Math 404

Quadratic Sieve Implementation

Harshavardhan Srijay
Edvards Sneps
Abenezer A Gelaw

April 15, 2022

1 Introduction

The quadratic sieve is a popular algorithm for factoring large composite numbers into its unique set of prime factors. It is particularly notable considering the reliance of major public-key cryptosystems like RSA on the difficulty of efficiently factoring large numbers into their unique prime factors. The problem of factoring large numbers is an example of an NP-hard problem, meaning that it can be verified quickly, but solved in polynomial time in a non-deterministic Turing machine. No algorithms to date have been able to factor a b-bit number in $O(b^k)$ time, (strongly polynomial time), for some constant k.

Here, we will discuss in detail the quadratic sieve algorithm, a general-purpose factoring algorithm to factor large n, and discuss some optimizations that we implemented to the naive quadratic sieve algorithm, in order to improve the efficiency of our algorithm at factoring large numbers.

2 Fundamental Theorem of Arithmetic

To motivate our discussion of the quadratic sieve, we will first discuss the Fundamental Theorem of Arithmetic and its implications, as this is fundamental to any discussion of factoring large numbers. The proof for this theorem is relatively straightforward, and instrumental to our discussion of factoring algorithms.

First, we will show that every integer must be either a prime or a product of primes (the existence of a unique prime factorization). We have that 2 is prime (so a product of primes). Furthermore, let all numbers $i : 1 \leq i < n$, be prime or a product of primes, for some n. If n is composite, then $n = ab$, where $1 \leq a < n$ and $1 \leq b < n$. However, by our assumption, a and b must be product of primes, which means n is also a product of prime. Thus, by induction, every integer is either a prime or a product of primes.

Now, to prove uniqueness of the prime factors of an integer, we will assume to the contrary that a number has two distinct prime factorizations. Let the least such number be denoted as n. This means $n = p_1p_2...p_i = q_1q_2...q_j$. This means p_1 divides $q_1q_2...q_j$, so p_1 divides q_j for some $j \in \mathbb{Z}$. Since p_1 and q_j are both prime, $p_1 = q_j$, so $p_2...p_i = q_k$, where $k \neq j$. However, this contradicts the assumption that n is the smallest integer with 2 unique prime factorizations. Thus, every integer can be expressed as a unique product of primes. Now, given that every integer has a unique prime factorization, we will discuss some naive algorithms to find such a prime factorization for some large $n \in \mathbb{Z}$.

3 Factoring n Using Trial Division

The most trivial and naive implementation to factor a large number n is to consider every prime number up to and including the ceiling of the square root of n. For instance, to find the prime factorization of 51, one can check every number $i \leq \lceil \sqrt{n} \rceil = 8 : 2, 3, 5, 7$. A simple trial-and-error calculation checking yields $51 = 3 * 21 = 3^2 * 7$. However, this calculation requires $O(\sqrt{n})$ time, which for sufficiently large n, is practically infeasible.

One obvious optimization to this algorithm is to only consider the primes $p \leq \lceil \sqrt{n} \rceil = 8 : 2, 3, 5, 7$, as if a composite number divides n, then each of the prime factors of that composite number must also divide n. According to the prime number theorem, the number of primes less than n is asymptotic to $\frac{x}{\log x}$. So, by doing this, we reduce the number of tests from $\sqrt{n}$ to $2\frac{\sqrt{n}}{\log n}$. Furthermore, rather than run one computation to calculate all p, and then run another computation to find the p that are prime, the algorithm can check for primality at the same time as checking for trial division. One even further optimization could be to find primes less than or equal to the ceiling of the square root of n , by starting at 3, then go up by 2 at each iteration, and check for primality at each iteration (since even numbers cannot be prime). Despite these simple optimizations however, the runtime of this algorithm is still infeasible for large n. A more efficient sieving method is needed.

4 The Quadratic Sieve

In the quadratic sieve, the basic idea is to factor n by first setting up a congruence of squares modulo n, after which one can relatively easily find the prime factors of n. In order to set up such a congruence, the algorithm calls to iteratively sieve quadratic polynomials, and use Gaussian elimination on the information gathered to find a congruence of squares modulo n.

4.1 Congruence of Squares

In general, a congruence of squares modulo n can be expressed as the following:

$$x^2 \equiv y^2 \pmod{n} \text{ where } x \not\equiv \pm y \pmod{n}$$

This relationship is useful to factor a number, due to the following:

$$x^2 \equiv y^2 \pmod{n} \Rightarrow x^2 - y^2 \equiv 0 \pmod{n} \Rightarrow (x+y)(x-y) \equiv 0 \pmod{n}$$

This means n divides $(x+y)(x-y)$, but not either of these factors alone, meaning $(x+y)$ and $(x-y)$ each are multiples of factors of n. Thus, to find the factors of n, one can calculate $gcd(x-y,n)$, and $gcd(x+y,n)$, using the Euclidean algorithm. This is the general idea of Fermat's Factorization Method. There are several ways to generate such congruences, like the continued fraction algorithm, or the number field sieve.

4.2 Finding Congruence of Squares

In general however, a simple way to generate such congruence of squares modulo n is to take the first perfect square x above n ($x = \lceil \sqrt{n} \rceil$), and calculate $x^2 - n$, and see if this is a perfect square (y^2). If it is, then we have that $x^2 \equiv y^2 \pmod{n}$. If it is not, then try $x = \lceil \sqrt{n} \rceil + 1$, and check for a congruence of squares. If not, check $x = \lceil \sqrt{n} \rceil + 2$, etc. We also have that a number is a perfect square if and only if its prime factorization contains only even powers. So we can continue to do the above process until form some i we get $(\lceil \sqrt{n} \rceil + i)^2 \equiv y_i \pmod{n}$, where $y_i = p_1^{2e_1} p_2^{2e_2} ... p_k^{2e_k}$. However, this itself is not superior to trial division, especially for large n, as the number of calculations required for this process would grow exponentially. One seeming deficiency for the above approach is that for each step, we need to factor y_i in order to tell if it can be expressed as a square, which is ineffective. To address a solution to this issue, we must introduce the concept of smoothness.

4.3 Smooth Numbers

A number m is B-smooth if all of its prime factors are less than or equal to B. To illustrate the value of checking for smoothness in our above calculation, we must introduce the following lemma: If $m_1, m_2, ..., m_k$ are positive B-smooth integers, and if $k > \pi(B)$, where $\pi(B)$ denotes the number of primes in the interval [1,B], then some non-empty subsequence of (m_i) has product a square. While we will not cover the proof of this lemma here, this lemma allows us to, rather than check every y_i if it is a square (which requires cumbersome factoring for every i), we simply need to check if y_i is B-smooth. Then, we can simply keep calculating $\lceil \sqrt{n} \rceil + k$ for $k \geq i$, until we get at least $\pi(B)$ numbers $x = \lceil \sqrt{n} \rceil + k$ such that $x^2 - n$ is B-smooth. Then, using this collection of B-smooth numbers, we can form their exponent vectors. The exponent vector $\mathbf{v}$ for a number m that has the prime factorization

$$m = \prod_{i=1}^{\pi(B)} p_i^{\mathbf{v}_i}$$

is defined as $(v_1, v_2, ..., v_{\pi(B)})$. This prime factorization of small primes ($p_i \leq B$) is guaranteed and unique by the Fundamental Theorem of Arithmetic (see

section 2)). It follows that a number is a perfect square if and only if the elements in its exponent vector are all multiples of 2). So it suffices to take the exponent vectors of all the $y_i = x_i^2 - n$ modulo 2, and find a subset of vectors that sum to the 0-vector modulo 2. By the rank-nullity theorem in linear algebra, a linear dependency always exists when there are more vectors than elements in each vector ($\pi(B)$). So, we can simply use Gaussian elimination to find the subset of vectors that add to the 0 vector mod 2 in order to solve the system of linear equations. Then, we simply need to multiply the corresponding y_i's from this calculation together, to find y, and multiply the x_i's together to get x. This yields us our desired congruence of squares equation, from which we can calculate the primes of n using the steps described in section 4.2. The decision of which smoothness-bound (B) to choose in the above algorithm is non-trivial, but requires analytic number theory, which is outside the scope of this paper.

4.4 Sieving for Smoothness Checking

The above steps essentially replaces the necessity of having to factor every y_i to check if it is a square, into the necessity of checking every y_i to see if it is B-smooth, so we must ensure that this is in fact an improvement. Already, in trial division, we only need to check primes up to B. Furthermore, if we let $f(x) = y = x^2 - n$ (the number we need to check for B-smoothness), then we have the following:

$$f(x+kp) = (x+kp)^2 - n = x^2 = 2xkp + (kp)^2 - n = f(x) + 2xkp + (kp)^2 \equiv f(x) \pmod{p}$$

So, by solving $f(x) \equiv 0 \pmod{p}$, to get multiples of p, we get either 0, 1 (if $p|n$), or 2 solutions for $p > 2$. If there are no solutions, there is no need to sieve. If there are 2 solutions, then we simply need add each solution to the corresponding residue class. Indeed, this process is faster than trial division to check for smoothness, and faster than factoring to check if y_i is a perfect square.

5 The Multiple Polynomial Quadratic Sieve

The number of congruences we need to find to construct an equality of the form $x^2 = y^2 \pmod{n}$ is approximately equal to the number of primes in our factor base. To find enough congruences we must choose a large M, which involves dealing with large values of $g(x)$. This presents a problem: large values are less likely to be smooth, which adversely impacts the efficiency of the algorithm.

This shortcoming of the standard QS algorithm is addressed by the so-called multiple polynomial quadratic sieve (MPQS), described in [2]. In this version of the algorithm, we generate the required number of congruences by sieving many polynomials, each over a smaller interval. Instead of sieving the single polynomial

$$g(x) = (x + b)^2 - n,$$

we take polynomials of the form

$$g_{a,b}(x) = (ax + b)^2 - n,$$

with parameters a and b.

As in the standard QS, if we find a value of x such that $g_{a,b}(x)$ is smooth, then letting $u = ax + b$ and $v = g_{a,b}(x)$, we have obtained a smooth relation of the desired form:

$$u^2 \equiv v \pmod{n}.$$

Suppose we choose coefficients a and b for the polynomial $v = g_{a,b}(x)$. Then, analogous to the standard QS above, we have to compute

$$\texttt{soln1}_p = a^{-1}(t_p - b) \pmod{p}$$

$$\texttt{soln2}_p = a^{-1}(-t_p - b) \pmod{p}$$

for each prime p in the factor base, where t_p is the solution to $t^2 \equiv n \pmod{p}$.

As noted in [2], this is a relatively time-consuming computation. If we want to use a small M (i.e. sieve over a small array), the algorithm will spend more time computing the above values than sieving, which is inefficient. To get around this issue, we use the self-initializing quadratic sieve.

6 The Self-Initializing Quadratic Sieve

The self-initializing quadratic sieve (SIQS) is a further improvement on the MPQS algorithm, and it allows us to change polynomials more efficiently, which in turn makes the use of small M feasible.

Once again, we take polynomials of the form $g_{a,b}(x) = (ax + b)^2 - n$. We choose b such that $b^2 - n = ac$ for some integer c and we choose a to be some product of primes q_i in our factor base. Then

$$g_{a,b}(x) = (ax + b)^2 - n = a(ax^2 + 2bx + c)$$

is smooth if and only if $ax^2 + 2bx + c$ is smooth. Furthermore, given a, there are multiple choices of b that satisfy the above constraints. This fact allows us to change polynomials efficiently: for a given a, we can generate a series of polynomials with different values of b in such a way that each polynomial can be calculated quickly from the last.

We use the following idea. Suppose we have

$$a = q_1 \cdots q_s.$$

We want to find integers B_l for $1 \leq l \leq s$ such that

$$B_l^2 \equiv n \pmod{q_l} \quad \text{and} \quad B_l \equiv 0 \pmod{q_j} \text{ for } j \neq l.$$

Then any combination $\pm B_1 \pm \ldots \pm B_s$ will satisfy

$$(\pm B_1 \pm \ldots \pm B_s)^2 \equiv N \pmod{a}$$

by the Chinese Remainder Theorem. So we obtain a value of b that has the desired properties by letting

$$b = \pm B_1 \pm \ldots \pm B_s.$$

This gives us 2^s possible values of b for a given a (but $g_{a,b}$ yields the same residues as $g_{a,-b}$, so we only use 2^{s-1} polynomials).

We can compute these values of B_l using the formula

$$B_l = \gamma_l \left(\frac{a}{q_l} \right),$$

where γ_l is given by

$$\gamma_l \equiv t_l \left(\frac{a}{q_l} \right)^{-1} \pmod{q_l}$$

and t_l is a solution to $t_l^2 \equiv n \pmod{q_l}$. Since t_l has two possible values, so does γ_l. We pick the smallest value of γ_l.

After the values B_l have been found, we can generate the sequence of b values. We start by taking

$$b_1 = B_1 + \ldots + B_s.$$

The subsequent values of b_i are generated through an iterative process. Suppose we have b_i and want to generate b_{i+1}. Write i in binary and let ν denote the location of the rightmost 1, counting from the right (i.e. the rightmost bit has location 1, the next bit from the right has location 2 etc.). Let

$$e = \begin{cases} 1 & \text{if the bit in position } \nu + 1 \text{ is 1} \\ -1 & \text{otherwise} \end{cases}$$

Then

$$b_{i+1} = b_i + eB_\nu.$$

It can be shown (e.g. in [2]) that this procedure will cycle through all 2^{s-1} combinations of the form

$$b = \pm B_1 \pm B_2 \pm \ldots + B_s$$

(where we have fixed the sign of B_s to be positive since we only need 2^{s-1} polynomials).

To do the sieving for a given polynomial $g_{a,b_i}(x)$, we need to find solutions to

$$g_{a,b_i}(x) \equiv 0 \pmod{p}.$$

However, with our iterative method of generating the polynomials, these solutions can also be computed iteratively, significantly speeding up the process. When we generate the first polynomial,

$$g_{a,b_1}(x) = (ax + b_1)^2 - n,$$

we obtain as before two solutions for each p:

$$\texttt{soln1} = a^{-1}(t_p - b_1) \pmod{p}$$

$$\texttt{soln2} = a^{-1}(-t_p - b_1) \pmod{p}.$$

But with SIQS we also compute and store

$$2B_j a^{-1} \pmod{p}$$

for all j. Then to solve $g_{a,b_{i+1}}(x) \equiv 0 \pmod{p}$, we can simply take the two solutions to $g_{a,b_i}(x) \equiv 0 \pmod{p}$ and update them as

$$\texttt{soln1}_{i+1} = \texttt{soln1}_i + e \times 2B_\nu a^{-1} \pmod{p}$$

$$\texttt{soln2}_{i+1} = \texttt{soln2}_i + e \times 2B_\nu a^{-1} \pmod{p}.$$

To summarize, the SIQS algorithm consists of the following steps:

1. Choose bound F for factor base primes. Choose bound M for sieving polynomials in the interval [-M,M].

2. The factor base consists of primes $p < F$ such that

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

   has a solution t. For each prime in the factor base, store this solution as

   $$\texttt{tmem}_p.$$

   Also for each prime, store

   $$\texttt{lp = round(log(p))}.$$

3. Initialize the first polynomial:

   Write

$$a = \prod_{l=1}^{s} q_l$$

   for some primes q_i in the factor base.

   For $l = 1, 2, ..., s$:

   Compute $\gamma_l = \texttt{tmem}_p \times \left(\frac{a}{q_l}\right)^{-1} \pmod{q_l}$

   If $\gamma_l > \frac{q_l}{2}$, replace γ_l with $q_l - \gamma_l$.

   Compute $B_l = \gamma_l \left(\frac{a}{q_l}\right)$

   For p in factor base where $p \nmid a$:

   Compute $\texttt{ainv}_p \equiv a^{-1} \pmod{p}$

   For $j = 1, 2, ..., s$:

   Compute $\texttt{Bainv2}_{j,p} \equiv 2B_j a^{-1} \pmod{p}$

   Let $b_1 = b_1 = B_1 + \ldots + B_s$.

   For p in factor base where $p \nmid a$:

   $\texttt{soln1} = \texttt{ainv} * (\texttt{tmem}_p - b_1) \pmod{p}$

   $\texttt{soln2} = \texttt{ainv} * (-\texttt{tmem}_p - b_1) \pmod{p}$

4. Sieve:

   Initialize array `sieve_array` of zeros, indexed $-M$ to M.

   For each odd p in factor base:

   For i such that $-M \leq \texttt{soln1}_p + ip \leq M$:

   $\texttt{sieve\_array}[\texttt{soln1}_p + \texttt{i*p}]$ `+= lp`

5. Scan the array for values that are at least $\log(M/\sqrt{n})$. Do trial division to factor $g_{a,b}(x)$. If $g_{a,b}(x)$ factors into primes less than F, save the obtained relation.

6. Initialize the $(i+1)$-st polynomial using the i-th polynomial:

   Let ν denote the location of the rightmost 1 in the binary expansion of i.

   Let $e = \begin{cases} 1 & \text{if the bit in position } \nu + 1 \text{ is 1} \\ -1 & \text{otherwise} \end{cases}$

   Compute $b_{i+1} = b_i + eB_\nu$.

   For p in factor base where $p \nmid a$:

   Compute $\texttt{soln1}_{i+1} = \texttt{soln1}_i + e \times \texttt{Bainv2}_{\nu,p} \pmod{p}$

   and $\texttt{soln2}_{i+1} = \texttt{soln2}_i + e \times \texttt{Bainv2}_{\nu,p} \pmod{p}$

7. Repeat steps 4-6 until enough relations are found.

8. Perform the linear algebra step to find a relation of the form $x^2 \equiv y^2 \pmod{n}$.

References

[1] Pomerance, Carl. *Smooth Numbers and the Quadratic Sieve* (2007).

[2] Contini, Scott Patrick. *Factoring Integers with the Self-Initializing Quadratic Sieve* (1997).

[3] Koç, Çetin K., and Sarath N. Arachchige. *A Fast Algorithm for Gaussian Elimination over GF and its Implementation on the GAPP.* (1991).

7 SIQS implementation Details

Initialization stage for first polynomial:
Find primes $q_1, \ldots, q_s$ in the factor base whose product is $\approx \frac{\sqrt{2N}}{M}$. Let $a = \prod_{l=1}^{s} q_l$.
For $l = 1$ to s
    Compute $\gamma = tmem_p \times (a/q_l)^{-1} \bmod q_l$.
    If $\gamma > \frac{q_l}{2}$ then replace γ with $q_l - \gamma$.
    Let $B_l = \frac{a}{q_l} \times \gamma$.
For each prime p in the factor base that does not divide a
    Compute $ainv_p = a^{-1} \bmod p$
    For $j = 1$ to s
        Compute $Bainv2_{j,p} = 2 \times B_j \times ainv_p \bmod p$
Let $b = B_1 + \ldots + B_s$ and $g_{a,b}(x) = (ax + b)^2 - N$.
For each prime p in the factor base that does not divide a
    Compute $soln1_p = ainv \times (tmem_p - b) \bmod p$
    Compute $soln2_p = ainv \times (-tmem_p - b) \bmod p$

Initialization stage for the next $2^{s-1} - 1$ polynomials:
(Assume we just sieved polynomial $\#i$, so we are initializing polynomial $\#(i+1)$, $1 \leq i \leq 2^{s-1} - 1$)
Let ν be the integer satisfying $2^\nu \parallel 2i$.
Let $b = b + 2 \times (-1)^{\lceil i/2^\nu \rceil} \times B_\nu$ and $g_{a,b}(x) = (ax + b)^2 - N$.
For each prime p in the factor base that does not divide a
    Compute $soln1_p = soln1_p + (-1)^{\lceil i/2^\nu \rceil} \times Bainv2_{\nu,p} \bmod p$
    Compute $soln2_p = soln2_p + (-1)^{\lceil i/2^\nu \rceil} \times Bainv2_{\nu,p} \bmod p$

Sieve stage: Initialize a sieve array of length $2M + 1$ to 0's. Assume the indices of the sieve array are from $-M$ to $+M$. For each odd prime p in the factor base, add l_p to the locations $soln1_p + ip$ for all integers i that satisfy $-M \leq soln1_p + ip \leq M$. Similarly, add l_p to the locations $soln2_p + ip$ for all integers i that satisfy $-M \leq soln2_p + ip \leq M$. For the prime $p = 2$, sieve only with $soln1_p$.

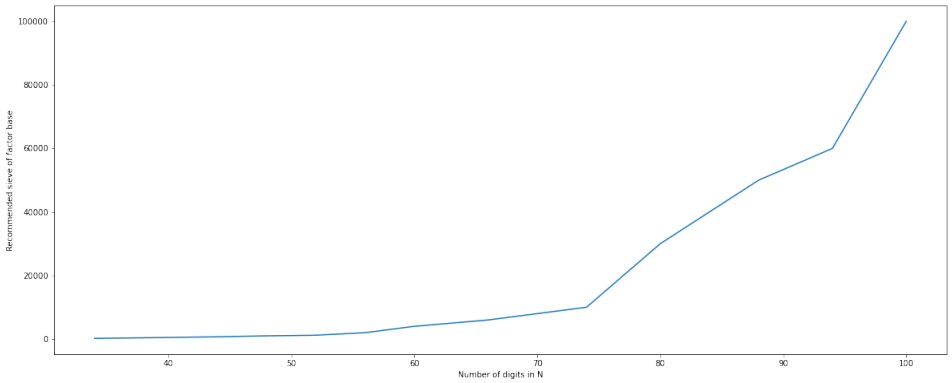
Trial division stage: Scan sieve array for locations x that have accumulated a value of at least $\log(M\sqrt{N})$ minus a small error term. Trial divide $g_{a,b}(x)$. If $g_{a,b}(x)$ factors into primes less than F, then save smooth relation. After scanning entire sieve array, if we have more smooth relations than primes in the factor base, then go to linear algebra stage. Otherwise, go to initialization stage.

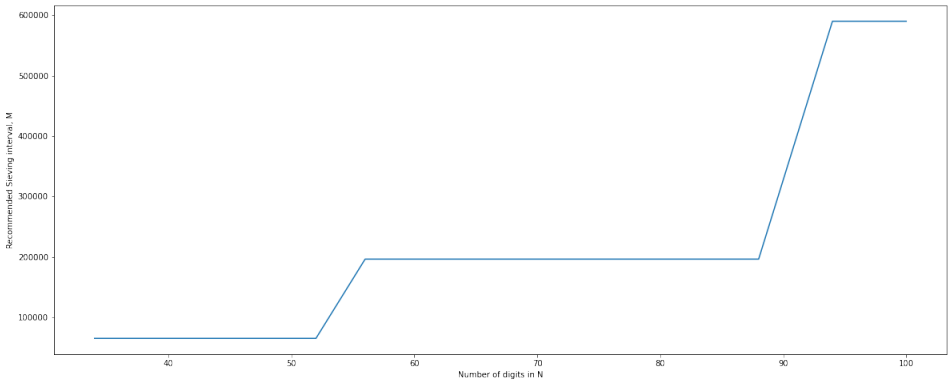
Linear algebra stage: Solve linear algebra problem described in chapter 4. For each null space basis vector, construct relation of form $X^2 \equiv Y^2 \bmod N$. Attempt to factor N by computing $\gcd(X - Y, N)$. If all null space vectors fail to give factorization, then return to sieving stage.

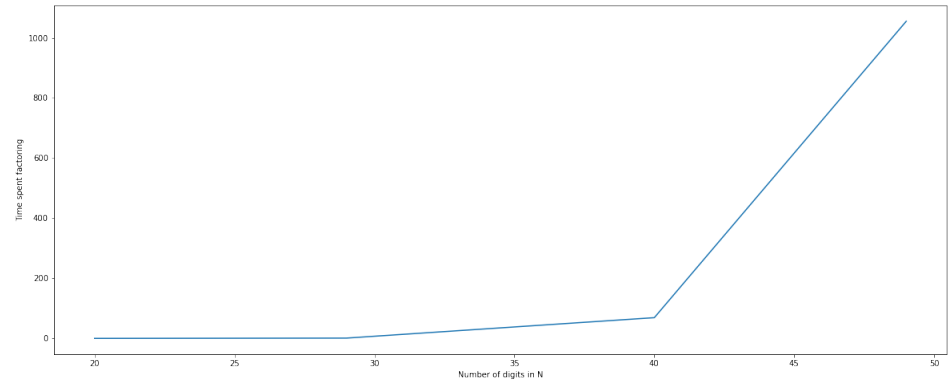
Clearly, in the Quadratic Sieve algorithm and its variations, one of the trickiest and time consuming steps is the Gaussian Elimination process on the sparse yet expansive binary matrix to get pairs of perfect squares mod n. Such pairs would form relations and would be the basis of our factorization. As such, the speed and efficiency of the QS algorithm is dependent on both the sieving and matrix operation procedures. One of the key parts of our implementation is the introduction of Skollman's implementation of Fast Gaussian Elimination. This algorithm is so fast that our implementation now solely depends on the sieiving speed.

This algorithm makes of the binary nature of the vast matrix and performs the linear algebra step in $\Omega(M^2N + M^2 - M)$ bit operations for a matrix of size $M \times N$. Since $N < M$, this is a worst time complexity of approximately $\Omega(M^3)$, a considerable speed-up compared of traditional left null space finding algorithms. This is possible due to clever bit manipulation techniques.

7.1 Some Illustrative Graphs







In [29]:
```
from math import sqrt, log2, ceil, floor, gcd
import random
import sys
from builtins import ValueError
from IPython.display import display, Latex
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

In [30]:
```
MILLER_RABIN_TRIALS = 5
SIQS_TRIAL_DIVISION_EPS = 25
SIQS_MIN_PRIME_POLYNOMIAL = 400
SIQS_MAX_PRIME_POLYNOMIAL = 4000
```

Utility Functions

the follwoing functions will be essential in implementing the SIQS algorithm.

compute moudular exponentiation and return y

$y \equiv x^e \ (modp)$

In [31]:
```
def pow_mod(x, y, z):
    "Calculate (x ** y) % z efficiently."
    number = 1
    while y:
        if y & 1:
            number = number * x % z
        y >>= 1
        x = x * x % z
    return number
```

return index of least significant bit in in the given number

In [32]:
```
def lowest_set_bit(n):
    return int(log2(n^(n&(n-1)))) if n else None
```

Solve the congruence $x^2 = a \ (modp)$ and return x.

We use an implementation of The Tonelli–Shanks algorithm

In [82]:
```
def sqrt_mod_prime(a, p):
    # Algorithm from http://www.mersennewiki.org/index.php/Modular_Square_Root
    assert a < p
    assert is_probable_prime(p)
    if a == 0:
        return 0
    if p == 2:
        return a
```

```
    if p % 2 == 0:
        return None
    p_mod_8 = p % 8
    if p_mod_8 == 1:
        # Shanks method
        q = p // 8
        e = 3
        while q % 2 == 0:
            q //= 2
            e += 1
        while True:
            x = random.randint(2, p - 1)
            z = pow_mod(x, q, p)
            if pow_mod(z, 2 ** (e - 1), p) != 1:
                break
        y = z
        r = e
        x = pow_mod(a, (q - 1) // 2, p)
        v = (a * x) % p
        w = (v * x) % p
        while True:
            if w == 1:
                return v
            k = 1
            while pow_mod(w, 2 ** k, p) != 1:
                k += 1
            d = pow_mod(y, 2 ** (r - k - 1), p)
            y = (d ** 2) % p
            r = k
            v = (d * v) % p
            w = (w * y) % p
    elif p_mod_8 == 5:
        v = pow_mod(2 * a, (p - 5) // 8, p)
        i = (2 * a * v * v) % p
        return (a * v * (i - 1)) % p
    else:
        return pow_mod(a, (p + 1) // 4, p)
```

calculate the Legendre Symbol $(\frac{a}{p})$

$$(\frac{a}{p}) = a^{\frac{p-1}{2}}(modp) = \begin{cases} 1, QR \wedge a \not\equiv 0(modp) \\ 0, a \equiv 0(modp) \\ -1, !QR \end{cases} \quad (1)$$

where QR means a is a quadratic residue (modulo p)

In [83]:

```
def legendre(a, n):
    x = (n - 1)//2
    if x == 0:
        return 1

    z = 1
    a %= n

    while x != 0:
        if x % 2 == 0:
```

```
            a = (a ** 2) % n
            x //= 2
        else:
            x -= 1
            z = (z * a) % n
    return z
```

compute the modular inverse

given a number a and a co-prime number, p, find and return x such that

$ax \equiv 1(modp)$. We use the extended Euclidean algorithm for this.

In [84]:
```
def gcdExtended(a, b):
    if b == 0:
        return a, 1, 0
    else:
        d, x, y = gcdExtended(b, a % b)
        return d, y, x - y * (a // b)

def inv_mod(a,p):
    gcd,x,y = gcdExtended(a,p)
    if x > 0:
        return x
    else:
        return p+x
```

generate small primes less than 1,000,000 and store it in $small_primes$

This will be useful in finding the base factor later in SIQS

In [85]:
```
is_prime = [True for i in range(10**6 + 1)]
small_primes = []
n = 10**6
p = 2

while (p * p <= n):
    if (is_prime[p] == True):
        # Update all multiples of p
        for i in range(p * p, n + 1, p):
            is_prime[i] = False
    p += 1


for i in range(2, n + 1):
    if is_prime[i]:
        global small_primes
        small_primes.append(i)
```

given the array $small\_primes$ of primes $< 1,000,000$, can we write the given number, n, as a perfect power of some combinations of small primes?

If so, factorizing would be immensely easier.

In [86]:
```
def check_perfect_power(n):
    for i in small_primes:
        if n % i == 0:
            c = 0
            while n % i == 0:
                n//= i
                c += 1
            if n == 1:
                return (i, c)
            else:
                return None
```

Miller-Rabin Primality Test

We use the Miller-Rabin Primality Test to test for primality of numbers. Useful for seiving later on. Note that if a number is composite, we can say it is definitely composite using the test. Of course, the probability that a number is a strong pseudoprime to a base b is just 1/4 as always.

In [87]:
```
def is_probable_prime(n):
    assert n >= 2
    # special case 2
    if n == 2:
        return True
    # ensure n is odd
    if n % 2 == 0:
        return False
    # write n-1 as 2**s * d
    # repeatedly try to divide n-1 by 2
    s = 0
    d = n-1
    while True:
        quotient, remainder = divmod(d, 2)
        if remainder == 1:
            break
        s += 1
        d = quotient
    assert(2**s * d == n-1)

    # test the base a to see whether it is a witness for the compositeness of n
    def try_composite(a):
        if pow(a, d, n) == 1:
            return False
        for i in range(s):
            if pow(a, 2**i * d, n) == n-1:
                return False
        return True # n is definitely composite

    for i in range(MILLER_RABIN_TRIALS):
        a = random.randrange(2, n)
        if try_composite(a):
            return False

    return True # no base tested showed n as composite

def _try_composite(a, d, n, s):
    if pow(a, d, n) == 1:
```

```
        return False
    for i in range(s):
        if pow(a, 2**i * d, n) == n-1:
            return False
    return True # n  is definitely composite

def is_prime(n, _precision_for_huge_n=16):
    if n in _known_primes or n in (0, 1):
        return True
    if any((n % p) == 0 for p in _known_primes):
        return False
    d, s = n - 1, 0
    while not d % 2:
        d, s = d >> 1, s + 1
    # Returns exact according to http://primes.utm.edu/prove/prove2_3.html
    if n < 1373653:
        return not any(_try_composite(a, d, n, s) for a in (2, 3))
    if n < 25326001:
        return not any(_try_composite(a, d, n, s) for a in (2, 3, 5))
    if n < 118670087467:
        if n == 3215031751:
            return False
        return not any(_try_composite(a, d, n, s) for a in (2, 3, 5, 7))
    if n < 2152302898747:
        return not any(_try_composite(a, d, n, s) for a in (2, 3, 5, 7, 11))
    if n < 3474749660383:
        return not any(_try_composite(a, d, n, s) for a in (2, 3, 5, 7, 11, 13))
    if n < 341550071728321:
        return not any(_try_composite(a, d, n, s) for a in (2, 3, 5, 7, 11, 13, 17))
    # otherwise
    return not any(_try_composite(a, d, n, s)
                   for a in _known_primes[:_precision_for_huge_n])

_known_primes = [2, 3]
_known_primes += [x for x in range(5, 1000, 2) if is_prime(x)]
```

calculate and return $\sqrt{x}$

A fast square rooting algorithm based on binary search. Useful later for seiving.

In [88]:

```
def sqrt_int(x) :
    if (x == 0 or x == 1) :
        return x
    # Do Binary Search for floor(sqrt(x))
    start = 1
    end = x
    while (start <= end) :
        mid = (start + end) // 2

        # If x is a perfect square
        if (mid*mid == x) :
            return mid
        if (mid * mid < x) :
            start = mid + 1
            ans = mid

        else :
```

```
            end = mid-1
    return ans
```

The polynomial class definition.

We use this in SIQS. The Polynomial is defined as $a_n x^n + a_{n-1}x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0 \ldots$ with the coeffcients picked in reverse order from the given array, $coeff$

In [89]:
```
class Polynomial:
    """A polynomial used for the Self-Initializing Quadratic Sieve."""

    def __init__(self, coeff=[], a=None, b=None):
        self.coeff = coeff
        self.a = a
        self.b = b

    def eval(self, x):
        res = 0
        for a in self.coeff[::-1]:
            res *= x
            res += a
        return res
```

The factor base definition.

let a be the product of several primes $q_1, q_2, q_3, \ldots, q_j$, then let

$soln_1$ and $soln_2$ are thepossible solutions described in algorithm which later make up for our perfect squares for testing.

p is the polynomial we have above.

$tmem$ is used in calculating γ below.

lp is used in calculating the rouding factor we add in the seiving array for each possible solution. This is to account for us not seiving with with powers of primes and logarithmic rounding.

In [90]:
```
class FactorBasePrime:
    """A factor base prime for the Self-Initializing Quadratic Sieve."""

    def __init__(self, p, tmem, lp):
        self.p = p
        self.soln1 = None
        self.soln2 = None
        self.tmem = tmem
        self.lp = lp
        self.ainv = None
```

generate the factor base for the number based on the number sieve suggested.

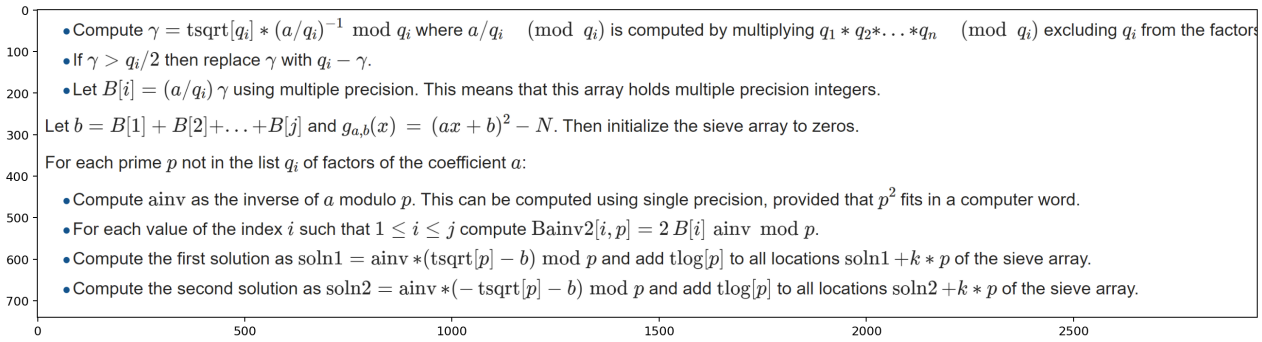
In [91]:
```
def siqs_factor_base_primes(n, nf):
    global small_primes
    factor_base = []
```

```
    for p in small_primes:
        if legendre(n, p) == 1: ## if true, a is a quadratic residue modulo a prime p
            t = sqrt_mod_prime(n % p, p)
            lp = round(log2(p))
            factor_base.append(FactorBasePrime(p, t, lp))
            if len(factor_base) >= nf:
                break
    return factor_base
```

Generating the first polynomial

In [92]:

```
from matplotlib.pyplot import figure
figure(figsize=(18, 16), dpi=180)
img = mpimg.imread('alg1.PNG')
plt.imshow(img)
plt.show()
```



In [93]:

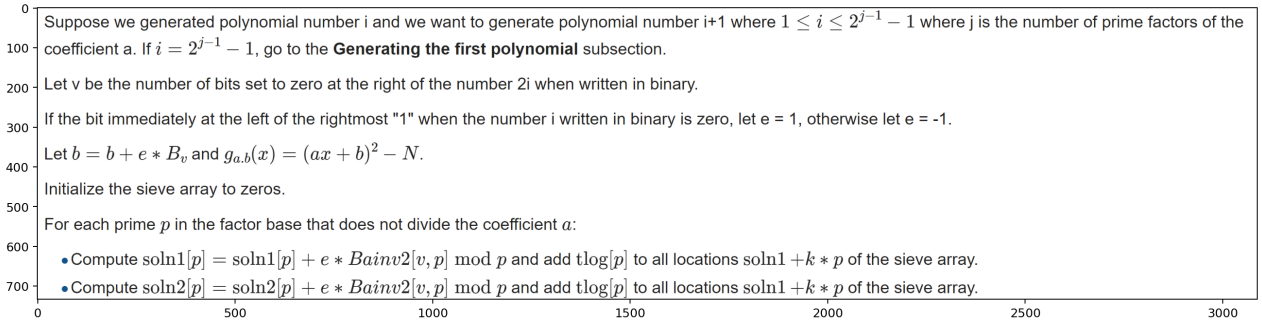
```
def siqs_find_first_poly(n, m, factor_base):
    """Compute the first of a set of polynomials for the Self-
    Initialising Quadratic Sieve.
    """
    p_min_i = None
    p_max_i = None
    for i, fb in enumerate(factor_base):
        if p_min_i is None and fb.p >= SIQS_MIN_PRIME_POLYNOMIAL:
            p_min_i = i
        if p_max_i is None and fb.p > SIQS_MAX_PRIME_POLYNOMIAL:
            p_max_i = i - 1
            break

    if p_max_i is None:
        p_max_i = len(factor_base) - 1
    if p_min_i is None or p_max_i - p_min_i < 20:
        p_min_i = min(p_min_i, 5)

    target = sqrt(2 * float(n)) / m
    target1 = target / ((factor_base[p_min_i].p +
                         factor_base[p_max_i].p) / 2) ** 0.5

    # find q such that the product of factor_base[q_i] is approximately
    # sqrt(2 * n) / m; try a few different sets to find a good one
    best_q, best_a, best_ratio = None, None, None
    for _ in range(30):
        a = 1
        q = []
```

```
        while a < target1:
            p_i = 0
            while p_i == 0 or p_i in q:
                p_i = random.randint(p_min_i, p_max_i)
            p = factor_base[p_i].p
            a *= p
            q.append(p_i)

        ratio = a / target

        # ratio too small seems to be not good
        if (best_ratio is None or (ratio >= 0.9 and ratio < best_ratio) or
                    best_ratio < 0.9 and ratio > best_ratio):
            best_q = q
            best_a = a
            best_ratio = ratio
    a = best_a
    q = best_q

    s = len(q)
    B = []
    for l in range(s):
        fb_l = factor_base[q[l]]
        q_l = fb_l.p
        assert a % q_l == 0
        gamma = (fb_l.tmem * inv_mod(a // q_l, q_l)) % q_l
        if gamma > q_l // 2:
            gamma = q_l - gamma
        B.append(a // q_l * gamma)

    b = sum(B) % a
    b_orig = b
    if (2 * b > a):
        b = a - b

    assert 0 < b
    assert 2 * b <= a
    assert ((b * b - n) % a == 0)

    g = Polynomial([b * b - n, 2 * a * b, a * a], a, b_orig)
    h = Polynomial([b, a])
    for fb in factor_base:
        if a % fb.p != 0:
            fb.ainv = inv_mod(a, fb.p)
            fb.soln1 = (fb.ainv * (fb.tmem - b)) % fb.p
            fb.soln2 = (fb.ainv * (-fb.tmem - b)) % fb.p

    return g, h, B
```

Generating the remaining polynomials

In [94]:

```
from matplotlib.pyplot import figure
figure(figsize=(18, 16), dpi=180)
img = mpimg.imread('alg2.PNG')
plt.imshow(img)
plt.show()
```



In [95]:
```
def siqs_find_next_poly(n, factor_base, i, g, B):
    """Compute the (i+1)-th polynomials for the Self-Initialising
    Quadratic Sieve, given that g is the i-th polynomial.
    """
    v = lowest_set_bit(i) + 1
    z = -1 if ceil(i / (2 ** v)) % 2 == 1 else 1
    b = (g.b + 2 * z * B[v - 1]) % g.a
    a = g.a
    b_orig = b
    if (2 * b > a):
        b = a - b
    assert ((b * b - n) % a == 0)

    g = Polynomial([b * b - n, 2 * a * b, a * a], a, b_orig)
    h = Polynomial([b, a])
    for fb in factor_base:
        if a % fb.p != 0:
            fb.soln1 = (fb.ainv * (fb.tmem - b)) % fb.p
            fb.soln2 = (fb.ainv * (-fb.tmem - b)) % fb.p

    return g, h
```

The sieving step of the SIQS. Return the sieve array after marking.

In [96]:
```
def siqs_sieve(factor_base, m):
    sieve_array = [0] * (2 * m + 1)
    for fb in factor_base:
        if fb.soln1 is None:
            continue
        p = fb.p
        i_start_1 = -((m + fb.soln1) // p)
        a_start_1 = fb.soln1 + i_start_1 * p
        lp = fb.lp
        if p > 20:
            for a in range(a_start_1 + m, 2 * m + 1, p):
                sieve_array[a] += lp

            i_start_2 = -((m + fb.soln2) // p)
            a_start_2 = fb.soln2 + i_start_2 * p
            for a in range(a_start_2 + m, 2 * m + 1, p):
                sieve_array[a] += lp
    return sieve_array
```

Check if the number can be fully factorised into primes from the factors base by trial and error.

If so, return the indices of the factors from the factor base.

In [97]:
```
def siqs_trial_divide(a, factor_base):
    """ If not, return None.
    """
    divisors_idx = []
    for i, fb in enumerate(factor_base):
        if a % fb.p == 0:
            exp = 0
            while a % fb.p == 0:
                a //= fb.p
                exp += 1
            divisors_idx.append((i, exp))
        if a == 1:
            return divisors_idx
    return None


def siqs_trial_division(n, sieve_array, factor_base, smooth_relations, g, h, m,
                        req_relations):
    """Perform the trial division step of the Self-Initializing
    Quadratic Sieve.
    """
    sqrt_n = sqrt(float(n))
    limit = log2(m * sqrt_n) - SIQS_TRIAL_DIVISION_EPS
    for (i, sa) in enumerate(sieve_array):
        if sa >= limit:
            x = i - m
            gx = g.eval(x)
            divisors_idx = siqs_trial_divide(gx, factor_base)
            if divisors_idx is not None:
                u = h.eval(x)
                v = gx
                assert (u * u) % n == v % n
                smooth_relations.append((u, v, divisors_idx))
                if (len(smooth_relations) >= req_relations):
                    return True
    return False
```

Build the matrix for the linear algebra step of the SIQS

In [98]:
```
def siqs_build_matrix(factor_base, smooth_relations):
    """Build the matrix for the linear algebra step of the Quadratic Sieve."""
    fb = len(factor_base)
    M = []
    for sr in smooth_relations:
        mi = [0] * fb
        for j, exp in sr[2]:
            mi[j] = exp % 2
        M.append(mi)

    m = len(M[0])
    cols_binary = [""] * m
    for mi in M:
        for j, mij in enumerate(mi):
            cols_binary[j] += "1" if mij else "0"
    return [int(cols_bin[::-1], 2) for cols_bin in cols_binary], len(M), m
```

Skollmann's fast Gaussian elimination

Utilize skollmann's fast Gaussian elimination to determine pairs of perfect squares mod n. Such pairs would be used to find factors along with our smooth_relations for the input N

In [99]:
```
def siqs_solve_matrix_opt(M_opt, n, m):
    row_is_marked = [False] * n
    pivots = [-1] * m
    for j in range(m):
        if M_opt[j] != 0:
            pivots[j] = lowest_set_bit(M_opt[j])
            row_is_marked[lowest_set_bit(M_opt[j])] = True
            for k in range(m):
                if k != j and (M_opt[k] >> lowest_set_bit(M_opt[j])) & 1:  # test M[i][
                    M_opt[k] ^= M_opt[j]
    perf_squares = []
    for i in range(n):
        if not row_is_marked[i]:
            perfect_sq_indices = [i]
            for j in range(m):
                if (M_opt[j] >> i) & 1:  # test M[i][j] == 1
                    perfect_sq_indices.append(pivots[j])
            perf_squares.append(perfect_sq_indices)
    return perf_squares
```

a, b are calculated from the solution such that a*a* = *b*b (mod n returned by siqs_solve_matrix_opt)

This yields $f = gcd(a - b, n)$, which is a factor

In [100…
```
def siqs_factor_from_square(n, square_indices, smooth_relations):
    """Given one of the solutions returned by siqs_solve_matrix_opt,
    return the factor f determined by f = gcd(a - b, n), where

    Return f, a factor of n (possibly a trivial one).
    """
    a = 1
    b = 1
    for idx in square_indices:
        a *= smooth_relations[idx][0]
        b *= smooth_relations[idx][1]
    b = sqrt_int(b)
    assert (a * a) % n == (b * b) % n
    return gcd(abs(a - b), n)


def siqs_find_factors(n, perfect_squares, smooth_relations):
    """Perform the last step of the Self-Initialising Quadratic Field.
    Given the solutions returned by siqs_solve_matrix_opt, attempt to
    identify a number of (not necessarily prime) factors of n, and
    return them.
    """
    factors = []
    rem = n
    non_prime_factors = set()
    prime_factors = set()
```

```
    for square_indices in perfect_squares:
        fact = siqs_factor_from_square(n, square_indices, smooth_relations)
        if fact != 1 and fact != rem:
            if is_probable_prime(fact):
                if fact not in prime_factors:
                    print ("SIQS: Prime factor found: %d" % fact)
                    prime_factors.add(fact)

                while rem % fact == 0:
                    factors.append(fact)
                    rem //= fact

                if rem == 1:
                    break
                if is_probable_prime(rem):
                    factors.append(rem)
                    rem = 1
                    break
            else:
                if fact not in non_prime_factors:
                    print ("SIQS: Non-prime factor found: %d" % fact)
                    non_prime_factors.add(fact)

    if rem != 1 and non_prime_factors:
        non_prime_factors.add(rem)
        for fact in sorted(siqs_find_more_factors_gcd(non_prime_factors)):
            while fact != 1 and rem % fact == 0:
                print ("SIQS: Prime factor found: %d" % fact)
                factors.append(fact)
                rem //= fact
            if rem == 1 or is_probable_prime(rem):
                break

    if rem != 1:
        factors.append(rem)
    return factors
```

Estimate the number of relations that will be required to pass filtering based on previous experiments.

We choose parameters sieve of factor base and m for sieving in $[-m, m]$. based on https://github.com/eniac/faas/blob/master/factor/factor.py

In [109…

```
def siqs_find_more_factors_gcd(numbers):
    res = set()
    for n in numbers:
        res.add(n)
        for m in numbers:
            if n != m:
                fact = gcd(n, m)
                if fact != 1 and fact != n and fact != m:
                    if fact not in res:
                        print("SIQS: GCD found non-trivial factor: %d" % fact)
                        res.add(fact)
                    res.add(n // fact)
                    res.add(m // fact)
    return res
```

We choose parameters nf (sieve of factor base) and m (for sieving in [-m,m]).

This is based on msieve-1.52

In [101…

```
def siqs_choose_nf_m(number_of_digits):
    if number_of_digits <= 34:
        return 200, 65536
    if number_of_digits <= 36:
        return 300, 65536
    if number_of_digits <= 38:
        return 400, 65536
    if number_of_digits <= 40:
        return 500, 65536
    if number_of_digits <= 42:
        return 600, 65536
    if number_of_digits <= 44:
        return 700, 65536
    if number_of_digits <= 48:
        return 1000, 65536
    if number_of_digits <= 52:
        return 1200, 65536
    if number_of_digits <= 56:
        return 2000, 65536 * 3
    if number_of_digits <= 60:
        return 4000, 65536 * 3
    if number_of_digits <= 66:
        return 6000, 65536 * 3
    if number_of_digits <= 74:
        return 10000, 65536 * 3
    if number_of_digits <= 80:
        return 30000, 65536 * 3
    if number_of_digits <= 88:
        return 50000, 65536 * 3
    if number_of_digits <= 94:
        return 60000, 65536 * 9
    return 100000, 65536 * 9
```

The Self-Initializing Quadratic Sieve algorithm

We need this identify one or more non-trivial factors of the given number

In [102…

```
def siqs_factorise(n):

    dig = len(str(n))
    nf, m = siqs_choose_nf_m(dig)

    factor_base = siqs_factor_base_primes(n, nf)

    required_relations_ratio = 1.05
    success = False
    smooth_relations = []
    prev_cnt = 0
    i_poly = 0
    while not success:
        print("*** Step 1/2: Finding smooth relations ***")
        required_relations = round(len(factor_base) * required_relations_ratio)
        print("Target: %d relations" % required_relations)
```

```
        enough_relations = False
        while not enough_relations:
            if i_poly == 0:
                g, h, B = siqs_find_first_poly(n, m, factor_base)
            else:
                g, h = siqs_find_next_poly(n, factor_base, i_poly, g, B)
            i_poly += 1
            if i_poly >= 2 ** (len(B) - 1):
                i_poly = 0
            sieve_array = siqs_sieve(factor_base, m)

            enough_relations = siqs_trial_division(
                n, sieve_array, factor_base, smooth_relations,
                g, h, m, required_relations)

            if (len(smooth_relations) >= required_relations or
                i_poly % 8 == 0 and len(smooth_relations) > prev_cnt):

                print("\rTotal %d/%d relations." % (len(smooth_relations), required_rel
                prev_cnt = len(smooth_relations)

        print("*** Step 2/2: Linear Algebra ***")
        print("Building matrix for linear algebra step...")

        M_opt, M_n, M_m = siqs_build_matrix(factor_base, smooth_relations)

        print("Finding perfect squares using matrix...")
        perfect_squares = siqs_solve_matrix_opt(M_opt, M_n, M_m)

        print("Finding factors from perfect squares...")
        factors = siqs_find_factors(n, perfect_squares, smooth_relations)
        if len(factors) > 1:
            success = True
        else:
            print("Failed to find a solution. Finding more relations...")
            required_relations_ratio += 0.05

    return factors
```

gateway function to factor n

In [103…

```
def find_all_prime_factors(n):
    """Return all prime factors of the given number n. Assume that n
    does not have very small factors and that the global small_primes
    has already been initialised.
    """
    rem = n
    factors = []

    print("Checking whether %d is a perfect power..." % n)
    perfect_power = check_perfect_power(n)
    if perfect_power:
        print("No small factors found, it is a perfect square with %d is %d^%d" % (n, p
        factors = [perfect_power[0]]

    else:
        print("Not a perfect power. Intializing SIQS")
        while rem > 1:
            if is_probable_prime(rem):
```

```
                factors.append(rem)
                break
            sub_factors = siqs_factorise(rem)
            for f in sub_factors:
                print("Prime factor found: %d" % f)
                assert is_probable_prime(f)
                assert rem % f == 0
                while rem % f == 0:
                    rem //= f
                    factors.append(f)

    return factors


### THESE TWO CAN BE JOINED IN
```

In [104…

```
def product(factors):
    """Return the product of all numbers in the given list."""
    prod = 1
    for f in factors:
        prod *= f
    return prod


def factorise(n):
    """Factorise the given integer n >= 1 into its prime factors."""

    if type(n) != int or n < 1:
        raise ValueError("Number needs to be an integer >= 1")

    print("Factorising %d (%d digits)..." % (n, len(str(n))))
    if n == 1:
        return []

    if is_probable_prime(n):
        return [n]
    factors = []
    for fr in find_all_prime_factors(n):
        factors.append(fr)
    factors.sort()
    assert product(factors) == n
    for p in factors:
        assert is_probable_prime(p)
    return factors
```

In [ ]:

```
l = [16921456439215439701, 468395662999369192342467268O9 ,617283580864197520363830491961
time1 = []
time2 = []
for N in l:
    print("\nSuccess. Prime factors: %s" % factorise(N))
    time1.append((time.time() - start_time))
```

```
Factorising 16921456439215439701 (20 digits)...
Checking whether 16921456439215439701 is a perfect power...
Not a perfect power. Intializing SIQS
*** Step 1/2: Finding smooth relations ***
Target: 210 relations
```

```
Total 210/210 relations.*** Step 2/2: Linear Algebra ***
Building matrix for linear algebra step...
Finding perfect squares using matrix...
Finding factors from perfect squares...
SIQS: Prime factor found: 2860486313
Prime factor found: 2860486313
Prime factor found: 5915587277

Success. Prime factors: [2860486313, 5915587277]
Factorising 468395662999369191234246726809 (29 digits)...
Checking whether 468395662999369191234246726809 is a perfect power...
Not a perfect power. Intializing SIQS
*** Step 1/2: Finding smooth relations ***
Target: 210 relations
Total 210/210 relations.*** Step 2/2: Linear Algebra ***
Building matrix for linear algebra step...
Finding perfect squares using matrix...
Finding factors from perfect squares...
SIQS: Prime factor found: 100000000105583
Prime factor found: 100000000105583
Prime factor found: 468395662504823

Success. Prime factors: [100000000105583, 468395662504823]
Factorising 617283580864197520363830491969135846966 3 (40 digits)...
Checking whether 617283580864197520363830491969135846966 3 is a perfect power...
Not a perfect power. Intializing SIQS
*** Step 1/2: Finding smooth relations ***
Target: 525 relations
Total 525/525 relations.*** Step 2/2: Linear Algebra ***
Building matrix for linear algebra step...
Finding perfect squares using matrix...
Finding factors from perfect squares...
SIQS: Prime factor found: 555555222777777773333
Prime factor found: 555555222777777773333
Prime factor found: 11111111111111111011

Success. Prime factors: [11111111111111111011, 555555222777777773333]
Factorising 3744843080529615909019181510330554205500926021947 (49 digits)...
Checking whether 3744843080529615909019181510330554205500926021947 is a perfect power...
Not a perfect power. Intializing SIQS
*** Step 1/2: Finding smooth relations ***
Target: 1260 relations
Total 1260/1260 relations.*** Step 2/2: Linear Algebra ***
Building matrix for linear algebra step...
Finding perfect squares using matrix...
Finding factors from perfect squares...
```

reference codes

https://github.com/skollmann/PyFactorise/blob/master/factorise.py

https://gist.github.com/kashapovd/486b712bf0cd232b904f5fc5a63cf3d8

https://eli.thegreenplace.net/2009/03/07/computing-modular-square-roots-in-python

and many others....

In [ ]: