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.

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

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

The symbol $\varphi(n)$ 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

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 $\varphi(n$

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

$$\lim_{x \to \infty} \quad \pi(x)$$

such as $341 = 11 \cdot$

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

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

□

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 muETB.955211hol-280(from-316(to)ea49(o)-326(to)e)-2 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

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

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

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

Proof. ( )

If there exists a $t \quad \mathrm{Z}^+$ 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

$$\begin{aligned} p \mid t \quad & pk = t \\ & (pk)^2 = t^2 \\ & p^2k^2 = t^2 \\ & p(pk^2) = t^2 \\ & p \mid t^2 \\ & p \; (\end{aligned}$$

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 $a, b, c, \ldots \quad \mathrm{Z}^{+}$

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 $b^m \equiv -1$ (mod 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

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 $\varphi(n)$ iterations to pass through each base if n is prime ($\varphi(p) = p - 1$ 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 $\varphi(n)$ 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

This theorem suggests that at least $\frac{3}{4}$ of all integers $[1, n-1]$ 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 $[1, n-1]$. To see the proof

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 $\frac{1}{4}\varphi(n) + 1$ 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 $\in [2, \sqrt{\varphi(r)} \lg n]$

                    return composite

3. For$(i \leq a \leq \sqrt{\varphi(r)} \lg n)$
   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 $g(x)^n \equiv g(x^n) \pmod{n}$*

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 is in G by closure under multiplication.

Further since $p >$ □$\overline{\varphi(r)} \lg n$ these polynomials are distinct and and non-zero in $Z_p[x]$. Indeed, they are all non-zero because p is prime and so $Z_p[x]$ is a principal ideal domain. That is, there are no zero divisors $(a, b \quad Z_p[x]$ with $a = 0, b = 0 \quad a \cdot b = 0)$. Since Z_p is a principal ideal domain each of its elements factors uniquely into irreducible elements. So we will only get repeatedonlywbfa, b

Let K K

where $(h(x$

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

polynomial. Therefore, $j_1 = j_2$ which implies

$$\begin{aligned} p^{a_1}(n/p)^{b_1} &= p^{a}_{2}(n/p)^{b_2} \\ p^{a_1}n^{b_1}p^{-b_1} &= p^{a_1}n^{b_2}p^{-b_2} \\ n^{b_1-b_2} &= p^{a_2-a_1-b_2+b_1}. \end{aligned}$$

Since our pairs (a_1, b_1) and (a_2, b_2) are distinct we know that $b_1 = b_2$. 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 $b_1 = b_2$. We have, then, n expressed in terms of p. By unique factorization in 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 $[2, \sqrt{\varphi(r)}\lg n]$. 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 $(x + a)^n \equiv x^n + a$ (mod $f($

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 $\sqrt{\varphi(r)} \lg n$ 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.

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.

miniscule chance of a false negative in proving compositeness.

3 Factorization

While primality testing is important to the creation of RSA keys, factoriza-

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 $j = i$. So we have that

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

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

$$a \bmod m_1 = a_1 \cdot M_1$$

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 $x = \sqrt{n}$ 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 $\sqrt{n}$. 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 1

prime

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 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 Z_n.

<u>Ex</u> 3. $x_0 = 3$ $\quad f(x) = x^2 + 1$ $\quad n = 799$

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

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 $x_i \equiv f(x_{i-1}) \pmod{n}$ [redacted] 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 $y_i \equiv f($

d

(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 $gcd(n, x_i - x_j$

Ex 4. $x_0 = 3$ $f(x) = x^2$

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 $p -$

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 Slgorithm,i abr

$p - 1$ 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

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

Lemma 6. *If p does not divide a 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 $\mathbb{Z}^+$ 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)ar.T11.95520011.9552102.884548955208ETBT1103(red011.9552272.0939

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

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]

```
$n = n \bmod m$;
$t = 1$;
$while(a = 0)\{$
    $while(a\%2 == 0)\{$
        $a = a/2$;
        $if(m$
```

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(mod

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

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

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)

$$r \equiv t$$

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

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

2. Perform the sieving procedure recording the $v(r)$ *vectors for each* $f(r)$ *to find enough* $f(r$

```
        return $x$
    }
2. [Case $p \equiv 1 \pmod 8$]
    Find a random integer $d$   $[2, p-1]$ with $(d/p) = -1$
    $p - 1 = 2^s \cdot t$, where $t$ is odd
    $A = a^t \bmod p$
    $D = d^t \bmod p$
    $m = 0$
    $for(0 \le i < s)\{$
        $if((AD^m)^{2^{s-1-i}} \equiv -1 \pmod p)\ m = m + 2^i$
    }
    $x = a^{(t+1)/2} \cdot D^{m/2} \bmod p$
    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 rithm 10.

$$\begin{aligned} &3, t = 1 \\ &5, t = 3 \\ &7, t = 1. \end{aligned}$$

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

$$\begin{array}{ll} t^2 \equiv 799 \equiv 7 \pmod{3^2}, t = 4 & t^2 \equiv 799 \equiv 16 \pmod{3^3}, t = 4 \\ t^2 \equiv 799 \equiv 70 \pmod{3^4}, t = 31 & t^2 \equiv 799 \equiv 24 \pmod{5^2}, t = 7 \\ t^2 \equiv 799 \equiv 15 \pmod{7^2}, t = 8 & \end{array}$$

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

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

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 $r \equiv t$ (mod p

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

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

$$
\begin{aligned}
f(22) &= -1 \cdot 3^2 \cdot 5 \cdot 7 \qquad\qquad 11001\\
f(23) &= - \qquad\quad =
\end{aligned}
$$

in 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 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.

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

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

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

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

The first step for the algorithm implementation was to establish the constant values needed to begin. From the given input n the value for $r = \sqrt{n}$ 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

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 mod 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

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 $B = L(n)^{1/2}$

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 $-$

$f(r)$ 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 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 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 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

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

[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 base. The problem is that as x gets farther away from $\sqrt{n}$ the values for $f(x$

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 $| b | \leq 1/2a$. So now we know that we will only be considering values of $x \quad [-M, M]$. It should be noted that this interval is no longer centered around $\sqrt{n}$ 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

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

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

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

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

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

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

$$(a^2M^2 -$$

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

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 e□ciency. 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 $(ax + b)^2 \equiv n \pmod p$

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

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

$$r(p) = (-b + t(p))a^{-}$$

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 $\mathrm{F}_p[x]$ 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 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.

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 aDien10e 10e1ren277.3e

[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.