_i \equiv a_j \pmod{n}.$$

If $a_i, a_j < n$

$$a_i = a_j \text{ but } a_i \text{ and } a_j,$$

but a_i and a_j are distinct so we derive a contradiction. Further, each r_i is relatively prime to n. To see this consider $(n, r_i) \neq 1$. That is, consider $(n, r_i) = m$ for some m $\in \mathbb{Z}$

$$b \cdot a_i - r_i = nk \text{ implies } b \cdot a_i = nk - r_i \text{ for some k} \in \mathbb{Z}.$$

But $(n, r_i) = m$ and so $m \mid nk - r_i$ meaning that $m \mid b \cdot a_i$. This would imply that a_i and n share a common divisor (*we know that $(n, b) = 1$ by our theorem assumptions*) which is a contradiction. So each r_i is coprime to n. Therefore $r_1, r_2, \ldots, r_t$ is a set of size $\phi(n)$ distinct integers coprime to n. That is, the set of r_i's is the same as our set of a_i's.

Hence,

$$a_1 \cdot a_2 \cdot a_3 \cdot \ldots \cdot a_t = r_1 \cdot r_2 \cdot r_3 \cdot \ldots \cdot r_t.$$

Therefore,

$$\begin{aligned} r_1 \cdot r_2 \cdot \ldots \cdot r_t &\equiv b \cdot a_1 \cdot b \cdot a_1 \cdot \ldots \cdot b \cdot a_t \pmod{n} \\ &\equiv b^t (a_1 \cdot a_2 \cdot a_3 \cdot \ldots \cdot a_t) \pmod{n} \\ &\equiv b^t (r_1 \cdot r_2 \cdot r_3 \cdot \ldots \cdot r_t) \pmod{n}. \end{aligned}$$

Divide both sides by our sequences of r_i's and we have

$$\begin{aligned} b^t &\equiv 1 \pmod{n} \\ b^{\phi(n)} &\equiv 1 \pmod{n}. \end{aligned}$$

□

Theorem 1 is simply the special case where $b = 2$.

The result of Euler's more general conditions from Theorem 2 allow us more flexibility within the base of the congruence. We can now restate our algorithm with a slightly more general calculation step.

Algorithm 2 (The Pseudoprime Test)**.** *Given n and a base b*

1. Calculate $k = b^{n-1} \bmod n$
2. if(k is equal to 1)
return *prime*
else **return** *composite*

Composite integers that pass this test for a given base b are called *Pseudoprimes*. An integer like 341 is called a pseudoprime for base 2. While these integers are very bad for our test, it is encouraging that these kinds of composites are relativley rare. Below 1 million there are only 245 pseudoprimes as compared to 78498 primes.

Another good result from Euler's criterion is that it gives us a larger pool of bases to draw from and the congruence must hold for each base if n is prime. It is conceivable that changing the base will reveal more composite numbers because we are guaranteed that a prime will pass this test for all b such that $(b, p) = 1$. Pseudoprimes like 341 are revealed in this manner. $3^{341-1} \equiv 56 \pmod{341}$. So the question is how many bases b should we test before strongly believing that our number is prime. Examining this we run into a problem. There are numbers that pass the base test for all b where $(b, p) = 1$. A number with this property is called a *Carmichael* number. These numbers are very rare but pose a problem in that they could pass every test regardless of the base b and yet be composite.

In order to get around this problem the Strong Pseudoprime test was developed by Pomerance, Selfridge, and Wagstaff.

Consider an n that is a pseudoprime relative to a base b. This imples that $b^{n-1} \equiv 1 \pmod{n}$. So

$$n \mid b^{n-1} - 1.$$

we know that n is odd (otherwise it is not prime). So, $n = 2m + 1$ for some m, and thus $n \mid b^{2m} - 1$, but $b^{2m} - 1 = (b^m + 1)(b^m - 1)$. This implies

that n divides $(b^m+1)(b^m-1)$. If n is prime it must divide at least one of these factors. Further, n cannot divide both of them or it would divide their difference. So consider $(b^m+1)-(b^m-1)=2$. We know that $n>2$ then n cannot divide both. So, if n is prime then $n \mid b^m+1$ or $n \mid b^m-1$. That is, either

$$b^m \equiv -1 \pmod{n} \text{ or } b^m \equiv 1 \pmod{n}. \tag{3}$$

If n is composite there is a good chance that some of its factors divide b^m+1 and some b^m-1. If this is the case then the factors combined would pass the pseudoprime test and since $(b^m+1)(b^m-1)$ combines the factors, n would divide $b^{2m}-1$ but would fail to hold Equation (3) from above.

Ex 1. *Consider* $n=341$*, so* $m=170$

We know that 341 *passes the pseudoprime test. To use our new criterion we look at* 2^{170} *reduced* mod n*, but* $2^{170} \equiv 1 \pmod{n}$*. So* 341 *passes this test, but we can apply the same trick that we used above because* $2^{170}-1$ *can be factored further. We know that* $341 \mid 2^{170}-1$ *but*

$$2^{170}-1=(2^{85}-1)(2^{85}+1).$$

If 341 *were a prime it divides exactly one side of this equation for the same reasons as above. So either*

$$341 \mid 2^{85}-1 \Rightarrow 2^{85} \equiv 1 \pmod{341}$$
$$\textit{or}$$
$$341 \mid 2^{85}+1 \Rightarrow 2^{85} \equiv -1 \pmod{341}.$$

but $2^{85} \equiv 32 \pmod{341}$

We see from this example that we reduce even exponents to their furthest factorization and if n is prime then it will divide exactly one of said factors. Therefore, we write $n=2^a \cdot t+1$ (since n is an odd) where t is odd and $a>1$ since $n>2$. We can write

$$b^{n-1}-1=(b^{2^{a-1}t}-1)(b^{2^{a-1}t}+1).$$

If n has passed the pseudoprime test, then n must divide exactly one of these. Looking back at the example we see that if n passes the prime test for $b^{2^{a-1}t}-1$ and 2^{a-1} is even then we can factor further and n will divide exactly one of the factors if it is prime. So split it

$$b^{2^{a-1}t}-1=(b^{2^{a-2}t}-1)(b^{2^{a-2}t}+1).$$

Hence we consider the complete factorization of b^{n-1} where $n = 2^a \cdot t + 1$. Which is

$$b^{n-1} = (b^t - 1)(b^t + 1)(b^{2t} + 1)(b^{4t} + 1) \dots (b^{2^{a-1}t} + 1). \tag{4}$$

If n is really prime then it must divide exactly one of these factors. Meaning if n is prime one and only one of these congruences holds

$$b^t \equiv 1 \text{ or } -1 \pmod{n} \text{ or } b^{2t} \equiv -1 \pmod{n} \dots \textit{etc.}$$

So we can now fully define our Strong Pseudoprime Test.

Algorithm 3 (Strong Pseudo Prime Test)**.** *Given n and a base b*

1. *Write $n = 2^a \cdot t + 1$*
2. *Reduce b^{n-1} to its complete expansion as in Equation* (4)
3. *Test each factor's individual congruence to determine if n divides exactly one of these factors*
4. *If n does divide one of the factors*
   return *prime*
   else **return** *composite*

It is natural wonder if there are composites that will pass the Strong Pseudoprime test. Unfortunately these composites do exist. We say that an integer is a *Strong Pseudoprime* if it is composite and passes the Strong Pseudoprime Test for a base b. Though Strong Pseudoprimes exist they are even more rare than regular pseudoprimes. There are 21853 pseudoprimes and only 4842 strong pseudoprimes base 2 under 25×10^9. If we string together multiple bases and test an n against each of them we eliminate even more composite numbers that are strong pseudoprimes only some of the bases. There is only one composite number that is a strong pseudoprime for the bases $2, 3, 5$ and 7 under 25×10^9 . That strong pseudoprime being $3215031751 = 151 \cdot 751 \cdot 28351$.

The good news for the strong pseudoprime test is that there is no analogue to Carmichael numbers. That is, there are no composite numbers that will pass the strong pseudoprime test for every base. To prove this some theorems need to be established first. We must also define a *quadratic residue*. Given an integer n and a prime p with $(n, p) = 1$, if there exists an integer t such that $n \equiv t^2 \pmod{p}$ then we say that n is a *quadratic residue modulo p*

Theorem 3. *Let p be an odd prime such that $p = 2m + 1$ and let b be a positive integer not divisible by p so that $(p, b) = 1$, then $b^m \equiv 1 \pmod{p}$ if and only if there exists a $t \in \mathbb{Z}^+$ such that $b \equiv t^2 \pmod{p}$ (that is b is a quadratic residue mod p)*

Proof. ($\Leftarrow$)

If there exists a $t \in \mathbb{Z}^+$ such that $b \equiv t^2 \pmod{p}$ then $b^m \equiv t^{2m} \equiv t^{p-1} \pmod{p}$ since $2m = p-1$. We know that t cannot be divisible by p since if

$$\begin{aligned} p \mid t &\Rightarrow pk = t \\ &\Rightarrow (pk)^2 = t^2 \\ &\Rightarrow p^2k^2 = t^2 \\ &\Rightarrow p(pk^2) = t^2 \\ &\Rightarrow p \mid t^2 \\ &\Rightarrow p \mid b. \end{aligned}$$

but this is a contradiction to our assumption that $(p, b) = 1$. So t is relatively prime to p. By Euler's theorem and Fermat observation

$$b^m \equiv t^{p-1} \equiv 1 \pmod{p}.$$

□

The other direction of the theorem is informed by the following two ideas and so is deferred.

Lemma 1. *Given* $i^2 \equiv j^2 \pmod{p}$ *then* $i \equiv j \pmod{p}$ *or* $i \equiv -j \pmod{p}$

Proof. If $i^2 \equiv j^2 \pmod{p}$, we know that $i^2 - j^2 = (i+j)(i-j) = pk$ for some $k \in \mathbb{Z}$. Therefore $p \mid (i+j)(i-j)$ but since p is prime it divides one or the other. So $i \equiv j$ or $-j \pmod{p}$. □

Each quadratic residue comes up exactly twice since each quadratic residue can be formed by squaring the integers 1 up to $p-1$ and taking their residues. Hence the number of quadratic residues that are positive and less than p is $\frac{(p-1)}{2}$

Theorem 4 (Wilson's Theorem)**.** *The integer* n *divides* $(n-1)! + 1$ *if and only if* n *is prime*

Proof. ($\Rightarrow$)

Consider the contrapositive. That is, if n is not prime (composite) then it does not divide $(n-1)! + 1$. There are two cases:

If n is composite and $n = 4$, $n \nmid (n-1)! + 1$ since $3! + 1 = 7$ and $4 \nmid 7$.

Now consider the case when $n > 4$

$$(n-1)! = (n-1)(n-2)(n-3)\dots(2)(1). \tag{5}$$

If n is composite it can be uniquely decomposed into a factorization of prime powers. That is, $n = p_1^a \cdot p_2^b \cdot p_3^c \cdot \ldots$ for some primes p_i with associated $a, b, c, \ldots \in \mathbb{Z}^+$. If each factor is strictly less than n, that is, each $p_i^x < n$, then each of the factors will occur on the right hand side of Equation (5). In this way we see that $n \mid (n-1)!$. If, however, n is a prime power, for example cases like $n = 9$, it has a single factor whose exponent makes it equal to n. That is to say $n = p^a$. Consider first is $a = 2$ then p occurs on the right hand side of Equation (5) since it is less than n. Also, since n is not 4 then $p > 2$. So, $2 \cdot p$ also occurs on the right hand side of Equation (5) and so $n \mid (n-1)!$. Now for the more general case. We know that p^{a-1} exists on the right hand side of Equation (5) since $p^{a-1} < n$. Also p must exist in our expansion (5). So, $p^{a-1} \cdot p$ occurs in the expansion (5). Thus $n \mid (n-1)!$. But if $n \mid (n-1)!$ it cannot divide $(n-1)! + 1$.
($\Leftarrow$)

If n is prime, for each $i < n$ there exists a unique positive integer $j < n$ such that $i \cdot j \equiv 1 \pmod{n}$. This is by the uniqueness of inverses in $\mathbb{Z}_n$. Further, if $i \neq 1$ or $n-1$, then i and j are distinct. If they were not then

$$\begin{aligned} i = j &\Rightarrow i^2 \equiv 1 \pmod{n} \\ &\Rightarrow n \mid i^2 - 1 \\ &\Rightarrow n \mid (i-1) \text{ or } (i+1). \end{aligned}$$

but this is not possible since i by definition is strictly less than n.

Thus consider our expansion from equation (5). We pair up the integers 2 through $n-2$ with their inverses mod n. Reducing mod n we see

$$\begin{aligned} (n-1)! &\equiv 1 \cdot \underbrace{(\text{product of ones})}_{\text{reduction of inverse pairs mod} n} \cdot (n-1) \pmod{n} \\ &\equiv n-1 \pmod{n} \\ &\equiv -1 \pmod{n}. \end{aligned}$$

So $n \mid (n-1)! + 1$. □

It may be observed that this is, in essence, a primality test but given the need to calculate factorial expansions of order n and the division step it's time and space complexity are limiting at best.

Now we can finish the other direction of Theroem 3.

Proof. ($\Rightarrow$)
Beginning again with the contrapositive. We will suppose that b is <u>not</u> a quadratic residue mod p, and will prove that $b^m \equiv -1 \pmod{p}$ (recalling

that if p is a prime of the form $2m+1$ then either $p \mid b^m - 1$ or $p \mid b^m + 1$ but not both). So the statement $b^m \equiv -1 \pmod{p}$ is exactly the negation of $b^m \equiv 1 \pmod{p}$.

For all positive $i < p$ there exists a unique positive $j < p$ such that $i \cdot j \equiv b \pmod{p}$. This statement can be seen through manipulation of the uniqueness of inverses in $\mathbb{Z}_p$. Indeed, in order to find j we consider j' such that $i \cdot j' \equiv 1 \pmod{p}$ and multiply through by b so that $i \cdot bj' \equiv \pmod{p}$. This can be done because $(b, p) = 1$. Reduce $bj' \bmod p$ and call this j. Hence, $i \cdot j \equiv b \pmod{p}$. j is unique. If it were not we would have $i \cdot j \equiv i \cdot k \equiv b \pmod{p}$ for some k. Multiply by the inverse of $i \bmod p$ so that

$$(i^{-1})(i)(j) \equiv (i^{-1})(i)(k) \pmod{p} \Rightarrow j \equiv k \pmod{p},$$

but both j and k are less than p. So j=k. Thus j is unique mod p. Also, since b is not a quadratic residue mod p, $i \neq j$

Now, pair those positive integers whose product is $b \bmod p$ and multiply them together. We obtain

$$(p-1)! \equiv b \cdot b \cdot \ldots \cdot b \pmod{p}.$$

There are m of these b's since our $p = 2m + 1$ Yielding

$$(p-1)! \equiv b^m \pmod{p}. \tag{6}$$

By Theorem 4 we know that $n \mid (n-1)! + 1$ if and only if n is prime. So $p \mid (p-1)! + 1$. Said differently, $(p-1)! \equiv -1 \pmod{p}$. Because of this and congruence relation (6) we have shown that that $b^m \equiv -1 \pmod{p}$. □

Our Strong Pseudoprime test constitutes a much stronger criterion for primality as it eliminates many of the normal pseudoprimes that we dealt with in the more naive test. However, the question of whether we will be plagued by composite numbers that are strong pseudoprimes for all b where $(b, n) = 1$ still remains. The most significant improvement for our strong pseudoprime test is that it does not have any numbers analogue to Carmichael numbers. The proof of this fact if n has atleast 2 distinct prime factors is considered below. The case where n is a prime power is excluded because methods like Newton iterations can be used to efficiently determine whether a given n is a prime power. For the full proof see [Bressoud 1989].

To prove this theorem we will first need another small lemma.

Lemma 2. *Let $r, s \in \mathbb{Z}^+$ with $g = gcd(r, s)$. Also let p be a prime and b be an integer not divisible by p.*

$$\textit{If } b^r \equiv 1 \pmod{p} \textit{ and } b^s \equiv 1 \pmod{p} \textit{ then } b^g \equiv 1 \pmod{p}$$

Proof. $g = a \cdot r + c \cdot s$ and so $b^g = b^{a \cdot r + c \cdot s}$. We see then that

$$(b^r)^a \cdot (b^r)^c \equiv 1 \pmod{p}.$$

□

Theorem 5. *Let n be an odd composite number with at least two distinct prime factors say p and q. Let m be any integer relatively prime to n such that m is a quadratic residue* mod$p$ *and is not a quadratic residue* mod$q$*. n will fail the Strong pseudoprime test for base m.*

Proof. We know that m exists by the Chinese Remainder Theorem that will be proven later. p and q can be written $p = 2^a \cdot s + 1$ and $q = 2^b \cdot t + 1$ where $a, b \geq 1$ and s, t are odd. Without loss of generality we can assume $a \leq b$. Write n as $2^c \cdot u + 1$ where $n \geq 3$ and u is odd, hence $c \geq 1$. We know that if n is a strong pseudoprime base m then each prime factor of n divides exactly one of

$$m^u - 1, m^u + 1, m^{2u} + 1, m^{4u} + 1, \ldots, m^{2^{c-1} \cdot u}. \tag{7}$$

Further n passes this test if and only if each prime factor divides the same one (Since n must divide it as well). Now, if an odd prime divides $m^{2^{j-1} \cdot u} + 1$ then it divides

$$m^{2^j \cdot u} - 1 = (m^{2^{j-1} \cdot u} + 1)(m^{2^{j-1} \cdot u} - 1).$$

But does not divide

$$m^{2^{j-1} \cdot u} - 1 = (m^{2^{j-1} \cdot u} + 1) - 2. \quad \text{Since, of course, } n \nmid 2.$$

Let j be the smallest integer such that $p \mid m^{2^j \cdot u} - 1$.
Let k be the smallest integer such that $q \mid m^{2^k \cdot u} - 1$.

We know that $0 \leq j, k \leq c$. We have that m is a quadratic residue mod$p$ so we know by Theorem 3, Euler's Criterion, that

$$\begin{aligned} m^{2^{a-1} \cdot s} &\equiv 1 \pmod{p} \\ &\Rightarrow p \mid m^{2^{a-1} \cdot s} - 1. \end{aligned}$$

Now consider, $2^j \cdot u$ and $2^{a-1} \cdot s$ with Lemma 2. m is relatively prime to p and we have

$$m^{2^j \cdot u} \equiv 1 \pmod{p},$$

and

$$m^{2^{a-1} \cdot s} \equiv 1 \pmod{p}.$$

So,

$$m^{gcd(2^j \cdot u, 2^{a-1} \cdot s)} \equiv 1 \pmod{p}.$$

If a were less than j then

$$gcd(2^j \cdot u, 2^{a-1} \cdot s) = 2^{a-1} \cdot gcd(u, s),$$

since the factors of 2 that divide both will be taken out by 2^{a-1} and all that will be left is an odd s in $(2^{a-1} \cdot s)$. So we are left with finding the factors that divide both u and s but that is precisely $gcd(u, s)$. Hence, we know that $2^{a-1} \cdot u$ is a multiple of $gcd(2^j \cdot u, 2^{a-1} \cdot s)$.

Thus, $m^{2^{a-1} \cdot u} \equiv 1 \pmod{p}$. To see this consider

$$\begin{aligned}(2^{a-1} \cdot u) = gcd(2^j \cdot u, 2^{a-1} \cdot s) \cdot k \Rightarrow (m^{gcd(2^j \cdot u, 2^{a-1} \cdot s)})^k &\equiv m^{2^{a-1} \cdot u} \\ &\equiv (1)^k \pmod{p} \\ &\equiv 1 \pmod{p}.\end{aligned}$$

But this congruence contradicts the minimality of j. Therefore $a > j$.

Now consider q. m is not a quadratic residue $\bmod p$ implies $q \nmid m^{2^{b-1} \cdot t} - 1$. So q must divide $m^{2^{b-1} \cdot t} + 1$ and therefore it must also divide $m^{2^b \cdot t} - 1$. Again applying Lemma 2 we know that

$$\begin{gathered}q \mid m^{2^k \cdot u} - 1 \text{ and } q \mid m^{2^b \cdot t} \text{ so,} \\ q \mid m^{gcd(2^k \cdot u, 2^b \cdot t)} - 1.\end{gathered}$$

If b is larger than k

$$gcd(2^b \cdot t, 2^k \cdot u) = 2^k gcd(t, u).$$

This implies that $m^{2^{b-1}} \equiv 1 \pmod{q}$. This is because $2^k \cdot t$ is a multiple of the $gcd(2^k \cdot u, 2^b \cdot t)$ making $2^{b-1} \cdot t$ a multiple since it can be written, $(2^{b-1} \cdot t) = (2^k \cdot t)2^z$ where $z = (b-1) - k$. We recognize that z is positive since $b > z$. If $2^{b-1} \cdot t$ is a multiple of $gcd(2^k \cdot u, 2^b \cdot t) this implies m^{2^{b-1} \cdot t} \equiv 1 \pmod{q}$. This contradicts our assumption that m is not a quadratic residue mod$q$. So $b \leq k$ and we get $j < a \leq b \leq k$. This implies that $j \neq k$. Therefore p and q divide different factors in our list in Eq.(7). So n fails the Strong Pseudoprime test for base m □

The fact that there is no analogue to Carmichael numbers for the Strong Pseudoprime test is significant. We have seen that the conditions of our test for primality can be used to rule out many composite numbers given a base b. Further, if we vary this base b we find that Strong Pseudoprimes become even more rare. This theorem guarantees that we do not have the issue of Carmichael numbers. That is, there will be a base b relatively prime to a composite n for which n will fail the Strong Pseudoprime test. Therefore, we

can prove the compositeness of an n if we can find this b. In general, to prove the primality of an n using only this fact we must go through every base b that is relatively prime to n and test n against each b. This naive algorithm takes $\phi(n)$ iterations to pass through each base if n is prime ($\phi(p) = p - 1$ if p is prime). So we still have a very slow way of deterministically testing whether a given n is actually prime. We have a lot of confidence that if a composite n passes the Strong Pseudoprime test for a large number of bases that it is probably prime because of how rare they are. So it is possible to only test a fraction of the bases in the set $\phi(n)$ and claim with high probability that n is prime. To test this idea we can examine real data produced by implementing the Strong Pseudoprime test.

In actual practice an implementation of the pseudoprime test can produce probable primes extremely quickly without switching bases very often. Indeed during experimentation a random number generator was used to create pairs of large numbers that were multiplied together to produce a list of composite numbers. The Strong Pseudoprime test was run with a random prime base that was pulled from a large list of primes. The test was run for each composite number in our generated list coprime to that prime. If the prime was not a witness to the compositeness of n, a first order change of base was recorded. Another prime p was then picked from our list and those n which were coprime to p and had yet to be proven composite were tested again. If that number had already been tested and still passed the test for the current base a second order change of base was recorded. Due to the already rare nature of Strong Pseudoprimes the necessity of changing bases to prove compositeness was already very small. Sample sizes of 1000 or more were often needed to engender even one base switch. As the order of the base changes increased the tally of numbers needing that many base switches decreased significantly. It was rare to get even one number that needed 3 base changes. This phenomenon clearly follows the reduction estimates of base switches over the primes $2, 3, 5$, and 7 in Bressoud's "Factorization and Primality Testing".

This suggests that our $\phi(n)$ estimate for the number of base changes needed to deterministically classify a composite number is pessimistic. Our experiments and intuition indicate that we should not have to go through $\phi(n)$ test steps to find a witness to n's compositeness.

Definition 1. $S(n) = \{a \bmod n \mid n \textit{ is a strong pseudoprime base } a\}$

Abusing notation we will let $S(n)$ also denote $\mid S(n) \mid$.

Theorem 6. *For all odd composite integers $n > 9$, $S(n) \leq \frac{1}{4}\phi(n)$*

This theorem suggests that at least $\frac{3}{4}$ of all integers $\in [1, n-1]$ coprime to n are witnesses for n. That is, n would fail the strong pseudoprime test for around $\frac{3}{4}$ of all integers relatively prime to $n \in [1, n-1]$. To see the proof of this theorem see [Crandall and Pomernace 2005].

This bound on $S(n)$ is extremely powerful. It implies that in order to produce a witness to the compositeness of n we need only, at most, test $\frac{1}{4}\phi(n) + 1$ bases (those bases being unique and elements of $\mathbb{Z}_n^*$) and we will either produce a witness to n's compositeness or n is prime. This gives us a deterministic primality test. However, it has a non-polynomial time complexity. The profitable use for this algorithm comes from its application as a probabilistic primality test. Indeed, versions of this and other algorithms it are still used in practice for real time cryptographic key production. While it is not a guarantee, the lower bound on the cardinality of $S(n)$ means that we can produce witnesses to a composite number rather quickly since the individual computation of each test is quite fast. The resulting algorithm is know as the Miller-Rabin Test.

Algorithm 4 (Miller-Rabin Test)**.**
Given $n > 3$ and a number of iterations k

1. for($i = 0; i < k;$ i++)
Choose a random integer a in the set of integers relatively prime to n and less than n
Using Algorithm 3 test is n passes the Strong Pseudoprime test for base a
2. if(n passed Algorithm 3 for all of the bases)
return *prime*
else **return** *composite*

If the Miller-Rabin test is done with a random base and conducted over 25 iterations, by Theorem 6 we have less than a $\frac{1}{4^{25}}$ chance that n is a strong pseudoprime for each base. This is less than a one in 8.88×10^{16} chance that n is actually a composite number after passing all of the tests. This chance is so small that the risk of a false prime is generally outweighed by the time a deterministic algorithm, such as the Agrawal-Kayal-Saxena (AKS) Algorithm, though polynomial, will take to prove primality for the very large n needed for RSA cryptography.

The AKS algorithm was first published in 2002. Its announcement created a significant stir in mathematics and computer science. The AKS primality test was the first fully proven deterministic polynomial time primality test. Earlier deterministic algorithms were released before the AKS algorithm but their proofs and time complexity relied on the Extended Riemann

Hypothesis. The generalized Riemann Hypothesis is one of the million-dollar millennium problems offered by the Clay Mathematics Institute and remains, as of yet, unproven. The Miller-Rabin test expects to prove compositeness in polynomial time but this time estimate still remains probabilistic since it is conceivable, if we are unlucky, that we will have to go through the full $\frac{1}{4}\phi(n)+1$ bases to prove compositeness. The Gauss sums primality test is a fully proved deterministic primality test that runs very close to polynomial time, see [Crandall and Pomernace 2005]. Its time bound estimate is $(\ln n)^{c\cdot \ln\ln\ln n}$ where $\ln\ln\ln n$ is non-polynomial but grows extremely slowly relative to n. The AKS algorithm surpassed all of its predecessors. To add to its accomplishment its algorithm and principles are relatively simple when considered against its achievement. As a quick discussion of notation $o(n)$ with n being an element of a group G means the order of the element n in G. Also, $f(x) \equiv g(x) \pmod{x^r-1, n}$ means that the remainder of $f(x)$ synthetically divided by x^r-1 with coefficients reduced mod$n$ is congruent to $g(x)$. The notation $\lg n$ denotes the $\log_2$ of n. Finally, the notations $\mathbb{Z}[x]$ denotes the integer polynomial ring and $\mathbb{F}[x]$ denotes the field of polynomials. The AKS algorithm,

Algorithm 5. *(AKS)*
Given $n \geq 2$

1. *if* n *is a square or higher power*

   return *composite*

2. *Calculate the least* $r \in \mathbb{Z}$ *such that* $o(n) \geq \lg^2 n$ *in* $\mathbb{Z}_r^*$
   If n *has a proper factor* $\in [2, \sqrt{\phi(r)}\lg n]$

   return *composite*

3. $For(i \leq a \leq \sqrt{\phi(r)}\lg n)$
   $If((x+a)^n \not\equiv x^n + a \pmod{x^r-1, n})$

   return *composite*

4. **return** *prime*

The theorems and proofs that follow demonstrate the correctness of this algorithm.

Theorem 7. *If* n *is prime then* $g(x)^n \equiv g(x^n) \pmod{n}$

For our algorithm it will only be necessary to consider $g(x) \in \mathbb{Z}[x]$ such that

$$(x+a)^n \equiv x^n + a \pmod{n}. \tag{8}$$

This holds for all $a \in \mathbb{Z}$ if n is prime.

Proof. Consider the expansion of $(x+a)^n$

$$\begin{aligned}(x+a)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k}a^k &= \binom{n}{0}x^n + \binom{n}{1}x^{n-1}a^1 + \ldots + \binom{n}{n}a^n \\ &= x^n + \frac{n!}{1!(n-1)!}x^{n-1}a + \ldots + a^n.\end{aligned}$$

the coefficients of the middle terms are given by $\frac{n!}{k!(n-k)!}$ for $1 \leq k < n$. Thus $k!$ and $(n-k)!$ contain no terms large enough to cancel n in the numerators of the factorial expansion. Further, since n is prime it remains undivided. Thus the middle coefficients can be rewritten $\frac{(n-1)!}{k!(n-k)!} \cdot n \equiv 0 \pmod{n}$. By Theorem 1, that we proved earlier we know that $a^n \equiv a \pmod{n}$ since $a^n = a^{n-1} \cdot a$ and $a^{n-1} \equiv 1 \pmod{n}$ if n is prime. So we can write $(x+a)^n \equiv x^n + a \pmod{n}$ if n is prime □

The more general statement of Theorem 7 can be seen using the multinomial coefficient.

Proof. Consider the general polynomial $(a_m x^m + \ldots + a_0)^n$. The multinomial expansion looks like

$$\sum_{k_1,k_2,\ldots,k_m} \binom{n}{k_1 k_2 \ldots k_m} (a_m x^m)^{k_1} (a_{m-1}x^{m-1})^{k_2} \ldots$$

each term has the following as its coefficient

$$\binom{n}{k_1 k_2 \ldots k_m} = \frac{n!}{k_1!k_2!\ldots k_m!} \text{ where } \sum_{i=1}^{n} k_i = n.$$

But if n is prime the only way that the n in the numerator cancels is if one of the k's is n and the rest are zeroes. Otherwise, n remains uncanceled and this makes those terms congruent to 0 mod n. Those terms drop out and so only those individual terms raised to the n will be left. It should be noted that the coefficients $a_m \to a_0$ will be raised to the n but again by Theorem 1 we know that $a^n \equiv a \pmod{n}$ if n is prime. □

Again we will only be concerned with the simpler equation(8) from above. It turns out that the converse of this statement is also true. Therefore if equation(8) holds for any value of a such that $gcd(n, a) = 1$ then n must be prime. Unfortunately if n is large, and generally it will be, $(x + a)^n$ is a long expansion calculation but if we consider $(x + a)^n$ with $f(x) \in \mathbb{Z}[x]$ an arbitrary monic polynomial then

$$(x + a)^n \equiv x^n + a \pmod{f(x), n},$$

holds if n is prime for every integer a. If the degree $f(x)$ is not too large the time to calculate this congruence will not be too great. Unfortunately, while this congruence is necessary for the primality of n it is not sufficient. So by adding the polynomial we significantly increase speed but we lose our complete guarantee of primality.

For the following theorem we will consider $f(x) = x^r - 1$.

Theorem 8 (Agrawal, Kayal, Saxena)**.** *Suppose $n \in \mathbb{Z}, n \geq 2$ and $r \in \mathbb{Z}^+$ coprime to n such that the order of n in $\mathbb{Z}_r^*$ is larger than $\lg^2 n$ and $(x+a)^n \equiv x^n + a \pmod{x^r - 1, n}$ holds for each $a \in [0, \sqrt{\phi(r)} \lg n]$. If n has a prime factor $p > \sqrt{\phi(r)} \lg n$ then $n = p^m$ for some $m \in \mathbb{Z}^+$.*

This implies that if n has no prime factors in $[2, \sqrt{\phi(r)} \lg n]$ and n is not a proper power (that is a square, cube, etc.) then n must be prime.

Proof. Consider a composite n with a prime factor $p > \sqrt{\phi(r)} \lg n$. Let

$$G = \left\{ g(x) \in \mathbb{Z}_p[x] \mid g(x)^n \equiv g(x^n) \pmod{x^r - 1} \right\}.$$

By our assumptions $x + a \in G$. Further we can see that G is closed under multiplication. Take $f(x), g(x) \in G$

$g(x)^n \equiv g(x^n) \pmod{x^r - 1}$.
$f(x)^m \equiv f(x^m) \pmod{x^r - 1}$.
Set $k(x) = f(x) \cdot g(x)$

$$\begin{aligned} k(x)^n = (f(x) \cdot g(x))^n &= f(x)^n \cdot g(x)^n \\ &= f(x^n) \cdot g(x^n) = k(x^n) \pmod{x^r - 1}. \end{aligned}$$

So $k(x) \in G$.
This implies that every monomial expression of the form

$$\prod_{0 \leq a \leq \sqrt{\phi(r)} \lg n} (x + a)^{\epsilon_a} \text{ where } \epsilon_a \in \mathbb{Z}^+, \tag{9}$$

is in G by closure under multiplication.

Further since $p > \sqrt{\phi(r)} \lg n$ these polynomials are distinct and and non-zero in $\mathbb{Z}_p[x]$. Indeed, they are all non-zero because p is prime and so $\mathbb{Z}_p[x]$ is a principal ideal domain. That is, there are no zero divisors ($a, b \in \mathbb{Z}_p[x]$ with $a \neq 0, b \neq 0 \Rightarrow a \cdot b \neq 0$). Since $\mathbb{Z}_p$ is a principal ideal domain each of its elements factors uniquely into irreducible elements. So we will only get repeated elements of $x + a$ in our factorization if $x + (a + p)$ also occurs in our factorization but $a < p$ so no elements are repeated in the factorization.

We now show that G is a union of residue classes modulo $x^r - 1$. That is, if $g(x) \in G$ and $f(x) \in \mathbb{Z}_p[x]$ and $f(x) \equiv g(x) \pmod{x^r - 1}$ then $f(x) \in G$.

Examine, $f(x) \equiv g(x) \pmod{x^r - 1}$ and replace x with x^n. we obtain $g(x^n) \equiv f(x^n) \pmod{x^{nr} - 1} (*)$ but we know that $x^r - 1 \mid x^{nr} - 1$ so this congruence holds $\bmod (x^r - 1)$ as well. We see that

$$\begin{aligned} \underbrace{f(x)^n \equiv g(x)^n}_{\text{by closure of congruence}} & \pmod{x^r - 1} \\ \underbrace{\equiv g(x^n)}_{\text{by } g(x) \in G} & \pmod{x^r - 1} \\ \underbrace{\equiv f(x^n)}_{\text{by } (*)} & \pmod{x^r - 1}. \end{aligned}$$

Let

$$J = \left\{ j \in \mathbb{Z}^+ \mid g(x)^j \equiv g(x^j) \pmod{x^r - 1} \text{ for each } g(x) \in G \right\}.$$

By definition of $G, n \in J$. Trivially $1 \in J$. $p \in J$ since p is prime and for every $g(x) \in \mathbb{Z}_p[x]$, $g(x)^p = g(x^p)$ under our modulus by Theorem 7. We know that the relation in the theorem must hold $\bmod x^r - 1$ for every g including those elements in G since p is prime. J is closed under multiplication. Take $j_1, j_2 \in J$ and $g(x) \in G$. $g(x)^{j_1} \in G$ since G is closed under multiplication. Since $g(x)^{j_1} \in G$ this implies that $g(x)^{j_1} \equiv g(x^{j_1}) \pmod{x^r - 1}$. Then $g(x^{j_1}) \in G$ since G is the union of residue classes. So since $j_2 \in J$

$$\begin{aligned} \underbrace{g(x)^{j_1 j_2} \equiv g(x^{j_1})^{j_2}}_{\text{since } g(x)^{j_1} \in G} & \pmod{x^r - 1} \\ \underbrace{\equiv g((x^{j_1})^{j_2})}_{\text{since } g(x^{j_1}) \in G} & \pmod{x^r - 1} \\ \equiv g(x^{j_1 j_2}) & \pmod{x^r - 1}. \end{aligned}$$

So $j_1 \cdot j_2 \in J$.

Let K be the splitting field for $x^r - 1$ over the finite field $\mathbb{F}_p$. The splitting field is the smallest field which contains the roots of $x^r - 1$. Since K is over $\mathbb{F}_p$ it is of characteristic p. That is, arithmetic is done mod p and it contains the r-th roots of unity. Let $\zeta \in K$ be the primitive r-th root of 1. ζ, then, is a generator for the roots of unity.

Let $h(x) \in \mathbb{F}_p[x]$ be the minimum (irreducible) polynomial for ζ, so that $h(x)$ is also an irreducible factor of $x^r - 1$.

$$K = \mathbb{F}_p(\zeta) = \left\{a_0 + a_1\zeta + \ldots + a_{k-1}\zeta^{k-1} \mid a_i \in \mathbb{F}_p\right\}, \tag{10}$$

where k is the degree of $h(x)$.

Note that $\mathbb{F}_p[x]/(h(x))$ are those elements $\mathbb{F}_p[x]$ mod $h(x)$. That is, those $r(x)$ defined by $f(x) = g(x)h(x) + r(x)$ as the remainder polynomial from synthetic division. The power $r(x)$ is strictly less than $deg(h(x)) = k$. We are guaranteed a primitive root in our splitting field since $(r, n) = 1$ which means $(r, p) = 1$ and if this holds then the set of roots will be cyclic. Therefore, it contains a generator which we call ζ. The roots are generated by $\mathbb{F}_p(\zeta) = \left\{a_0 + a_1\zeta + \ldots + a_{k-1}\zeta^{k-1} \mid a_i \in \mathbb{F}_p\right\}$ because powers of $\zeta^a, a \geq k$ can be expressed in terms of this polynomial. Consider, $\mathbb{F}_5$ and $r = 6$

$$x^6 - 1 = (x - 1)(\underline{x^2 + x + 1})(x + 1)(\underline{x^2 - x + 1}).$$

both of these factors are irreducible over $\mathbb{F}_5$. If they were reducible they would contain zeros over elements of $\mathbb{F}_5$, but it is easy to check that it does not. So we form K as follows

$$\mathbb{F}_5[x]/(x^2 - x + 1) = \left\{a_0 + a_1\zeta + a_2\zeta^2 \mid a_0, a_1, a_2 \in \mathbb{F}_5\right\}.$$

Clearly ζ, and ζ^2 are in $\mathbb{F}_5$, but if $x^2 - x + 1$ is our minimum polynomial and ζ is its root, we know $\zeta^2 - \zeta + 1 = 0$. Multiply through by ζ and we get $\zeta^3 - \zeta^2 + \zeta = 0 \Rightarrow \zeta^3 = \zeta^2 - \zeta$. Further if we multiply through by ζ^2 we get $\zeta^4 - \zeta^3 + \zeta^2 = 0 \Rightarrow \zeta^4 - \zeta^2 + \zeta + \zeta^2 = 0 \Rightarrow \zeta^4 = -\zeta$ etc. So the powers of ζ will be covered and therefore all of the zeros will be in the field. Equation(10) gives us the way to form our splitting field K.

We know that $\mathbb{F}_p(\zeta) \cong \mathbb{F}_p[x]/(h(x))$. Consider the function

$$\Psi : \mathbb{F}_p[x] \longrightarrow \mathbb{F}_p(\zeta).$$

The image of $\Psi(f) = f(\zeta)$ We know that

$$f \in ker\Psi \Leftrightarrow f(\zeta) = 0.$$

but $f(\zeta)$ is only zero if $h(x) \mid f(x)$. This is because $h(x)$ is the minimum polynomial over $\mathbb{F}_p$ for ζ, and therefore $h(x)$ divides all polynomials for which ζ is a root.

$$h(x) \mid f(x) \Leftrightarrow f \in (h(x)),$$

where $(h(x))$ is the ideal ring generated by $h(x)$. Indeed, if we know that $h(x) \mid f(x)$ then $f(x) = h(x) \cdot g(x)$ for some $g(x) \in \mathbb{F}_p[x]$, but $(h(x))$ denotes the ideal generated by $h(x)$ over $\mathbb{F}_p[x]$. That is, the ideal generated in the manner $h(x) \cdot \mathbb{F}_p[x]$. We know that $g(x)$ is and element of $\mathbb{F}_p[x]$, so $f(x) \in (h(x))$. Because $h(x)$ is irreducible $(h(x))$ is a maximal ideal and is therefore a field so it is a division ring. Therefore, $f \in (h(x))$ implies $h(x) \mid f(x)$. This means that the $ker\Psi$ is exactly the maximal ideal $(h(x))$. Ψ is clearly onto since plugging ζ in will generate our set. So by the Fundamental Theorem of Ring Homomorphisms $\mathbb{F}_p[x]/(h(x)) \cong \mathbb{F}_p(\zeta)$

We know that K is the homomorphic image of the ring $\mathbb{F}_p[x]/(x^r - 1)$. This is because our function Ψ is onto K and $h(x) \mid x^r - 1$. Thus we see that $\mathbb{F}_p[x]/(x^r - 1) \subset \mathbb{F}_p[x]/(h(x))$ since clearly division over a larger degree polynomial is a smaller set. So the image of the coset x sent to ζ is K.

Let $\overline{G}$ denote the image of G the homomorphism taking the coset of x from $\mathbb{F}_p[x]/(x^r - 1)$ and applying ζ to it.

$$\overline{G} = \big\{\gamma \in K \mid \gamma = g(\zeta) \text{ for some } g(x) \in G\big\}.$$

Further, let d denote the order of the subgroup of $\mathbb{Z}_r^*$ generated by n and p where $\mathbb{Z}_r^* = \big\{t \mid (t, r) = 1\big\}$ in mod r space (to say generated by n and p of course means elements of the form $n^a p^b$, $a, b \in \mathbb{Z}$). Now, let

$$G_d = \big\{g(x) \in G \mid g(x) = 0 \text{ or } deg(g(x)) < d\big\}.$$

$d \leq \phi(r) < r$ since $\phi(r)$ is the number of integers less than r that are relatively prime to r. The elements of G_d must be distinct mod $(x^r - 1)$ since their degree is strictly less than $d < r$. We want to show that the homomorphism to K is one-to-one when restricted to G_d.

Consider $g_1(x), g_2(x) \in G_d$ and say that they map to the same element on the homomorphism. That is, $g_1(\zeta) = g_2(\zeta)$ $(*)$. If $j = n^a p^b$ for $a, b \in \mathbb{Z}^+$, then $j \in J$ since $n, p \in J$ and J is closed under multiplication. So

$$\begin{gathered}
\underbrace{g_1(\zeta^j) = g_1(\zeta)^j}_{\text{since } g_1 \in G \text{ and } j \in J} \\
\underbrace{= g_2(\zeta)^j}_{\text{by } (*) \text{ and closure of } G} \\
\underbrace{= g_2(\zeta^j)}_{\text{since } g_2 \in G \text{ and } j \in J} .
\end{gathered}$$

the equalities are given because these elements are equal in K. This equality holds for d distinct values of j mod r since d is the order of the subgroup

generated by $n^a p^b \bmod r$. But since ζ is a primitive r-th root of 1, ζ^j is distinct as long as j is distinct mod$r$. Thus $g_1(x) - g_2(x)$ has at least d distinct roots in K, since plugging in ζ to powers of j will yield $g_1(\zeta^j) = g_2(\zeta^j)$. In a field, the number of roots for a polynomial is bounded by the polynomials degree. Since, $g_1(x), g_2(x) \in G_d$ their degree is strictly less than d therefore $g_1(x)$ must be equal to $g_2(x)$.

Hence, restricting the domain of our homomorphism to G_d we have our one-to-one correspondence with elements in K, specifically all of $\overline{G}$.

Consider, then, the polynomials

$$g(x) = 0 \text{ or } g(x) = \prod_{0 \le a \le \sqrt{d}\lg n} (x+a)^{\epsilon_a}.$$

where each ϵ_a is 0 or 1. Because $d \le \phi(r)$, and $\phi(r)$ was our previous bound for a in the functions defined in equation(9) we know that $g(x) \in G$. Moreover, we know that $d > \lg^2 n$ by our assumption that the order of n is greater than $\lg^2 n$ in $\mathbb{Z}_r^*$. So

$$d > \lg^2 n \Rightarrow d^2 > d\lg^2 n \Rightarrow d > \sqrt{d}\lg n.$$

If we do not choose all the ϵ_a to be one the exponent on $(x+a)$ will not exceed d and therefore $g(x) \in G_d$ for all $g(x)$ of this form. Because each $g(x)$ is distinct there are at least

$$\underbrace{1}_{g(x)=0} + \Big(\underbrace{2^{\lfloor \sqrt{d}\lg n \rfloor + 1}}_{\text{powers } 0 \le a \le \sqrt{d}\lg n} - \underbrace{1}_{\text{case of all } \epsilon_a = 1} \Big) > 2^{\sqrt{d}\lg n} = n^{\sqrt{d}},$$

members in G_d and due to the one-to-one correspondence $G_d \leftrightarrow \overline{G}$ there are more than $n^{\sqrt{d}}$ members of $\overline{G}$.

Recall that $K \cong \mathbb{F}_p[x]/(h(x))$. If we denote the $deg(h(x)) = k$ then $K \cong \mathbb{F}_{p^k}$. So take j, j_0 such that $j \equiv j_0 \pmod{p^k - 1}$. If we eliminate zero from K the order of K is $p^k - 1$ by our isomorphism. But fields with 0 removed are cyclic and K has $p^k - 1$ elements, if 0 is removed, thus for a $\beta \in K$, $\beta^j = \beta^{j_0} \Leftrightarrow j \equiv j_0 \pmod{p^k - 1}$. Let

$$J' = \big\{ j \in \mathbb{Z}^* \mid j \equiv j_0 \pmod{p^k - 1} \text{ for some } j_0 \in J \big\}.$$

If $j \equiv j_0 \pmod{p^k - 1}, j_0 \in J$ and $g(x) \in G$ then

$$\underbrace{g(\zeta)^j = g(\zeta)^{j_0}}_{\text{since } g(\zeta) \in K}$$
$$\underbrace{= g(\zeta^{j_0})}_{\text{since } g(x) \in G \text{ and } j_0 \in J}$$
$$\underbrace{= g(\zeta^j)}_{\text{since } g(\zeta^{j_0}) \in K} .$$

Since J is closed under multiplication so is J' since J''s members are congruence classes of the $j_0 \in J$ and congruence is closed under multiplication. Here we recognize that $n/p \in J'$ since $np^{k-1} \equiv n/p \pmod{p^k - 1}$ and $np^{k-1} \in J$ since $n, p \in J$ and J is closed under multiplication. This implies that

$$\begin{aligned} np^{k-1} - n/p &= m(p^k - 1) \\ \frac{np^k - n}{p} &= \\ \frac{n(p^k - 1)}{p} &= m(p^k - 1). \end{aligned}$$

Clearly m is well defined since $p \mid n$ by our original assumptions. Thus we know that J' contains 1, p trivially, and n/p. Also, for each $j \in J', g(x) \in G$ we have $g(\zeta)^j = g(\zeta^j)$

Finally, consider the integers $p^a(n/p)^b$ where a, b are integers in $[0, \sqrt{d}]$. $p, n/p$ are in the ordered-d subgroup of $\mathbb{Z}_r^*$ generated by p and n. p^{-1} is an element because subgroups have well-defined inverse and p generates $\mathbb{Z}_r^*$ with n. Now there are more than d choices for a and b, that is, if a and b range from 0 to $\sqrt{d}$ there are exactly $(\sqrt{d}+1)^2 = d + 2\sqrt{d} + 1$ combinations of a and b. So there must be two choices $(a_1, b_1), (a_2, b_2)$ with $j_1 = p^{a_1}(n/p)^{b_1}$ and $j_2 = p^{a_2}(n/p)^{b_2}$ congruent mod r. These j_1, j_2 exist because the order of $\mathbb{Z}_r^*$ is d so the generation of more than d elements implies repeated values and since ζ is a primitive r-th root of unity, $\zeta^{j_1} = \zeta^{j_2} (@)$ due to their congruence. Further, j_1, j_2 are elements of J' since $p, n/p \in J'$ and J' is closed under multiplication. So

$$\underbrace{g(\zeta)^{j_1} = g(\zeta_1^j)}_{\text{since } j_1 in J'} \underbrace{= g(\zeta^{j_2})}_{\text{by the equality } (@)} \underbrace{= g(\zeta)^{j_2}}_{\text{since } j_2 \in J'}.$$

This holds for all $g(x) \in G$. This implies that $\gamma^{j_1} = \gamma^{j_2}$, <u>for all</u> $\gamma \in \overline{G}$ since γ corresponds to a $g(\zeta)$ for some $g(x) \in G$. However, $\overline{G}$ has more than $n^{\sqrt{d}}$ elements and $j_1, j_2 \leq p^{\sqrt{d}}(n/p)^{\sqrt{d}} = n^{\sqrt{d}}$. Thus, the polynomial $x^{j_1} - x^{j_2}$ has all $\gamma \in \overline{G}$ as roots. But the cardinality of $\overline{G} > n^{\sqrt{d}}$. So $x^{j_1} - x^{j_2}$ has too many roots since the equation is of degree $\leq n^{\sqrt{d}}$. So it must be the zero

polynomial. Therefore, $j_1 = j_2$ which implies

$$\begin{aligned} p^{a_1}(n/p)^{b_1} &= p_2^a(n/p)^{b_2} \\ p^{a_1}n^{b_1}p^{-b_1} &= p^{a_1}n^{b_2}p^{-b_2} \\ n^{b_1-b_2} &= p^{a_2-a_1-b_2+b_1}. \end{aligned}$$

Since our pairs (a_1, b_1) and (a_2, b_2) are distinct we know that $b_1 \neq b_2$. This is because if $b_1 = b_2$

$$n^{b_1-b_2} = n^0 = 1 = p^{b_1-b_2-a_1+a_2} = p^{a_2-a_1}. \tag{11}$$

Equation(11) is equal to 1 if and only if $a_2 - a_1 = 0$ which means that $a_2 = a_1$ If this were true then the pairs would not be distinct. So $b_1 \neq b_2$. We have, then, n expressed in terms of p. By unique factorization in $\mathbb{Z}$, n must be a power of p □

Theorem 8 demonstrates that correctness of the AKS algorithm. We first check that n is not a proper power. If it is, then it is obviously composite. If not we find our r and check to see if n has a factor over the interval $[2, \sqrt{\phi(r)}\lg n]$. If it does have a factor in this interval then, again, we know that it is clearly composite because we have found a factor. The last step is to check the binomial congruences. As discussed, the congruences $(x+a)^n \equiv x^n + a \pmod{f(x), n}$ must hold for all a if n is prime. Clearly this holds for the less general congruence, $(x + a)^n \equiv x^n + a \pmod{x^r - 1, n}$. So over the interval $[1 \leq a \leq \sqrt{\phi(r)}\lg n]$ if the binomial congruences do not hold for any a then n is composite. However, by Theorem 8 if it holds for all $a \in [1 \leq a \leq \sqrt{\phi(r)}\lg n]$ and n has a prime factor $p > \sqrt{\phi(r)}\lg n$ (if n is composite we've already ruled out factors less than $\sqrt{\phi(r)}\lg n$ in the previous step) then it must be a proper power of that prime. We have already checked that n is not a proper power so n must be prime.

We have seen that the AKS algorithm works and is a very elegant algorithm that will deterministically tell us whether or not a number is prime. On the other hand it seems very complex and actually involves doing some trial division which we had shied away from because of its time constraint difficulties. The algorithm is significantly more complex than the Strong Pseudoprime test and can appear less attractive because of this. While the algorithm is more complex it relies on deep number theory to carve out its result and thus provides us with a unique and outstanding approach. As mentioned before the AKS algorithm runs in fully proven polynomial time. What follows is a loose computation of the upper bound of the AKS' time complexity that demonstrates a polynomial runtime.

The time complexity of the primality test relies on checking the congruences $(x+a)^n \equiv x^n + a \pmod{x^r - 1, n}$ but first we need to show that our search for r can be done in polynomial time in $\ln n$. This bounding is obtained through application of the following theorem.

Theorem 9. *Given an integer $n \geq 3$, let r be the least integer with the $o(n)$ in $\mathbb{Z}_r^*$ exceeding $\lg^2 n$. Then $r \leq \lg^5 n$.*

Proof. Let r_0 be the least prime that does not divide

$$N = n(n-1)(n^2-1)\dots(n^{\lfloor \lg^2 n \rfloor} - 1).$$

Since r_0 is prime and does not divide N we know that $r_0 \nmid n$. So $(r_0, n) = 1$ which implies $n \in \mathbb{Z}_{r_0}^*$. We know that, $n^2 \neq 1$ in $\mathbb{Z}_{r_0}^*$. Say that $n^2 \equiv 1 \pmod{r_0} \Rightarrow n^2 - 1 = kr_0 \Rightarrow r_0 \mid n^2 - 1$. But if $r_0 \mid n^2 - 1$ then $r_0 \mid$. Thus we derive a contradiction. So $n^2 \neq 1$ in $\mathbb{Z}_{r_0}^*$. This process can be continued up to the last factor, $(n^{\lfloor \lg^2 n \rfloor} - 1)$. $n^{\lfloor \lg^2 n \rfloor} \neq 1$ in $\mathbb{Z}_{r_0}^*$ since if it were $n^{\lfloor \lg^2 n \rfloor} - 1 = kr_0$. By the same reasoning as above, $n^{\lfloor \lg^2 n \rfloor} \neq 1$ in $\mathbb{Z}_{r_0}^*$. This implies that $o(n)$ in $\mathbb{Z}_{r_0}^*$ is greater than $\lg^2 n$ and $r \leq r_0$ since it is the least integer such that $o(n) > \lg^2 n$ in $\mathbb{Z}_{r_0}^*$.

Note that the product of primes dividing N is at most N clearly since either N decomposes into a unique factorization of primes of power 1 or it has repeated prime factors and the product of the unique primes will be strictly less than N. Further, we know that $N < n^{1+1+2+\dots+\lg^2 n}$ because

$$n(n-1)(n^2-1)\dots(n^{\lfloor \lg^2 n \rfloor} - 1) < n(n)(n^2)\dots(n^{\lfloor \lg^2 n \rfloor}).$$

Further, $n^{1+1+2+\dots+\lfloor \lg^2 n \rfloor} = n^{1/2\lfloor \lg^2 n \rfloor^2 + 1/2\lfloor \lg^2 n \rfloor + 1}$. This is by the summation formula $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$. Applying this yields

$$1+2+3+\dots+\lfloor \lg^2 n \rfloor = (\lfloor \lg^2 n \rfloor + 1) \cdot 1/2\lfloor \lg^2 n \rfloor = 1/2\lfloor \lg^2 n \rfloor^2 + 1/2\lfloor \lg^2 n \rfloor.$$

We add one to our sequence because it is repeated in the exponent sequence. So we get, $n^{1/2\lfloor \lg^2 n \rfloor^2 + 1/2\lfloor \lg^2 n \rfloor + 1}$, which is clearly less than $n^{\lg^4 n} = 2^{\lg^5 n}$. Hence we get

$$N < n^{1+1+2+\dots+\lfloor \lg^2 n \rfloor} = n^{1/2\lfloor \lg^2 n \rfloor^2 + 1/2\lfloor \lg^2 n \rfloor + 1} < n^{\lg^4 n} = 2^{\lg^5 n}.$$

Now we take into consideration the Chebyshev-type estimate, see [Crandall and Pomernace 2005], on the product of primes less than a given x

$$\prod_{p \leq x} p > 2^x, \text{ where } x \geq 31.$$

That is, the product of primes $[1, x]$ exceeds 2^x if $x \geq 31$. Consider our $N < 2^{\lg^5 n}$. The product of the prime factors of N are less than or equal to N. If we choose $x = \lg^5 n$, then the product of primes less than or equal to x exceeds $2^{\lg^5 n}$ by the Chebyshev-type estimate. $2^{\lg^5 n}$ is strictly greater than N so this list of primes less than or equal to x in the product contains non-factor primes of N. Choose r_0 to be the least of said primes. Therefore $r_0 \leq \lg^5 n$ if $\lg^5 n \geq 31$. As long as $n \geq 4, lg^5 n \geq 31$. If $n = 3$ the least r is 5 since in $\mathbb{Z}_5^*$

$$\begin{aligned} 3^5 = 243 \bmod 5 = 3 \\ \lg^2 3 = 2.51211 < 5 \\ 5 \leq \lg^5 n = \lg^5 3 = 10.00218. \end{aligned}$$

□

The powers test in step 1 of the AKS algorithm can be done very simply with a kind of binary search approach. This approach involves solving $a^b - n = 0$ by a simple educated guessing. For example if we are trying to determine if n is a square. We deal with the equation $a^2 - n = 0$. We first guess the prime p_1 closest to $n/2$. We plug it in and check to see if it is n. If it is higher than n, we guess p_2 to be the closest prime to $p_1/2$. If it is lower we guess p_2 to be the closest prime to $(n - p_1)/2$. We repeat this until we find a p value that works or the high and low bounds of our search converge so that no p exists. This application of binary search is bounded by $O(\lg n)$. We will perform this for k power steps. We know $2^k < a^k = n$. So $k < \lg n$. So this algorithm to perform step 1 is bounded by $O(\lg^2 n)$.

Our proof of finding an integer r with order of n in $\mathbb{Z}_r^*$ exceeding $\lg^2 n$ in polynomial time shows that step 2. can be done in polynomial time. This is because r is bounded by $\lg^5 n$ by Theorem 9 and we need only check the primes up to $\sqrt{\phi(r)} \lg n$ as factors of n.

As discussed earlier, the third step is the most crucial. We have done the first two steps in polynomial time and have set the necessary groundwork to fit Theorem 8. When discussing the congruence testing earlier we mentioned that the expansion of $(x + a)^n$ is very expensive and time consuming. We added the extra modulus of a monic polynomial with the intention of decreasing the expansions time complexity. The time saved can be seen by considering an example. Take the polynomial $(x + 1)^n$ to be expanded and examine the time to check the congruence associated with it.

Ex 2. *If* $n = 10001$ *and* $r = 3$
$(x + 1)^{10001} \pmod{x^3 - 1, 10001}$. *This can be expanded very quickly by repeated squaring and modulus reduction.* $(x + 1)^2 = x^2 + 2x + 1$ *reduce the*

polynomial mod x^3-1 *and the coefficients* mod 10001. $(x+1)^4 = (x^2+2x+1)^2 = (x^4+4x^3+6x^2+4x+1)$ *reducing* mod x^3-1 *we get,* $6x^2+5x+5$. *Now,* $(x+1)^8 = (6x^2+4x+1)^2$ *and reduce* mod x^3-1. *We can continue this process to obtain the complete expansion under the modulus conditions.*

The expansion will never be larger than degree r because of our modulus. This is why we take the least r to create our monic polynomial, if the size of r is not too large it greatly reduces the number of terms to be expanded at each squaring step.

The number of repeated squaring steps that we will have to perform is approximately $\lg n$. At each step we must first expand our polynomial which requires at most r^2. This is because our polynomial has at most r terms because it is bounded in degree by r. After expanding our polynomial at each squaring step we must reduce it $\mod(x^r-1)$ by synthetic division. Synthetic division is bounded by $O(\ln^2 n)$. We must also reduce the coefficients of our polynomials. This takes approximately $2r \lg n$ steps. Now we can derive our complete bounding. Our approximate step count is $\lg n(r^2 \lg^2 n + 2r \lg n) = r^2 \lg^3 n + 2r \lg^2 n$. This leads to a complexity estimate of $O(r^2 \lg^3)$. If we consider that these congruences are checked over the interval $0 \leq a \leq \sqrt{\phi(r)} \lg n$ and $\sqrt{\phi(r)}$ is bounded by $r^{1/2}$, this complexity is repeated, less than, $r^{1/2} \lg n + 1$ times. So we can say that the time complexity for the congruences is $O(r^{2.5} \ln^4 n)$. Using our result from Theorem 9, $r = O(\ln^5 n)$. Thereby we observe a very rough estimate of $O(\ln^{16.5} n)$. This bounding is large but it is clearly and definitively polynomial. For a more precise time bounding estimate see [Crandall and Pomernace 2005]. With optimization and tighter bound criterion the actual time-complexity bound for the AKS algorithm is significantly lower. However, the fact of the matter still remains, even under naive upper bound assumptions and overestimation of the time needed for the algorithms steps, the algorithm still exhibits polynomial-timed characteristics.

The AKS primality test was a breakthrough in primality testing and number theory. The algorithm, in its entirety, is much simpler than many of its predecessors that attempted to approach deterministic polynomial run-time. Finally its proof, while relying on concepts in algebra, is complete and understandable. Nonetheless, while the AKS primality test is deterministic and polynomial it still remains to be seen whether it will be effective in proving primality under more rigorous time constraints. As mentioned before, the strong pseudoprime test is still widely used in RSA key generation. This is because the cryptographic key process often demands the generation of large primes for key creation in a very small amount of time. Under such conditions the speed of algorithms, like the Miller-Rabin test, outweigh the

miniscule chance of a false negative in proving compositeness.

3 Factorization

While primality testing is important to the creation of RSA keys, factorization is equally important to those systems' security. Primality contributes to the security of the RSA system with what it can do and how fast. Factorization is important for what it cannot do quickly. Currently there are no polynomial-time algorithms to factor a large integer n into its prime factors. It is believed that there is no deterministic polynomial factorization algorithm. This is extremely important to the RSA security system as mentioned earlier. If factorization were easy then the decryption key d can be reverse engineered from the known factorization of n and the public encryption key e. The problem of factoring n, if n is large, it is so difficult in fact that the largest "hard" number factored to date in a regulated setting is RSA-200. A "hard" number to factor is one of the form $n = p \cdot q$ where p and q are large primes. It is no coincedence that numbers used in the public key system take this form. RSA-200 was one of the RSA challenge numbers. It is the largest of these challenge numbers factored to date at 200 digits long and composed of two 100 digit primes. It was factored in 2005 using state of the art techniques and a cluster of more than 80 2.2 GHz processing computers. Even with this parallel computing strategy it took nearly two years to split the number according to the note released announcing the accomplishment [Kleinjung Franke Boehm and Bahr 2005].

In order to understand some of the more complex factorization algorithms we will need to look at the Chinese Remainder theorem.

Theorem 10. *Let $m_1, m_2, \ldots m_r$ be positive integers that are pairwise relatively prime, that is, no two m_i's share a common factor other than one. Let $a_1, a_2, \ldots a_r$ be integers, then there exists $a \in \mathbb{Z}$ such that*

$$\begin{aligned} a &\equiv a_1 \pmod{m_1} \\ a &\equiv a_2 \pmod{m_2} \\ &\vdots \\ a &\equiv a_r \pmod{m_r}. \end{aligned}$$

Moreover, a is unique mod $M = m_1 \cdot m_2 \cdot \ldots \cdot m_r$.

Proof. Define $M_i = (M/m_i)^{\phi(m_i)}$.
We know that M/m_i is relatively prime to m_i. This is because

$$M/m_i = (m_1 \cdot m_2 \cdot \ldots \cdot m_{i-1} \cdot m_{i+1} \cdot \ldots \cdot m_r)^{\phi(m_i)}.$$

None of these shares a common factor with m_i since they are pairwise relatively prime. Moreover, M/m_i is divisible by m_j for all $j \neq i$. So we have that

$$\begin{aligned} M_i &\equiv 1 \pmod{m_i} \\ M_j &\equiv 0 \pmod{m_j} \qquad \text{for } j \neq i. \end{aligned}$$

We proceed by forming a. So let $a = a_1 \cdot M_1 + a_2 \cdot M_2 + \ldots + a_r \cdot M_r$. If we construct a in said manner it satisfies all of our modulo criteria. Consider

$$\begin{aligned} a \bmod m_1 &= a_1 \cdot M_1 + a_2 \cdot M_2 + \ldots + a_r \cdot M_r \bmod m_1 \\ &= a_1 \cdot M_1 + 0 + \ldots + 0 \bmod m_1 \\ &= a_1 \cdot 1 \bmod m_1 \\ &= a_1 \bmod m_1 \\ &\Rightarrow a \equiv a_1 \pmod{m_1}. \end{aligned}$$

a is unique. If it were not we would have another integer b satisfying our congruence. Then for each $m_i, a \equiv b \pmod{m_i}$ implies $m_i \mid a - b$. If this is true of every i we know that $M \mid a - b$ (this is because each of the m_i have no common factors and therefore make up unique factors of $a - b$). This implies $a \equiv b \pmod{M}$. Therefore a is unique mod M. □

The first instinct in factorization is the same first instinct that one has when approaching primality testing: trial division. While this process seems quite quick for small numbers it has obvious drawbacks as the numbers get larger. As discussed earlier, even with a large list of primes and testing only those primes less than $\sqrt{n}$, we must still perform a large number of division steps which grows very quickly as n increases. The first algorithm to consider is not used very often anymore but is an improvement on trial division and contains the logical foundations of the more modern techniques.

Fermat's algorithm begins by trying to split an integer n into the integer product of $a \cdot b$. If this can be done, then a and b can be broken again into factors, if indeed they are composite, more quickly than n. Thus, a prime factorization can be iteratively determined. The idea behind the algorithm is to write n as the difference of two perfect squares.

$$n = x^2 - y^2 = (x - y)(x + y)$$

Hence we have two smaller factors of n. Further, if n is odd (which it will be most of the time, since if n is even we divide by a power of 2 to obtain

an odd n' to factor) then every n can be expressed as such. Indeed, consider $n = a \cdot b$ and $x = \frac{(a+b)}{2}, y = \frac{(a-b)}{2}$

$$\begin{aligned} x^2 - y^2 &= \frac{a^2 + 2ab + b^2}{4} - \frac{a^2 - 2ab + b^2}{4} \\ &= \frac{a^2 - a^2 + 4ab + b^2 - b^2}{4} \\ &= a \cdot b \\ &= n. \end{aligned}$$

Given an odd integer n, we start with $x = \lceil \sqrt{n} \rceil$ and try increasing y until $x^2 - y^2 \leq n$. If $x^2 - y^2 = n$, we have our factorization. However, if it is less than n we reseed $x = x + 1$ and start increasing y again from 1. This algorithm can be sped up, avoiding calculating the squares, by initially setting

$$r = x^2 - y^2 - n,$$

and $u = 2x + 1, v = 2y + 1$ thereafter. u is the amount r increases as a result of $(x + 1)^2$ since x_2^2 will be equal to $x_1^2 + 2x_1 + 1 = x_1^2 + u$. Likewise, v is the amount r decreases as y is augmented $(y + 1)^2$. This avoids costly squaring in the run-time of the algorithm. Nevertheless, it requires a large number of loops, depending on n, especially if the factors of n are far away from $\lceil \sqrt{n} \rceil$. Either way, we do not perform costly trial division and if n has two large factors Fermat's algorithm will outperform trial division.

Kraitchik observed that finding a factor for n using Fermat's algorithm could be sped up by attempting to find an x and y such that

$$x^2 \equiv y^2 \pmod{n}.$$

This pair no longer guarantees a factorization of n, but it does imply that $n \mid x^2 - y^2$ which further implies that $nk = (x - y)(x + y)$. We have at least a 50-50 chance that the prime divisors of n are distributed between these two factors and therefore the $gcd(x - y, n)$ is a non-trivial factor of n. The other possibility is that all of n's factors could be in one or the other giving a $gcd(x - y, n) = 1$ or n. If we can find a systematic and efficient way to produce these x, y values then this relation gives us a good chance of finding factors of n.

Before considering the powerful application of Kraitchik's improvement on Fermat's algorithm we need to consider two algorithms that are extremely efficient at producing factors to composites on the order of 10^{10} to 10^{20}. Both algorithms are due to J.M. Pollard. The first is the Pollard Rho algorithm.

We start with a composite number n and denote a non-trivial factor of n as d. Let $f(x)$ be a small irreducible polynomial over $\mathbb{Z}_n$. Starting with an x_0, we create a list of x_i's such that

$$x_i = f(x_{i-1}) \pmod{n} \tag{12}$$

Equation(12) is the reason why it is important that $f(x)$ is irreducible over $\mathbb{Z}_n$.

Ex 3. $x_0 = 3 \qquad f(x) = x^2 + 1 \qquad n = 799$

$$\begin{array}{ll} x_0 = 3 & x_5 = 383 \\ x_1 = 10 & x_6 = 473 \\ x_2 = 101 & x_7 = 10 \\ x_3 = 614 & x_8 = 101 \\ x_4 = 668 & \ldots \end{array}$$

Let $y_i = x_i \bmod d$. *If we choose* $d = 17$, *then our sequence of* y_i*'s will be*

$$\begin{array}{ll} y_0 = 3 & y_5 = 9 \\ y_1 = 10 & y_6 = 14 \\ y_2 = 16 & y_7 = 10 \\ y_3 = 2 & y_8 = 2 \\ y_4 = 5 & \ldots \end{array}$$

We have $x_i \equiv f(x_{i-1}) \pmod{n}$ and $y_i \equiv x_i \pmod{d}$. This implies that $x_i = nk + f(x_{i-1})$ for some k. So we can write $y_i = nk + f(x_{i-1}) \pmod{d}$ but d is a factor of n so $y_i \equiv f(x_{i-1}) \pmod{d}$. There are only a finite number of congruence classes for d, that is, at most d of them. So eventually in our list of y_i's we will have a $y_i = y_j$. But once this happens we will keep cycling and for all $t \in \mathbb{Z}$

$$y_{i+t} = y_{j+t}.$$

If $y_i = y_j$ then

$$x_i \equiv x_j \pmod{d}.$$

This can be seen by $y_i = x_i \bmod d = y_j = x_j \bmod d$. So, $d \mid x_i - x_j$ and hence we have a good chance that x_i, x_j are not equal outside of $\mathbb{Z}_d$ space. If they are not, $gcd(n, x_i - x_j)$ is a non-trivial factor of n (since d divides $x_i - x_j$ and is a divisor of n). The problem is that we don't know d. If we did it would not make sense to run the algorithm. We are, nonetheless guaranteed to have infinitely many pairs of i, j for which y_i, y_j are equal since it is infinitely cyclic. If we have a cycle length c, the difference between the i and j indices for which $x_i \equiv x_j \pmod{d}$, once we are in the cycle any pair

(i, j) such that $c \mid j - i$ will work. So we just need to find the i, j combination that is within the cycle and are spaced at the cycle length. There are many ways to approach searching for this pair. The form of choosing pairs (i, j) and computing the $gcd(n, x_i - x_j)$ to follow is due to Brent. His suggestion is compute the following x_i's checking if the $gcd(x_i - x_j, n)$ is a non-trivial factor.

$$\begin{aligned} x_1 - x_3 \qquad & c = 2 \\ \\ x_3 - x_6 \qquad & c = 3 \\ x_3 - x_7 \qquad & c = 4 \\ \\ x_7 - x_{12} \qquad & c = 5 \\ x_7 - x_{13} \qquad & c = 6 \\ x_7 - x_{14} \qquad & c = 7 \\ x_7 - x_{15} \qquad & c = 8 \end{aligned}$$

In general taking pairs

$$x_{2^k - 1} - x_j, \qquad 2^{k+1} - 2^{k-1} \leq j \leq 2^{k+1} - 1.$$

With this approach we can state our algorithm completely.

Algorithm 6. *Given n, x_0, and $f(x)$, $i = 1$*

1. *$for(2^{i+1} - 2^{i-1} \leq j \leq 2^{i+1})$*
   compute $x_{2^i - 1}$ and x_j
   compute $gcd(x_{2^i - 1} - x_j, n) = g$
   $if(g \neq 1$ and $g \neq n)$
   return *g*
   else continue with the for loop
2. *$i = i + 1$ and start step 1 over*

The beauty of this method is that while the index of the first x is increasing to guarantee that we get off the tail (that is, the part of the y_i's that are not part of the cycle) the difference between i and j is increasing by one each time searching for the proper c value. This systematic approach attempts to save space while testing all the gcd's necessary to find d. This test is probabilistic and known as a Monte Carlo method because our success in factorization depends heavily on our choice of $f(x)$. In the Example 3 at the beginning we can see that we have made a bad choice for $f(x)$ since our x_i's are equal within the cycle. When we take the $gcd(x_i - x_j, n)$ we will get n. In this case, we must choose a new $f(x)$ and start again.

Ex 4. $x_0 = 3 \qquad f(x) = x^2 + 2 \qquad n = 799$

$$\begin{array}{ll} x_0 = 3 & x_5 = 640 \\ x_1 = 11 & x_6 = 514 \\ x_2 = 123 & x_7 = 300 \\ x_3 = 749 & \vdots \\ x_4 = 105 & \end{array}$$

Yielding a list of y_i's

$$\begin{array}{ll} y_0 = 3 & y_5 = 11 \\ y_1 = 11 & y_6 = 4 \\ y_2 = 9 & y_7 = 1 \\ y_3 = 1 & \vdots \\ y_4 = 3 & \end{array}$$

So the gcd$(105 - 3, 799) = 17$

With our new choice of $f(x)$ we obtain the cycle that we want and using the first repeated values we obtain the non-trivial factor of n that we wanted. As mentioned, the algorithm's approach to the solution is not quite as simple. Our examples used the fact that $d = 17$ to obtain the y congruences which help us choose the (i, j) pair. Brent's approach to the cycle testing for the Pollard-Rho algorithm will typically take a little longer to find the factor since we do not know the tail or cycle lengths and therefore do not know which (i, j) pair to use..

We implemented the Pollard-Rho algorithm to test its efficiency. In the java implementation the calculations were done using the big integer class to avoid run over in binary digits as the numbers tested grew beyond the double precision 32-bit representation. Further, the algorithm included a max number of iterations limit in order to avoid problematic run-time engendered by unlucky x_0 or $f(x)$ picks. For simplifications sake the irreducible polynomial used was $f(x) = x^2 + c$ where c was a user input integer. It is easy to see that this polynomial is always irreducible over $\mathbb{Z}_n$ and while it limits the kinds of $f(x)$'s available it offers a good way to vary the generating polynomial without having to constantly perform reducibility tests.

In practice, as the numbers grew larger the run-time of the algorithm also grew significantly. This is because the cycle length is generally dependent on the size of the smallest factor of n. If n is a two factor composite number its factors grow along with the size of n. The Pollard-Rho algorithm was efficient on numbers up to around the order of 10^{19} (that is around 19 digits long) but after that it was harder to find factors under any maximum number

of iterations in a reasonable time. While this constitutes a large number it is no where near the size needed to break our target numbers of 100 digits or more. This is also slightly under the estimate in Bressoud's Factorization and Primality Testing of 10^{20}. This is most likely due to the fact that the implemented algorithm does not utilize some of the optimization strategies like time saving gcd calculations or running out the polynomial generations before starting cycle testing to get off the tail more quickly. These suggestions would, no doubt, increase efficiency especially on larger n. While these time saving techniques were not employed we were still able to approach the limit set by Bressoud. This is most likely due to the large jumps in computing power and memory storage since the algorithm was initially proposed. These leaps in memory capacity and processing speed have benefitted the Quadratic Sieve and Multiple Polynomial Quadratic Sieve still more.

Similar results were achieved with the second algorithm suggested by Pollard. The Pollard p-1 algorithm is similar to the Pollard-Rho algorithm and shares a similar factoring threshold of about $10^{10} - 10^{20}$. The correctness of the algorithm rests on Theorem 1

$$2^{p-1} \equiv 1 \pmod{p}. \tag{13}$$

We consider our n to be factored. Suppose that n has a prime factor p such that the primes dividing $p-1$ are less than 10000. We will work with the slightly more restrictive condition that $p-1 \mid 10000!$. If this holds we can compute

$$m = 2^{10000!} \bmod n.$$

very quickly since this constitutes exponentiation mod n. Such exponentiation can be done quickly even though 10000! is a very large number, since we calculate

$$(((2^1)^2)^3 \ldots)^{10000} \bmod n,$$

where we reduce modulo n after each exponentiation calculation. One should note that $p-1 \mid 10000!$ is only a slightly larger restriction over the statement that $p-1$ has factors below 10000 since it only rules out factorizations with repeated large factors like 9999^a for some a such that $9999^a > 10000!$

If $m = 2^{10000!} \bmod n$ and we know that $10000! = r(p-1)$, these conditions imply

$$\begin{aligned}
nk + m = 2^{10000!} &\Rightarrow nk + m = 2^{r(p-1)} \\
&\Rightarrow m = 2^{r(p-1)} - nk \\
&\Rightarrow m \equiv 2^{r(p-1)} - nk \equiv 2^{r(p-1)} - 0 \pmod{p} \qquad \text{since } p \mid n \\
&\Rightarrow m \equiv 2^{r(p-1)} \equiv \underbrace{(2^{p-1})^r \equiv (1)^r}_{\text{by Equation(13)}} \equiv 1 \pmod{p}.
\end{aligned}$$

Therefore $p \mid m-1$. There is a good chance that n does not divide $m-1$ and thus $t = gcd(m-1, n)$ will be a non-trivial divisor of n. 2 is simply a special case. Referring back to Theorem 2 we see that these calculation should hold for any base. That is, our observations will hold for $b^{10000!}$ for any b relatively prime to n.

In actual implementation we have no way of telling how close we need to get to 10000 before we find our p such that $p-1$ divides 10000! and p divides n. We do not want to compute the full 10000! if we don't have to for two reasons. The first is that extra computation and wasted effort is never good for an algorithm implementation. Second, if all of n's factors are picked up by computing m fully our $gcd(m-1, n)$ will yield n which is a useless result in terms of actually splitting n. So we've done all of the computation work without coming up with a useful result. For these reasons it is best to continually check the $gcd(b^{k!}-1, n)$ and augment k, up to $10,000!$, between assessments. If the gcd is 1 then we know that we haven't picked up our primes. If it is n then we have picked up all of them and either we must subtract from k and recalculate the gcds more often or try a different base value for b. As we proved above, if the gcd is anything but 1 or n we have our non-trivial factor of n.

Algorithm 7 (Pollard P-1)**.** *Given n and base b, $i = 10$*

1. *Calculate* $m = b^{i!} \bmod n$
2. *Calculate* $t = gcd(m-1, n)$
3. *If*$(t > 1$ *and* $t < n)$
   return t
   else $i = i + 10$ *and repeat from step 1*

Like the Pollard-Rho algorithm, the Pollard p-1 algorithm assumes that n is known to be composite. Both the Pollard-Rho and p-1 algorithms can have extremely long and inefficient run-times if n is prime.

Our $p-1$ algorithm is plagued by the same problems that the Pollard-Rho algorithm had. That is, the gcd step calculation may return n as a value and therefore we will need to go back and change the base value to a different integer and retry our calculations. Even worse, if our initial assumption of $p-1$ having factors of less than $10,000$ does not actually hold. We may never be able to find a factor of n and this algorithm will continue indefinitely without producing a usable result. To handle this we could increase our value of $10,000$, but this spreads our calculations farther and farther, slowing the algorithm.

It should be noted that there is an extra caveat on RSA key generation that both primes, p and q, used to create the keys have the property that

$p-1$ and $q-1$ have large factors for just this reason. If these two values had small integer factors then n can be factored very quickly by the $p-1$ algorithm. To form such p and q values we start with a large prime value p_1 and q_1 such that $p = 2p_1 + 1$ and $q = 2q_1 + 1$ are also prime. Forming p and q in this manner makes them resistant to quick factorization by the $p-1$ method. For more on this method and formation of cryptographic keys see [Stinson 2006].

These two algorithms represent a change in the basic philosophy of the factoring methods discussed so far. While trial division and Fermat's approach are both deterministic, Pollard's two factoring algorithm's are probabilistic. That is, for bad polynomial choices for the Pollard-Rho and unlucky base choices or high $p-1$ factor values for the factors of n we may have to rerun the algorithm or, worse, the process may run forever. All of the modern high power factoring methods are based on probabilistic approaches. A fast deterministic factoring algorithm simply does not exist yet. The trick is finding a way to improve our luck and take faster algorithmic approaches for large values of n.

Before beginning our discussion of the Quadratic Sieve we must discuss one last property called Quadratic Reciprocity. Earlier we discussed Euler's criterion which involved the definition of a quadratic residue. Recall that b is a quadratic residue if there exists a $t \in \mathbb{Z}$ such that $b \equiv t^2 \pmod{p}$.

We would like to find a way to tell when an integer b is a quadratic residue for a given prime p. Euler's criterion gave a deterministic algorithm by reducing the question to finding the residue of $b^{(p-1)/2} \bmod p$. The problem is such exponentiation, though fast, is relatively expensive when weighed against the important but small fact of deciding if b is a quadratic residue. A simpler algorithm should exist. To start we will introduce some notation

Let p be an odd prime and $n \in \mathbb{Z}$. The Legendre symbol (n/p) is defined to be 0 if p divides n, $+1$ if n is a quadratic residue $\mod p$, and -1 if it is not. Therefore, we can restate Euler's criterion as follows

$$n^{(p-1)/2} \equiv (n/p) \pmod{p}.$$

If we set $n = -1$ we come up with the following Lemma,

Lemma 3. *The Legendre Symbol* $(-1/p)$ *is* 1 *if* $p \equiv 1 \pmod 4$ *and is* -1 *if* $p \equiv 3 \pmod 4$

With this new definition we can also see the following properties of the Legendre Symbol.

Lemma 4. *Given* $a, b \in \mathbb{Z}$ *then,* $(a \cdot b/p) = (a/p) \cdot (b/p)$

Lemma 5. *If $a, b \in \mathbb{Z}$ and $a \equiv b \pmod{p}$ then $(a/p) = (b/p)$.*

Lemma 6. *If p does not divide $a \in \mathbb{Z}$ then $(a^2/p) = 1$*

Our next theorem to consider is due to Gauss. It is extremely important to our goal of an algorithm to very quickly decide whether an integer is a quadratic residue quickly. It is used to prove Quadratic Reciprocity which will be defined later.

Theorem 11 (Gauss' Criterion)**.** *Let p be an odd prime and $b \in \mathbb{Z}^+$ not divisible by p. For each positive odd integer $2i-1$ less than p let r_i be the residue of $b \cdot (2i-1) \bmod p$. That is*

$$r_i \equiv b \cdot (2i-1) \pmod{p} \qquad 0 < r_i < p.$$

Further let t be the number of r_i which are even. Then

$$(b/p) = (-1)^t.$$

The Legendre symbol can be proven out for small numbers like 2 so that no complex calculations need to be carried out. By investigation it can be observed that 2 is a quadratic residue mod p when $p \equiv 1$ or $-1 \pmod 8$ and is not a quadratic residue when $p \equiv 3$ or $-3 \pmod 8$. Using Theorem 11 we this fact can easily be proven. This implies the following Lemma.

Lemma 7. *If p is an odd prime then*

$$(2/p) = (-1)^{(p^2-1)/8}$$

To see that this formula holds just verify that $(p^2-1)/8$ is even if $p \equiv 1$ or $-1 \pmod 8$ and odd if $p \equiv 3$ or $-3 \pmod 8$.

We can continue proving these properties for larger and larger numbers but the Lemma statements become increasingly complex and harder and harder to prove. What is really needed is a more general property that will allow for systematic reduction and from which we can use our compilation of Lemmas to produce an efficient algorithm.

From Lemma 4 we know that computing (n/p) for an odd prime q can be reduced to finding (p_i/q) for those primes p_i that divide n if n's prime factorization is known. In order to help us with this calculation we need the following theorem due to Gauss known as Quadratic Reciprocity.

Theorem 12 (Quadratic Reciprocity)**.** *If p and q are odd primes and at least one of them is congruent to $1 \bmod 4$ then*

$$(p/q) = (q/p)$$

if both p and q are congruent to $3 \bmod 4$ then

$$(p/q) = -(p/q)$$

While this result is nice when combined with our previous theorems and Lemmas, it is not particularly useful in calculating (n/p) unless we know the prime factorization of n. If we are going to use the Legendre Symbol in our factorization algorithms for a composite n we will obviously not have its prime factorization handy. This problem was resolved by Carl Jacobi with the Jacobi Symbol.

Definition 2. *Let n be a positive integer and m be any positive odd integer such that*

$$m = p_1 \cdot p_2 \cdot \ldots \cdot p_r$$

where p_i are odd primes that can be repeated. The Jacobi Symbol (n/m) is computed

$$(n/p) = (n/p_1) \cdot (n/p_2) \cdot \ldots \cdot (n/p_r)$$

where (n/p_i) is the Legendre Sybmol

While the Jacobi symbol does not indicate whether n is a quadratic residue modulo m, it does have two very important implications. The first is that if m is prime then the Jacobi symbol is exactly the Legendre Symbol. The second important fact is that the Jacobi symbol satisfies the same computational properties as the Legendre symbol. These properties are

1. $(n/m) \cdot (n/m') = (n/(m \cdot m'))$
2. $(n/m) \cdot (n'/m) = ((n \cdot n')/m)$
3. $(n^2/m) = 1 = (n/m^2)$, given that $(n, m) = 1$
4. if $n \equiv n' \pmod{m}$, then $(n/m) = (n'/m)$
5. $(-1/m) = 1$ if $m \equiv 1 \pmod{4}$, $= -1$ if $m \equiv -1 \pmod{4}$
6. $(2/m) = 1$ if $m \equiv 1$ or $-1 \pmod{8}$, $= -1$ if $m \equiv 3$ or $-3 \pmod{8}$
7. $(n/m) = (m/n)$ if n and/or $m \equiv 1 \pmod{4}$, $= -(m/n)$ if n and $m \equiv 3 \pmod{4}$

What this implies is that aside from pulling out factors of 2 as they arise in the Legendre symbol reductions, we do not need to worry about whether the numerator is prime and can proceed with quadratic reciprocity regardless. Here is an example.

Ex 5.

$$\begin{aligned}(1003/1151) &= -(1151/1003)\\ &= -(148/1003) = -(4/1003)\cdot(37/1003)\\ &= -(37/1003) = -(1003/37)\\ &= -(4/37)\\ &= -1\end{aligned}$$

So we can now build our algorithm.

Algorithm 8 (Calculation of the Jacobi/Legendre symbol)**.**
Given n, m we return (n/m)
1. [Reduction using Lemmas]

$n = n \bmod m$;
$t = 1$;
$while(a \neq 0)\{$
  $while(a\%2 == 0)\{$
    $a = a/2$;
    $if(m \bmod 8 == 3 \ or \ == 5)\ t = -t$;
  $\}$
  $(a, m) = (m, a)$ *(variable swap)*
  $if(a \equiv m \equiv 3 \pmod 4)\ t = -t$;
  $a = a \bmod m$;
$\}$

2. [Termination Stage]
  $if(m == 1)$ ***return*** t;
  return 0;

This algorithm finds the Jacobi symbol of (n/m) and if m is an odd prime (n/m) is also the Legendre symbol [Crandall and Pomernace 2005]. The logical next question after determining if n is a quadratic residue $\bmod\ p$ is what the value of t is for $n \equiv t^2 \pmod p$. We will discuss the algorithm to calculate these square roots in $\mathbb{Z}_p$ in the next sections.

Even with our original definition of the Legendre symbol from Euler's criterion we can calculate the symbol directly from modulus arithmetic in $O(\ln^3 p)$ time complexity. Using our corllaries and Quadratic reciprocity we can achieve a bit complexity of $O(\ln^2 m)$ when $\mid n \mid < m$.

Now that we have an efficient algorithm to decide whether a number is a quadratic residue $\bmod\ p$ where p is a prime, we can proceed to the Quadratic sieve. It is one of the fastest and most powerful factoring algorithms available. The Quadratic Sieve is one of our best options when it comes to factoring

numbers of immense size where trial division, Pollard-Rho, Pollard p-1, and other mid-range factorization approaches become impractical. While the Quadratic Sieve is extremely powerful and significantly more efficient on these larger numbers than our other algorithms, it takes almost as much work to factor numbers of large magnitude as it does to split numbers of smaller magnitude. As such, the QS algorithm should be used in place of our earlier algorithms only if they have failed or the number is very large.

To start let's recall the suggestion that Kraitchik made to improve Fermat's factoring algorithm. If we could find "random" integers x and y such that

$$x^2 \equiv y^2 \pmod{n},$$

then we have a good chance that the $gcd(n, x-y)$ is a non-trivial factor of n. The Quadratic Sieve is a systematic approach to finding such x and y values relatively efficiently. Its method is built on a slightly simpler algorithm due to John Dixon.

To start we choose a random integer r and compute

$$g(r) = r^2 \bmod n.$$

Now we attempt to factor $g(r)$. Dixon's algorithm demands a lot of $g(r)$ values and therefore we do not spend very much time trying to factor each individual $g(r)$. Instead we set a limit for trial division of each computed value. Say that limit is $5,000$. If $g(r)$ does not factor under our limit then we will simply choose a different r and try again. This process is repeated until we have more completely factored $g(r)$'s than primes below our limit of trial division. In the case of $5,000$ there are 669 primes under this limit and hence we would need more than 669 r values for which we can factor the corresponding $g(r)$.

For our example let $p_1, p_2, \ldots, p_{669}$ be the first 669 primes. If $g(r)$ factors completely under our limit then each $g(r)$ can be written

$$g(r) = p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_{669}^{a_{669}}.$$

We can simplify our representation of $g(r)$ to the vector

$$v(r) = (a_1, a_2, a_3, \ldots, a_{669}).$$

If all of the entries in $v(r)$ are even then we have our answer since

$$g(r) \equiv r^2 \pmod{n},$$

and $g(r)$ is a square number. This is, however, a rare occurrence. The scarcity of this happening is the reason why we are required to have more

$g(r)$'s than primes below our trial division limit. If we have more $v(r)$ vectors than the length of any of the vectors respectively then by linear algebra we are guaranteed a sum of distinct $v(r)$'s that have all even entries. This can be achieved by creating a new vector, call it $w(r)$. Each $w(r)$ will be created by reducing each entry in every $v(r)$ mod 2. Yielding

$$\begin{aligned} w(r) &= (b_1, b_2, \ldots, b_{669}), \quad \text{such that} \\ b_i &= 1, \qquad \text{if } a_i \text{ is odd} \\ b_i &= 0, \qquad \text{if } a_i \text{ is even.} \end{aligned}$$

Performing Gaussian elimination in $\mathbb{Z}_2$ with these new vectors we can find a combination of r values for which the sum of their $v(r)$ vectors has all even elements and therefore is a perfect square. Observe that our vectors over $\mathbb{Z}_2$ are guaranteed to be linearly dependent because the matrix has more rows than columns.

Since the sum of our selected $v(r)$'s has all even components the product of the corresponding $g(r)$'s will combine to a perfect square. If we have kept track of our $g(r)$'s and r values correctly we find a congruence that represents our sought-after x and y values

$$g(r_1) \cdot g(r_2) \cdot \ldots \cdot g(r_k) \equiv r_1^2 \cdot r_2^2 \cdot \ldots \cdot r_k^2 \pmod{n}.$$

We now have the good odds, which Kraitchik recognized, that this will yield us a non-trivial factor of n. If we are unlucky and it does not then we can continue our Gaussian elimination to find another linearly dependent combination of $w(r)$'s. While this algorithm is probabilistic, in practice it will find a large factor of n much faster than any deterministic approach.

This algorithm's strategy was refined by Carl Pomerance with the addition of a sieving procedure that gives this algorithm its name. Instead of choosing our r's at random we will let

$$k = \lfloor \sqrt{n} \rfloor,$$

and take $k+1, k+2, \ldots$ for values of r. We define $f(r)$ as

$$f(r) = r^2 - n.$$

We see that $f(r) = g(r)$ if r ranges between k and $\sqrt{2n}$. Our goal is to once again find $f(r)$'s that factor completely under our trial division limit. So let p be an odd prime under $5,000$ that does not divide n. This is a reasonable assumption since before using the Quadratic Sieve we should have already

tried some small scale trial division to pull out any small factors of n that may exist. If p divides $f(r)$ then

$$p \mid r^2 - n \Rightarrow n \equiv r^2 \pmod{p}. \tag{14}$$

Therefore the Legendre symbol (n/p) must equal $+1$. So we need only consider those primes for which (n/p) is equal to $+1$ for our factoring of each $f(r)$. This set of primes is called our factor base. An $f(r)$ is known as B-smooth if all of its prime factors are less than or equal to a limit B. If n is a quadratic residue mod p then there exists a t such that

$$n \equiv t^2 \pmod{p}.$$

Therefore by equation(14)

$$r \equiv t \text{ or } -t \pmod{p}.$$

Moreover, if $r \equiv t$ or $-t \pmod{p}$ then p must divide $f(r)$ since

$$r \equiv t \text{ or } -t \pmod{p} \Rightarrow r^2 \equiv t^2 \equiv n \pmod{p} \Rightarrow p \mid r^2 - n.$$

We can now make two passes through our list of $f(r)$'s for each prime in our factor base. Once we find the first r in our list congruent to the corresponding t modulo p, we know that its associated $f(r)$ and every p^{th} $f(r)$ thereafter (since each p^{th} r afterwards also maintains the congruence) will be divisible by p. This constitutes the first pass. Then we find the first r congruent to $-t$ and make a similar second pass through our list recording prime factors into our $v(r)$ vectors as we find them. Because prime powers may also divide into our $f(r)$ list it is important to also solve the congruences

$$t^2 \equiv n \pmod{p^a},$$

where p is an odd prime in our factor base. In our implementation of the Quadratic Sieve Bressoud's estimation for how far a should range and the resulting extent of congruences solved was used. That is, these congruences should be resolved for each a running up to about

$$\frac{2 \log L}{\log p},$$

where L is the largest prime in the factor base.

The QS algorithm can be summarized as follows.

Algorithm 9 (Quadratic Sieve)**.**

1. *Build our factor base with primes such that* $(n/p) = +1$ *and solve the congruences*

$$t^2 \equiv n \pmod{p^a}, \quad 1 \leq a \leq \frac{2 \log L}{\log p}$$

2. *Perform the sieving procedure recording the* $v(r)$ *vectors for each* $f(r)$ *to find enough* $f(r)$*'s that factor completely over our factor base.*
3. *Perform Gaussian elimination in* $\mathbb{Z}_2$ *to find a combination of* $f(r)$*'s which is a perfect square.*
4. *Solve for* x *and* y *and compute* $gcd(x - y, n) = k$.
If (k *is not* 1 *or* n*)* ***return*** k
else repeat step 3

We will go into the details of each of these steps sequentially.

Building the factor base is easy since we already have a compact and efficient way of determining the Legendre symbol thanks to the theorems we have already developed. What remains somewhat unclear is how big to make our factor base. We do not want to make it too small or we will decrease the chances of our $f(r)$'s completely factoring. We must also avoid making the factor base too large so as to reduce the number of $f(r)$'s that we need to produce dependence in the Gaussian elimination step. A complete discussion of this is deferred until we discuss the actual implementation of the algorithm.

After constructing the factor base we need to solve the quadratic congruences

$$t^2 \equiv n \pmod{p}, \tag{15}$$

for each p in our factor base. To solve these congruences quickly we can use the following algorithm.

Algorithm 10. *Given an odd prime* p *and an integer* a *with* $(a/p) = 1$

1. [Check the simple cases of $p \equiv 3, 5, 7 \pmod 8$*]*
$a = a \bmod p$
$if(p \equiv 3, 7 \pmod 8)\{$
$\quad x = a^{(p-1)/4} \pmod p$
$\quad$ ***return*** x
$\}$
$if(p \equiv 5 \pmod p$ *and* $n^{2k+1} \equiv 1 \pmod p)\{$
$\quad x = n^{k+1} \pmod p$
$\quad$ ***return*** x
$\}$
$if(p \equiv 5 \pmod p$ *and* $n^{2k+1} \equiv -1 \pmod p)\{$
$\quad x = (4n)^{k+1} \cdot (\frac{p+1}{2}) \pmod p$

return x

}

2. *[Case* $p \equiv 1 \pmod 8$*]*

Find a random integer $d \in [2, p-1]$ *with* $(d/p) = -1$

$p - 1 = 2^s \cdot t$, *where* t *is odd*

$A = a^t \bmod p$

$D = d^t \bmod p$

$m = 0$

$for(0 \leq i < s)\{$

$if((AD^m)^{2^{s-1-i}} \equiv -1 \pmod p)\ m = m + 2^i$

}

$x = a^{(t+1)/2} \cdot D^{m/2} \bmod p$

return x

This algorithm is an amalgamation of two different approaches, see [Crandall and Pomernace 2005 and [Bressoud 1989]. This algorithm will deterministically and in polynomial time calculate the answer to Equation(15). In our implementation that will be discussed later this is the exact algorithm use to calculate the square roots in $\mathbb{Z}_p$ for a prime p.

Step 2 in our QS algorithm is the sieving operation. The easiest way to fully explain this operation is by a small example. Take $n = 799$. So $\lfloor \sqrt{n} \rfloor = 28$. If we want a factor base of 5 primes and 20 $f(r)$ values rather than taking r values between $[29, 49]$ it is advantageous to take r values that straddle the square root

$$18 < r < 39$$
$$r = 18 + i, \quad 1 \leq i \leq 20.$$

This straddling of r gives us better values for $f(r) = r^2 - n$ since it keeps the values closer to zero. They are, therefore, more likely to factor over our factor base. We do come up with negative $f(r)$ values with this range but we can factor out the -1 and treat -1 as the first prime in our factor base. We will also automatically insert 2 into the factor base since it is a special prime. In the sieving process all of the $f(r)$'s will be reduced to $f(r) = 2^s \cdot t$. The value of s will be recorded in the $v(r)$ and further sieving will be done on t. For our example we come up with our factor base of

$$-1 \quad 2 \quad 3 \quad 5 \quad 7.$$

For each of the odd primes in our base we solve $n \equiv t^2 \pmod p$ with Algo-

rithm 10.

$$3, t = 1$$
$$5, t = 3$$
$$7, t = 1.$$

Solving $n \equiv t^2 \pmod{p^a}$ we get

$$t^2 \equiv 799 \equiv 7 \pmod{3^2}, t = 4 \qquad t^2 \equiv 799 \equiv 16 \pmod{3^3}, t = 4$$
$$t^2 \equiv 799 \equiv 70 \pmod{3^4}, t = 31 \qquad t^2 \equiv 799 \equiv 24 \pmod{5^2}, t = 7$$
$$t^2 \equiv 799 \equiv 15 \pmod{7^2}, t = 8$$

The procedure used to find these values is called Hensel lifting and will be discussed in depth later. The last thing to do is find the first r values congruent to our t's modulo their corresponding p's. This process is simple and very quick. We must also remember that $-t$ is also valid. The r values are as follows

$$r = 19 \equiv 1 \pmod{3} \qquad r = 32 \equiv 7 \pmod{5^2}$$
$$r = 20 \equiv -1 \pmod{3} \qquad r = 18 \equiv -7 \pmod{5^2}$$
$$r = 22 \equiv 4 \pmod{3^2} \qquad r = 18 \equiv 3 \pmod{5}$$
$$r = 23 \equiv -4 \pmod{3^2} \qquad r = 22 \equiv -3 \pmod{5}$$
$$r = 31 \equiv 4 \pmod{3^3} \qquad r = 22 \equiv 1 \pmod{7}$$
$$r = 23 \equiv -4 \pmod{3^3} \qquad r = 20 \equiv -1 \pmod{7}.$$

There is no r is our range $\equiv 8$ or $-8 \pmod{7^2}$.

Now we are ready to begin sieving. We run over each $f(r)$ generated in our list, starting with the first $r \equiv t \pmod{p}$, factoring out p from every p^{th} $f(r)$. This process is done for every p in our factor base and we come up with this list of $f(r)$'s

$f(19) = 19^2 - 799 = -438 = -1 \cdot 2 \cdot 3 \cdot 73$
$f(20) = 20^2 - 799 = -399 = -1 \cdot 3 \cdot 7 \cdot 19$
$f(21) = 21^2 - 799 = -358 = -1 \cdot 2 \cdot 179$
$f(22) = 22^2 - 799 = -315 = -1 \cdot 3^2 \cdot 5 \cdot 7$
$f(23) = 23^2 - 799 = -270 = -1 \cdot 2 \cdot 3^3 \cdot 5$
$f(24) = 24^2 - 799 = -223 = -1 \cdot 223$
$f(25) = 25^2 - 799 = -174 = -1 \cdot 2 \cdot 3 \cdot 29$
$f(26) = 26^2 - 799 = -123 = -1 \cdot 3 \cdot 41$
$f(27) = 27^2 - 799 = -70 = -1 \cdot 2 \cdot 5 \cdot 7$
$f(28) = 28^2 - 799 = -15 = -1 \cdot 3 \cdot 5$
$f(29) = 29^2 - 799 = 42 = 2 \cdot 3 \cdot 7$

$f(30) = 30^2 - 799 = 101 = 101$
$f(31) = 31^2 - 799 = 162 = 2 \cdot 3^4$
$f(32) = 32^2 - 799 = 225 = 3^2 \cdot 5^2$
$f(33) = 33^2 - 799 = 290 = 2 \cdot 5 \cdot 29$
$f(34) = 34^2 - 799 = 357 = 3 \cdot 7 \cdot 17$
$f(35) = 35^2 - 799 = 426 = 2 \cdot 3 \cdot 71$
$f(36) = 36^2 - 799 = 497 = 7 \cdot 71$
$f(37) = 37^2 - 799 = 570 = 2 \cdot 3 \cdot 5 \cdot 19$
$f(38) = 38^2 - 799 = 645 = 3 \cdot 5 \cdot 43.$

It should be noted that several values (e.g. f(20) or f(21)) do not completely factor over the factor base. Our sieving process leaves us with a selection of 7 $f(r)$'s that are completely factored over our base.

The final step in our process is Gaussian elimination on our $v(r)$ vectors reduce mod 2. Our selected $f(r)$'s looks as follows

$f(22) = -1 \cdot 3^2 \cdot 5 \cdot 7$	11001
$f(23) = -1 \cdot 2 \cdot 3 \cdot 5$	01111
$f(27) = -1 \cdot 2 \cdot 5 \cdot 7$	11011
$f(28) = -1 \cdot 3 \cdot 5$	01101
$f(29) = 2 \cdot 3 \cdot 7$	10110
$f(31) = 2 \cdot 3^4$	00010
$f(32) = 3^2 \cdot 5^2$	00000.

We write our vectors so that the element in the right most column is the power for -1 and the ascending primes in our factor base are recorded thereafter starting from the right. In order to keep track of what combination of $f(r)$'s gives us the perfect square we adjoin an identity matrix to our existing matrix with dimensionality equal to the number of B-smooth $f(r)$'s. So our matrix looks like this

11001	1000000
01111	0100000
11011	0010000
01101	0001000
10110	0000100
00010	0000010
00000	0000001

In this example, we are lucky; we have an $f(r)$ with all even exponents already. If we were not so lucky we would have to perform Gaussian elimination

in $\mathbb{Z}_2$ to find a linear combination of $f(r)$'s that lead to a perfect square as follows. Starting with the first column, we locate the first vector with a 1 in that column and eliminate down the column by adding that row in $\mathbb{Z}_2$ to those vectors with a 1 in that column. Now there will be no 1's in the first column. This step is repeated for each column until we get a vector with the non-identity part zeroed out. The identity part of the vector will contain 1's in column's indicating the row from which each $f(r)$, in the perfect square product, came from in the original matrix. In our example

$$00000 \qquad 0000001,$$

there is a 1 in the 7^{th} column of the identity part of the vector indicating that it is the 7^{th} $f(r)$ in our list that gave us the perfect square. Continuing with our example we perform the final step of our algorithm.

$$\begin{aligned} 32^2 &\equiv 3^2 \cdot 5^2 \pmod{799} \\ (32^2) &\equiv (15)^2 \pmod{799} \end{aligned}$$

Performing our 4^{th} step in the QS algorithm we compute

$$\begin{gathered} 32 \bmod 799 = 32 \text{ and } 15 \bmod 799 = 15 \\ gcd(32 - 15, 799) = 17. \end{gathered}$$

17 is a non-trivial factor of 799.

Our implementation of the Quadratic Sieve algorithm was done in java. The first decision that had to be made was how to determine the size of the factor base and the length of the interval that would be used to generate our $f(r)$ values. Initially it was easiest for testing and simplicity's sake to have a user input the value for the factor base size and the number of $f(r)$'s to be generated. While this is a viable option for the algorithm it requires the user to execute mathematical estimation or trial and error. After the initial testing periods the suggested interval lengths in Crandall and Pomerance were implemented. For the Quadratic Sieve algorithm they recommend calculating a limit $B = \lceil L(n)^{1/2} \rceil$ where $L(n) = e^{\sqrt{\ln n \ln \ln n}}$ and building the factor base by taking those primes p_i less than B such that $(n/p_i) = +1$. The interval over which our r values will run is then calculated $[-M, M]$ where $M = B^2 = L(n)$. It is noted in [Crandall and Pomernace 2005] that this is an estimate of the optimal values for the factor base size and the interval length. There were cases in testing when these values were insufficient to split a given n. The good part about these estimates for B and M is that they are easily automated and adjust based on the size of our given n.

The first step for the algorithm implementation was to establish the constant values needed to begin. From the given input n the value for $r = \lfloor \sqrt{n} \rfloor$ had to be calculated. Because of the magnitude of the numbers being used the primitive int representation was not sufficient. As a result many of the number values had to be done in the BigInteger java class. As a class BigInteger does not have a square root function built in. Because of this a static function was built based on Algorithm 9.2.11 in [Crandall and Pomernace 2005] to calculate the integer part of a square root. After this calculation and storing values for B and M the algorithm proceeds to building the factor base.

The factor base is stored as a Linked List of BigIntegers. Its first two values are initialized to always be -1 and 2. The subsequent primes are pulled from a text file list of primes. The current list is one million primes long but can easily be expanded. The list of primes was generated using a simple iterative algorithm and the isProbablePrime function provided by the java BigInteger class. Starting with 3 the algorithm increases its testing value by two each pass. It then tests whether that value is a probable prime with a confidence value of 25. With a confidence value of 25 the probability of a false prime designation is $1/2^{25} = 2.980232239 \times 10^{-8}$. This kind of probability estimation suggests that the IsProbablePrime function merely calculates some version of the Miller-Rabin Test that we discussed before. With this level of confidence the risk of a false positive is almost negligible. Once a prime is pulled from the list the Legendre symbol (n/p) is calculated using Algorithm 8. If the symbol is $+1$ then it is added to the factor base. If it is not then the next prime is drawn from the list until we have a satisfactory number of primes in our factor base.

The next step in the algorithm is to find, for each p in the factor base, values of t such that

$$n \equiv t^2 \pmod{p^a}, \quad \text{for } 1 \leq a \leq \frac{2 \log L}{\log p}.$$

To store the values of t another Linked List was used but an object called a *pair* was created to hold the values. The *pair* class holds a number of values in order to do things like associate a t value to a specific p value. The variables in the *pair* class are the value p^a, the t value for that p^a, the exponent a, the base p, and the index of that p in our factor base linked list. It also holds the first r value for which $r \equiv t \pmod{p}$ and the first r value such that $r \equiv (-t) \pmod{p}$ but these will be set later. To solve for our t values each prime p in our factor base is cycled through and the maximum exponent needed to be solved is estimated using Bressoud's suggested formula. There are two cases to deal with when solving for t. The first is the simpler case, which can use

Algorithm 10 that was discussed earlier

$$n \equiv t^2 \pmod{p}.$$

The second case is

$$n \equiv t^2 \pmod{p^a}, \tag{16}$$

where a runs over $[2, \frac{2 \log L}{\log p}]$. In general, it is a very hard problem to solve quadratic congruences modulo a composite number, but in this case we don't have an ordinary composite number. It is a prime power, and we have already solved the square root for that prime. It turns out there is a relationship between these two congruences and it can be exploited using a form Hensel Lifting.

Suppose that we have an odd prime p and an A such that $A^2 = n \bmod p$ and we want to solve

$$x^2 = n \bmod p^2.$$

We are looking for a solution in the range of $0, \ldots, p^2 - 1$. So we can write x in the form $Bp + A$ where $0 \leq B < p$. Indeed, we can write x in this form since in general, if $n \equiv A^2 \pmod{p}$ and we are looking for $n \equiv x^2 \pmod{p^a}$ then the following will hold

$$\begin{aligned} n - A^2 = kp &\Rightarrow n = kp + A^2 \\ n - x^2 = rp^a &\Rightarrow n = rp^a + x^2. \end{aligned}$$

Setting them equal we have

$$\begin{aligned} A^2 - x^2 &= rp^a - kp \\ A^2 - x^2 &= p(rp^{a-1} - k) \\ A^2 &\equiv x^2 \pmod{p}. \end{aligned}$$

So we plug $x = Bp + A$ into our equation to get

$$\begin{aligned} (Bp + A)^2 &= n \bmod p^2 \\ B^2p^2 + 2ABp + A^2 &= n \bmod p^2 \\ 2ABp &= n - A^2 \bmod p^2. \end{aligned} \tag{17}$$

Since p is an odd prime, $2A$ has a multiplicative inverse modulo p^2. Thus, we can find an integer C such that $2AC = 1 \bmod p^2$. So multiply both sides by C.

$$Bp = (n - A^2)C \bmod p^2.$$

Moreover, we know by our initial calculation that $(n - A^2)$ is divisible by p. So we can write $(n - A^2) = kp$ for some k yielding

$$\begin{aligned} Bp &= kpC \bmod p^2 \\ B &= kC \bmod p. \end{aligned}$$

Finally, we take this solution for B in the range $0, \ldots, p-1$ to get our answer for x. Consider an example.

Ex 6. *We have* $6^2 = 17 \bmod 19$ *and we would like to solve* $x^2 = 17 \bmod 19^2$. *Using equation*(17) *from above we have*

$$2 \cdot 6 \cdot 19 \cdot B = -19 \bmod 361.$$

The multiplicative inverse of $12 \bmod 361$ *can be calculated very quickly using Euclid's algorithm. The inverse,* C, *is* -30.

$$\begin{aligned} 19B &= (-19)(-30) \bmod 361 \\ B &= 30 \bmod 19. \end{aligned}$$

So we take $B = 11$ *and plugging it into our equation for* x *we see that* $x = 11 \cdot 19 + 6 = 215$.

If we have a value for $A^2 = n \bmod p^2$. We can use the same procedure to find $x^2 = n \bmod p^3$. We can use our calculations from the example above.

Ex 7. *We would like to solve* $x^2 = 17 \bmod 19^3$. *So we write* $x = 361B + 215$ *(using our previous solution for A) and proceed*

$$\begin{aligned} (361B + 215)^2 &= 17 \bmod 19^3 \\ 530 \cdot 361B &= 17 - 215^2 \bmod 19^3. \end{aligned}$$

The inverse of $530 \bmod 19^3$ *is* 2213. *Proceeding*

$$\begin{aligned} 361B &= (17 - 225^2) \cdot 2213 \bmod 19^3 \\ B &= -175 \cdot 2213 \\ &= 2 \bmod 19. \end{aligned}$$

So $x = 2 \cdot 361 + 215 = 937$.

We can calculate each t value successively for equation(16) with this algorithm. Once we calculate all of the t's they are stored in the *pair* format and added to our linked list. As mentioned above these square roots modulo p must be found for every p in our factor base. This means that we have to

access each member of our factor base. This brings up a concern that must be addressed before continuing the description of our algorithm. Indexing in java for its list objects and arrays is done by simple ints. A problem will arise then if we have more primes in our factor base than can be represented by a 32-bit int scheme. There would be no way to input a proper index into the get function of java's linked list if its size exceeded the largest primitive int. It is not out of the question to believe that for very large n, say 90 digits long, we may need a factor base of extremely large magnitude. The good news is that if we use the estimation of $B = L(n)^{1/2}$, then even for 90 digit numbers the B value and therefore the number of primes less than B, is much less than the maximum integer displayable by a primitive int. Generally speaking this algorithm was programmed with a mind towards factoring a maximum of 90 digit composites. The bad news is that as the magnitude of n grows past 100 digits B begins to surpass the maximum integer representation of the java primitive int. In order to solve this problem either a custom implementation of a linked list must be written or a new language would have to be used.

The last step before we begin our sieving is to find the first r value for which $r \equiv t \pmod{p^a}$ and the first r value for which $r \equiv (-t) \pmod{p^a}$. It is in this step that we set the firstR variables in each of the *pair*s in our linked list in order to associate the first r values with the correct p^a. Finding the first r's for which these congruences hold is very fast and relatively simple. Using the first r value in our range we calculate $r \bmod p^a$ and add or subtract the appropriate amount given the t and $-t$ values to obtain the proper r values. These values are then stored inside each *pair* within the linked list of *pairs*.

Once this process is complete we can begin the sieving operation. The simplest way to implement the sieving operation was to generate our values for $f(r)$ and store them in a large column vector. Then, iteratively, we would start with every *pair* object in our list of *pairs* and make passes over the list of stored $f(r)$ values. Using the *pair* object we can access both of the first r values and the correct associated p^a value. If the r that generates the $f(r)$ value is equal to either of the first r values for p^a or either of the first r values plus a p^{th} multiple then we divide our $f(r)$ value by the stored p^a value. In this step we must be very careful because $f(r)$ values that are divisible by higher powers of p will obviously be divisible by the lower orders of p. That is, if $f(r)$ is divisible by p and p^2, for example, then if we are not careful we will sieve out p *and* p^2 from $f(r)$ but $f(r)$ is not necessarily divisible by p^3. To avoid this problem we run over our linked list of *pairs* backwards. Since the powers of each p are calculated in order they are also stored in order. Going through the list backwards guarantees that we sieve out the highest p^a values first, and if we have already sieved out a higher power from an $f(r)$ then we simply do not sieve out the lower powers.

While performing the sieving operation we simultaneously build our factor vectors. These vectors are another implemented object. They consist of a primitive int row vector whose first part is the factor base component and the second is the correct row of the identity matrix for the vector. As primes are sieved from our list each vector is updated to include the newly sieved out prime with its corresponding power a. The special cases of -1 and 2 are handled slightly differently. If a number is negative then it is made positive and the -1 position in our vector is set to 1. For 2 we simply divide out as many powers of 2 as possible and record the highest power for which the $f(r)$ is divisible in our factor vector. When an $f(r)$ value reaches 1, we know that it has completely split over our factor base. Its vector and associated r value are then added to a Gaussian matrix object. The object is simple carrying a linked list of vector objects and a list of associated r values. Each time a vector is added to the list the corresponding r value is added to the list of r values. In this way the r values occupy the same index in their list as their associated vectors do in the vector linked list. This will become important later. When we have run through each element in our list of *pairs* then we have finished sieving and have collected all of our B-smooth values of $f(r)$.

This implementation of the sieving process mimics the one shown in our example of sieving from above. While this process works well we run into problems as we increase the magnitude of n. The number of r values in the interval over which we generate our $f(r)$'s grows very quickly. Consequently, the magnitude of the number of values to be stored in our column vector gets extremely large and iterative operations, indexing of the values, and arithmetic work get increasing slower and more difficult. In order to solve this storage problem we must take a different approach to the sieve. Instead of storing each $f(r)$ value, we generate them one by one starting at the center of our $[-M, M]$ interval and slowly moving outward to the ends of the interval. Note that we begin at the center and more outward on either side because the center of the interval will contain the smallest function values. Sieving over the complete list of *pairs* is then performed using the same update procedure for the vectors. If the $f(r)$ value completely factors over our base then we add it to the Gaussian matrix. If not, we move on to generate the next value to be sieved. This storage saving technique does come with a time disadvantage. If we generate each of the $f(r)$ values one by one, then during the sieving process we must test if the r is equal to the first r value that we have stored or a p^{th} r value thereafter for each p value in our list. This testing is in lieu of just finding the first r value and then skipping down the list from there on. These calculations are very quick but for each $f(r)$ there are a lot of them and they will be repeated for each $f(r)$. In order to mitigate this volume of calculations most algorithms generate the

$f(r)$ values in chunks so as to take advantage of skipping down part of the list but also saving space.

There are two different stopping conditions suggested for the sieving process. The first is a complete sieve over the interval of r values. The upside to this method is that it will give us the maximum number of completely factorable $f(r)$'s over $[-M, M]$. The downside is, we will need to sieve and run over every r value every time. The second is a time saving approach. Instead of sieving over the entire interval of r values we instead stop the sieving process when we have found s completely factored $f(r)$'s where s is equal the size of our factor base plus one. As discussed earlier, having more $f(r)$ vectors than factors in the factor base guarantees linear dependence of our vectors during Gaussian elimination over $\mathbb{Z}_2$. Under this termination condition we are still guaranteed to have at least one perfect square for which to test the *gcd*, and we do not necessarily have to sieve over our entire interval of $f(r)$'s. These two conditions are a matter of taste and time management. In our algorithm we opted for the second technique.

The process of Gaussian elimination is handled completely by the GaussianMatrix class. The process looks very much like the one described in the QS algorithm discussion. First we check if there is already a zeroed row in our matrix like our example from above. If there is, we perform our *gcd* calculations. If not then we proceed with the elimination steps. We look through the matrix and find the first row-column pair with a 1 element and eliminate down with that row by adding (in $\mathbb{Z}_2$) it to subsequent vectors with a one in that column. The elimination row is then removed from the linked list of vectors and thrown away. We then check if we have created a zeroed row. In the event of a zeroed row then we check the *gcd*. If this calculation yields a factor then we are done. If not, then we remove the zeroed row and continue to eliminate. This process is repeated until either we run out of vectors or find a factor. When we find a zeroed row the retrieval of our corresponding r values is made simple by our ordered storage of associated r values in the other linked list. The appended identity vector will contain the indices of those original rows responsible for combining to form the perfect square value. To obtain the r values that go with these rows we need only pull the encoded indices from the identity vector and use them to extract the r values from our list. So we can now calculate our x and y values and efficiently check the $gcd(x - y, n)$. Gaussian elimination goes quickly in practice. The time complexity estimate is good but a somewhat high, about $O(n^3)$ when our matrix is $n \times n$. This time bound is actually somewhat smaller for our application of Gaussian elimination since it is done over $\mathbb{Z}_2$ and therefore lends itself well to computer calculation. However, as the numbers that we try to factor grow the matrix that we must eliminate

on grows in two dimensions. This time bound begins to become problematic and the elimination stage of the Quadratic Sieve begins to bog down. This problem will be discussed later.

Sieving is the part of the algorithm that generally takes the longest to run. One of the first Quadratic Sieve implementations was done in 1982 by Gerver at Rutgers, see [Bressoud 1989], on a 47 digit number. Solving the congruences took seven minutes. The Gaussian elimination took six minutes. However, the sieving process took about 70 hours of CPU time to complete. The sieve is where much of the optimization work can be done. There are a good many ways to improve our own implementation of the Quadratic Sieve. Indeed, if we were going to attempt to crack a 90 digit number these improvements would be imperative. If we were to go back and attempt to improve our runtime efficiency the first place to look would be the actual division and reduction of the $f(r)$ values. As it stands right now the algorithm simply performs division on each of the $f(r)$ values in order to reduce them. We know that the value is completely factorable over the factor base if we are left with a 1 where the original $f(r)$ value used to be. The problem with these calculations is that we must perform division on a very large n. Division is an inefficient algorithm in terms of computational speed. In order to avoid strict division it is possible to instead store the logarithm of $f(r)$ to either double or single precision and subtract off the logarithms of p. This is very much like division since

$$\log(\frac{x}{y}) = \log x - \log y.$$

Even though our floating point storage does not maintain exact arithmetic, and therefore does not maintain this relationship perfectly, we have a good number of decimal places in either precision. Therefore when the remaining logarithm stored for a particular $f(r)$ is sufficiently close to 0, then that $f(r)$ is completely factorable. This type of calculation is desirable because instead of division we can calculate the log quickly and then perform simple subtraction from then on.

To further build on this optimization we could also implement a suggestion by Silverman. Rather than calculating the absolute logarithm of each $f(r)$, which can be expensive as the values grow larger we start with all of the values equal to zero. Then for each $f(r)$ that p should divide we add $\log p$. If we are sieving over the interval $[-M, M]$ then the logarithm of the absolute value of $(\lfloor\sqrt{n}\rfloor - M + i) - n$, which represents our $f(r)$ calculation, will be about

$$(\log n)/2 + \log M. \tag{18}$$

When we are finished sieving we will simply look for values that are close to this value. There will be sufficiently few of these so that trial division can be done without hurting efficiency to verify that they completely factor over our factor base. For more on on this optimization modification and the quantitative estimates of how close to Equation(18)'s value that we need to get see [Bressoud 1989] and [Silverman 1987]. The Silverman modification does mean that a few r values for which $f(r)$ completely factors may be missed but the increase in speed for the sieve compensates for this loss.

In Crandall and Pomerance's discussion of the quadratic sieve. They suggest that solving the congruences for and sieving over the higher powers of the primes in our factor base might be skipped. That is, we would not solve the congruence $t^2 \equiv n \pmod{p^a}$ for those a greater than 1. The claim is that these prime powers do not contribute significantly to finding B-smooth $f(x)$ values [Crandall and Pomernace 2005]. If we were to ignore the calculation of such prime powers and completely forgo sieving over these numbers then we could in practice speed up our sieve even more.

At the end of our implementation example we examined very briefly the run-time estimate of the Gaussian elimination step. As mentioned earlier we can see a few problems developing with the Gaussian matrices that we will be working with if the magnitude of n begins to grow very large. The first problem is that the matrix that we will be eliminating on will also begin to grow very large. In practice we will be storing a factor matrix that has dimension $A + 1 \times A$ where A is the factor base size. This kind of matrix demands a significant amount of storage. Therefore, as the size of the matrix gets large so does the awkwardness of systematically manipulating its elements. The initial factor matrix will be very sparse, containing only a small number of ones while the rest is filled by zeros. To save space we can keep track of the indices of the ones which implicitly indicates the location of the zeroes. If there are a relatively small number of ones then through this sparse matrix storage we significantly reduce the amount of information needed to describe our matrix. As the elimination process begins we merely throw out the indices of ones that are flipped to zeros and add the coordinates of those zeros that are flipped to ones.

Because of the sparse nature of the matrix there will be very few operations carried out in the beginning. This is a huge calculation complexity advantage. However, as Gaussian elimination over $\mathbb{Z}_2$ progresses, the matrix begins to lose its sparse nature. Because sparse matrix calculations are so quick it is advantageous to keep the matrix sparse for as long as possible. Several techniques for Gaussian elimination have been discovered that can be employed to help preserve matrix sparseness for as long as possible. Such methods, often referred to as structured Gaussian methods, can be found in

[Odlyzko 1985]. The methods described in Odlyzko's paper are founded on a simple observation used in work with sparse matrices. That is, that if a matrix is sparser on one end than the other, then it is better to start Gaussian elimination from the more sparse end. Odlyzko suggests arranging the columns of the matrix so that the columns with the least spareness are on the left and the most are on the right. The algorithm to perform the sparse matrix Gaussian elimination is detail in the paper and attempts to reduce the complexity of solving the system. Other algorithms can also be found in [Pomerance and Smith 1992].

While these methods speed up Gaussian elimination, alternative sparse-matrix algorithms have been suggested that can often efficiently solve linear systems. These methods are intended to replace Gaussian elimination. Such methods are the Conjugate Gradient method and Lanczos method, both of which must be adapted to suit this specific problem.

The point of this discussion is that our implementation simply stores the vector values in a large matrix form and so as n grows large so will the space and time-complexity of our Gaussian elimination step. As n begins to expand, our matrix the elimination step, without optimization, begins to overtake the sieving stage in terms of computation time. In order to avoid this we would certainly need to attempt an implementation of one of these approaches.

During testing our Quadratic Sieve algorithm was never pushed the algorithm to factor a really sizable composite. However, we were able to factor a few numbers on the order of 35 digits around 3 hours. An example of one of the larger numbers we were able to split was

$$\begin{aligned} &15639966601613347038730050396273177 7 \\ &= 288691785595328641 \times 541753086924909697. \end{aligned}$$

This integer is quite large but falls well short of the magnitude of primes used for cryptographic keys. Due to time constraints pushing the Quadratic Sieve to its limit was not undertaken but given more time and more optimization it is feasible that we could have attempted much larger numbers than this and eventually split them.

The Multiple Polynomial Quadratic Sieve is an attempt to build on the basic principles of the Quadratic Sieve in order to speed up the factoring process on even larger n values. In the QS method our generating x values run over integers within an $[-M + \sqrt{n}, M + \sqrt{n}]$ interval in order to search for B-smooth $f(x)$. As stated earlier, the x values are taken to straddle $\lfloor \sqrt{n} \rfloor$ in order to minimize the size of our function values. This is because smaller integers are more likely to be completely factorable over our factor

base. The problem is that as x gets farther away from $\lfloor\sqrt{n}\rfloor$ the values for $f(x)$ begin to grow rapidly. Therefore as our values of x get larger and larger we will begin to see a steady decline in the number of completely factorable $f(x) = x^2 - n$. In the QS algorithm our sieving performance is hurt by the rapid increase of $f(x)$ values along our interval.

The multiple polynomial variant of the QS algorithm addresses this particular problem by considering a new family of polynomials. The following idea is due to Montgomery. His idea is to intelligently construct a new polynomial for $f(x)$ that does not grow too rapidly as x moves towards the ends of our interval.

Suppose we have integer coefficients a, b, c with $b^2 - ac = n$. Now consider the polynomial $f(x) = ax^2 + 2bx + c$. Then we know that

$$af(x) = a^2x^2 + 2abx + ac = (ax + b)^2 - n, \tag{19}$$

since

$$(ax + b)^2 - n = a^2x^2 + 2abx + b^2 - n = a^2x^2 + 2abx + b^2 - b^2 - ac.$$

This implies that

$$(ax + b)^2 \equiv af(x) \pmod{n}.$$

Now we notice that if we have a value of a that is a square times a B-smooth number and an x for which $f(x)$ is B-smooth then the exponent vector for $f(x)$, once it is reduced modulo 2, will give us a row in our Gaussian elimination matrix. Note that it must be a square times a completely factorable number so that $af(x)$ will still be a perfect square when Gaussian elimination is finished. If, for example, a were not a square times a B-smooth number we would end up with a vector which should be a perfect square after elimination but is thrown off because the part of a not included in the Gaussian reduction is not necessarily square.

Moreover, the odd primes that can divide our $f(x)$ and do not divide n are those primes such that $(p/n) = 1$. Indeed, if p divides $f(x)$ then

$$p \mid (ax + b)^2 - n \Rightarrow (ax + b)^2 \equiv n \pmod{p}.$$

So n is a quadratic residue mod p. This is very good since it means that our set of p's in our factor base does not depend on the polynomial we use but only on the n we are trying to split.

We are using coefficients a, b, and c with the restriction that $b^2 - ac = n$ and a being a square times a B-smooth number but our main motivation for redefining $f(x)$ is to minimize the overall size of $f(x)$'s values. As such we will want further conditions on a, b, and c so that over our sieving interval

we may achieve small $f(x)$ integers. Our conditions on a, b, and c will clearly depend on our x value interval length. So let the interval length be $2M$. By Equation(19) we can also decide that we will take b so that it satisfies $\mid b \mid \leq 1/2a$. So now we know that we will only be considering values of $x \in [-M, M]$. It should be noted that this interval is no longer centered around $\lfloor \sqrt{n} \rfloor$ but rather zero. It is easy to see that the largest absolute value for $f(x)$ will be reached at the interval endpoints. At these points

$$\begin{aligned} af(x) = (a(M) + b)^2 - n \approx a^2M^2 - n \quad \text{so,} \\ f(x) = (a^2M^2 - n)/a. \end{aligned}$$

The least value for $f(x)$ is at $x = 0$ which yields

$$\begin{aligned} af(x) = (a(0) + b)^2 - n \approx -n \quad \text{so,} \\ f(x) = -n/a. \end{aligned}$$

So set the absolute values of these two estimates approximately equal in order to calculate an approximate a. This gives us the equation

$$a^2M^2 - n \approx -n \Rightarrow a^2M^2 \approx 2n \Rightarrow a \approx \sqrt{2n}/M.$$

If a satisfies our approximation then the absolute value of $f(x)$ is approximately bounded by $(M/\sqrt{2n})$ since

$$\begin{aligned} (a^2M^2 - n)/a &= ((\sqrt{2n}/M)^2M^2 - n)/\sqrt{2n}/M \\ &= ((2n/M^2)M^2 - n)/\sqrt{2n}/M \\ &= (2n - n)/\sqrt{2n}/M \\ &= nM/\sqrt{2n}. \end{aligned}$$

We can compare this bounding to the original QS polynomial bounding. If we use the interval $[\sqrt{n} - M, \sqrt{n} + M]$ and $f(x) = x^2 - n$, then we find an approximate bounding of

$$(\sqrt{n} + M)^2 - n = n + 2\sqrt{n}M + M^2 - n = 2\sqrt{n}M + M^2 \approx 2\sqrt{n}M.$$

where M is much smaller in magnitude than n. Just with our new polynomial we have saved a factor of $2\sqrt{2}$ in the size of our $f(x)$ values over our previous polynomial. But with our new ability to change polynomials we can also run over a much shorter x interval.

In the basic QS method, as the values of $f(x)$ continue to grow we have no choice but to continue to compute $f(x)$ until the end of the interval even as the $f(x)$'s that completely factor over our base become scarcer and

scarcer. In our new approach we can change polynomials while still keeping our previously discovered B-smooth $f(x)$ values. We can keep our $f(x)$'s and their associated prime factorization vectors because our factor base does not depend on our polynomial. Thus the B-smooth numbers we discovered for previous polynomials will still be completely factorable after we switch. In this way, as we switch polynomials we will gather more and more B-smooth values until we have enough to guarantee dependence. Since our bounds on the absolute value of $f(x)$ for our MPQS polynomials are smaller than the normal QS polynomial and we can change our generating polynomial, we can attempt to sieve on a much smaller M interval than the one we used for the basic Quadratic Sieve. We can choose $M = B = L(n)^{1/2}$, see [Crandall and Pomernace 2005]. This is smaller by a factor of B than the estimate discussed earlier for the Quadratic Sieve which was $M = B^2 = L(n)$.

The time savings for this algorithm are all very good but we still have not prescribed a way to find the b and c coefficients to match a. If we can solve $b^2 \equiv n \pmod{a}$ for b then we can find our b such that $\mid b \mid \leq a/2$. If b is defined as such then we know that

$$\begin{aligned} b^2 - n &= ak \\ b^2 - ak &= n \\ k = \frac{n - b^2}{-a} &= \frac{b^2 - n}{a}. \end{aligned}$$

So set $c = k$.

If a is odd, we know the prime factorization of a, and for each p that divides a, we have $(n/p) = 1$ then we will be able to solve b efficiently using the Chinese Remainder Theorem and Algorithm 10.

One way to effectively create our coefficients is to take a prime $p \approx (2n)^{1/4}/M^{1/2}$, with $(n/p) = 1$, and make $a = p^2$. Then a meets all of our criteria outlined before.

1. a is a square times a B-smooth number.
2. We have $a = p^2 \approx ((2n)^{1/4}/M^{1/2})^2 = (\sqrt{2n}/M)$.
3. We can efficiently solve for $b^2 \equiv n \pmod{a}$ to obtain b.

The congruence $b^2 \equiv n \pmod{a}$ will have two solutions if $a = p^2$. However, the two solutions will generate equivalent polynomials so we will use only one, $0 < b < 1/2a$.

While we have see that we gain speed through changing polynomials, there is a problem with changing them too quickly. If we have $2M = B$ and

n is in the range of 50 to 150 digits, choices for B range from about 10^4 to 10^7. Sieving is very fast over intervals of this length. The sieve operation is so fast in fact, that the computational overhead created by having to switch the generating polynomial would become a burden to our overall efficiency. This overhead is due, in most part, to what is known as the initialization problem. Given a, b, c we must determine if $p \mid f(x)$. More precisely we must solve

$$ax^2 + 2bx + c \equiv 0 \pmod{p},$$

for each p in our factor base. Then we need to determine the roots, $r(p) \bmod p$ and $s(p) \bmod p$, to this congruence relation. These roots are analogous to those first $r \equiv t$ or $-t \pmod{p}$ in the QS algorithm. So we solve

$$t(p)^2 \equiv n \pmod{p},$$

assuming that p does not divide $a \cdot n$. More clearly, we assume that p does not divide a nor does it divide n. If $(ax + b)^2 \equiv n \pmod{p}$

$$(ax + b)^2 - n = pk \Rightarrow p \mid f(x).$$

But if $t(p)^2 \equiv n \pmod{p}$ then

$$(ax + b) \equiv t(p) \text{ and } (-t(p)) \pmod{p}.$$

So we can solve for x in either case calling the two solutions $r(p)$ and $s(p)$

$$\begin{aligned} ax + b &\equiv t(p) \pmod{p} \\ ax &\equiv -b + t(p) \pmod{p} \\ x &\equiv (-b + t(p))a^{-1} \pmod{p}. \end{aligned}$$

But we want the first x so

$$r(p) = (-b + t(p))a^{-1} \bmod p. \tag{20}$$

$s(p)$ is derived similarly giving

$$s(p) = (-b - t(p))a^{-1} \bmod p. \tag{21}$$

Since $t(p)$ does not depend on our polynomial we can use the same residue values for $t(p)$ each time we switch and recalculate $r(p)$ and $s(p)$. So the main work in our computation is calculating $a^{-1} \bmod p$ for each p_i and the two mod p multiplications. While the inverse calculation can be done quickly using Euclid's Algorithm if there are a lot of primes in our factor base it is enough work that we should avoid doing it too often if we can.

Self Initialization is an approach that can help minimize the number of times these calculations must be repeated. The basic idea is to find several polynomials for the same value of a. For every a value we calculate the corresponding coefficient b by solving $b^2 \equiv n \pmod{a}$ and choose $0 < b < a/2$. We can say that the number of choices that we will have for such a b with a given value of a is 2^{k-1} where k is the number of distinct prime factors of a. In our initial creation of $a = p^2$ there was only one distinct prime factor, p, for a. So the number of choices for b were

$$2^{k-1} = 2^{1-1} = 1,$$

which is what we had. This equation for the number of b values is guaranteed by the Chinese Remainder Theorem that was discussed and proved earlier. The Chinese Remainder Theorem states that if we have an $M = m_1 \cdot m_2 \cdot \ldots \cdot m_r$ then, the system comprised of r relations and the inequality

$$b \equiv n_i \pmod{m_i} \quad 0 \leq b < M,$$

has a unique solution modulo M. Applying this theorem we know that there are k distinct p comprising a and

$$b^2 \equiv n \pmod{p_i},$$

has two solutions for b. Thus we come up with 2^k residue systems with unique solutions. Only half of these solutions will be less than $a/2$ so we have a total of 2^{k-1} viable solutions to $b^2 \equiv n \pmod{a}$. Here is a very small example.

Ex 8. *Say we want to solve* $b^2 \equiv 4 \pmod{15}$*, then we break 15 into its distinct prime factors and set up those congruences.*

$$b^2 \equiv 1 \pmod{3} \Rightarrow b = 1, 2$$
$$b^2 \equiv 4 \pmod{5} \Rightarrow b = 2, 3$$

Now we set up the systems and solve them

$b \equiv 1 \pmod{3}$ $b \equiv 2 \pmod{5}$	$b \equiv 1 \pmod{3}$ $b \equiv 3 \pmod{5}$	$b \equiv 2 \pmod{3}$ $b \equiv 2 \pmod{5}$	$b \equiv 2 \pmod{3}$ $b \equiv 3 \pmod{5}$
7	13	2	8

We then choose 7 *and* 2 *since they are less than* $1/2(15) = 7.5$

Calculation of the solutions to these systems goes extremely fast because the solutions are merely the formula used to prove the Chinese Remainder Theorem.

Now suppose that we choose a to be composed of the product of 10 different primes p. Then we will have $512 = 2^9$ choice for the corresponding b. Each of these b generates a new and distinct polynomial. Hence we need only calculate $a^{-1} \bmod p$ once and it can be used in the initialization calculation problems for each of the 512 polynomials. Further, if we generate a from 10 primes in our factor base we can just include them in our factorization vectors and we need not have a be a square times a B-smooth number.

There are two problems that arise from this discussion of Self Initialization. If we create a from primes in our factor base we must effectively take them out of our sieving step. As we saw earlier when we solve for the roots of $\mathbb{F}_p[x]$ in Equations(20) and (21) we must assume that p does not divide an. If $p \mid a$ then our calculations of $r(p)$ and $s(p)$ will fail since

$$r(p) = (-b + t(p))a^{-1} \bmod p,$$

but $a^{-1} \bmod p = 0$ since $a \equiv 0 \pmod{p}$. We see then that $r(p)$ will equal zero. Clearly this is a problem when performing our sieving. In our java implementations of the MPQS we merely drop those p_i that compose a from our list and update the factor vectors accordingly.

The second problem that arises in the Self Initialization problem is how to effectively generate our a values so that they satisfy $a \approx \sqrt{2n}/M$ and are a product of k unique primes from our factor base. In our algorithm the approach taken is to find a center prime $p \approx (\sqrt{2n}/M)^{1/k}$. Our search through our factor base is done with a basic binary search since our list of primes is ordered. A more efficient search should be implemented when considering optimization. Once we have found our central prime p we more away from p sequentially, oscillating between smaller and greater primes than p. Once we have picked up k values, including p, we have our approximate a value. The idea is that the product of k primes approximately equal to $(\sqrt{2n}/M)^{1/k}$ will be approximately equal to $\sqrt{2n}/M$.

Once again due to time constraint we were unable to push our implementation of the MPQS algorithm to its highest capacity. But we were able to factor slightly larger numbers than the Quadratic Sieve in around the same time. The largest value factored was about 43 digits taking around 6 hours to complete. This time bound is not what we had hoped for but there is still significant optimization that could be done on the code especially in the way of sparse matrix optimization and sieving calculations. Often there were times that we had to modify or completely ignored the B limit calculations and the interval size calculations suggested in [Crandall and Pomernace 2005] as the polynomials failed to produce B-smooth $f(x)$ values.

The final and possibly best feature of the MPQS algorithm is that it lends itself perfectly to distributed computing. Using a central computer to

generate the polynomial coefficients it is extremely easy to farm out those coefficients to subordinate computers to run the Quadratic Sieve up to the elimination step. Instead of the subordinate computers performing Gaussian elimination on the completely factorable $f(x)$ values that are found they simply collect them and send back the B-smooth $f(x)$'s generated by each subordinates unique polynomial. Once it sends back its information it immediately receives a new set of coefficients with which to generate a completely new set of $f(x)$'s to sieve on. Once the central computer has been given enough $f(x)$ values to guarantee dependence in the Gaussian elimination step it will perform the elimination and test the resulting gcd's to see if we have found a non-trivial divisor of n. If we have not then the central computer continues to receive and process the incoming $f(x)$ values from the subordinate machines. If we have found one then it stops all of the clustered machine processes and reports a successful factorization.

Originally our java implementation of the MPQS was going to be generalized so that it could be used as a distributed computing algorithm and tested on large scale numbers. Due to time constraints this implementation was never finished. However, it is in distributed computing that the MPQS algorithm really shows what it can do. Our advantage of shorter sieving intervals is multiplied by the fact that using distributed processing power we can attempt to generate B-smooth $f(x)$ values for many different polynomials. This fact greatly decreases the time it takes to accumulate our required number of $f(x)$ values that completely factor over our base and multiplies the computing power of the algorithm past many of the fastest machines available.

Quantum computing presents an interesting twist in large integer factorization. While no polynomial-time factorization algorithm is thought to exist for conventional computing there is already an efficient factorization algorithm posited for quantum calculations. This algorithm was introduced by Peter Shor and is known as Shor's algorithm. In theory, on a quantum computer, Shor's algorithm is thought to be able to demonstrate polynomial-time factorization factor on an integer n. While quantum computing is still in its infancy this algorithm challenges many of the boundaries set for conventional computers. As developments continue in quantum computing Shor's algorithm's potential is slowly being recognized. In 2001, IBM Scientists were able to successfully use Shor's algorithm, using quantum computation, to factor 15 into 3 and 5. While this number is small the result represented the most complicated quantum calculation at that time. For more information on Shor's algorithm see [Shor 1994].

Primality and Factorization are invariably tied together. If a number can be factored it is clearly not prime. If we can prove that a number is prime

then we clearly cannot factor it. We use the properties of primes to split large numbers with the Quadratic and Multiple Polynomial Quadratic sieves and often we naively use trial division and factorization to prove that a number is prime or composite. While Primality and Factorization are mathematically interwoven their computational complexities are extremely different. As we have seen the development of primality tests has progressed through the years and in 2002 the first definitive and fully proven polynomial-time primality test, the Agrawal-Kayal-Saxena algorithm, was discovered. While primality testing has been proven to be in P, it is believed that there is no polynomial-timed algorithm, probabilistic or deterministic, to factor a given composite n. While the development of factorization algorithms has progressed significantly it still remains an extremely difficult problem. For large composite n the time to factor it is still measured in days. While there have been new innovations in factorization like the Number Field Sieve, the Quadratic Sieve and Multiple Polynomial Quadratic Sieves still represent some of the most powerful approaches to large integer factorization. The challenge of fast and efficient factorization algorithm has, thus far, stood beyond the reach of Computer Science and Numerical Analysis faster factorization algorithms are still being produced but still we do not expect to find really efficient methods.

References

[Agrawal et al. 2004] M. Agrawal, N. Kayal, and N. Saxena, "PRIMES is in P", *Annals of Mathematics*, **160** (2004), pages 781-793.

[Bressoud 1989] David M. Bressoud, "Factorization and Primality Testing", New York, Springer-Verlag, 1989.

[Crandall and Pomernace 2005] Richard Crandall and Carl Pomerance, "Prime Numbers A Computational Perspective", New York, Springer, 2005.

[Diffie and Hellman 1976] W. Diffie and M. Hellman, "New Directions in Cryptography", 1976

[Kleinjung Franke Boehm and Bahr 2005] T. Kleinjung, J. Franke, F. Boehm, and F. Bahr. "Crypto-World" 9 May 2005, Accessed 29 April 2009, http://www.crypto-world.com/announcements/rsa200.txt.

[Koblitz 1994] Neal Koblitz, "A Course In Number Theory and Cryptographiy", New York, Springer-Verlag, 1994.

[Odlyzko 1985] A. Odlyzko, "Discrete logarithms in finite fields and their cyptographic significance", In *Advances in Cryptology, Proceedings of Eurocrypt 84, a Workshop on the Theory and Application of Cryptographic Techniques*, pages 224-314, Springer-Verlag, 1985.

[Pomerance and Smith 1992] Carl Pomerance and J. Smith, "Reduction of Huge, Sparse Matrices over Finite Fields Via Created Catastrophes", In *Experimental Mathematics*, **1** (1992), pages 89-94.

[Rivest Shamir and Adleman 1978] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *Communications of the ACM*, **21** (1978), pages 120-126.

[Saracino 1992] Dan Saracino, "Abstract Algebra A First Course", New York, Waveland Press, 1992.

[Shor 1994] Peter Shor, "Polynomial-time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer, in *SIAM Journal of Computing*, **26** (1997), pages 1484-1509.

[Silverman 1987] Robert D. Silverman, "The Multiple Polynomial Quadratic Sieve", in *Mathematics of Computation*, **48** (1987), pages 329-340.

[Stinson 2006] Douglas R Stinson, "Cryptography Theory and Practice", Boca Raton, Chapman & Hall/CRC, 2006.

[Teitelbaum 1998] J. Teitelbaum, "Euclid's Algorithm and the Lanczos Method over Finite Fields", In *Mathematics of Computation*, **67** (1998), pages 1665-1678.