Integer Factorization Algorithms

Daniel Vogel, Yinka Onayemi, Vlad Murad

May 2, 2016

1 Introduction

Integer factorization represents the decomposition of integers into a product of smaller integers. This has been a problem of great interest throughout the ages, but with the recent growth of sensitive personal data being stored online, integer factorization has become a very relevant problem, as most cryptosystems depend on this problem not being easily solvable. Fortunately for most, unfortunately for others, there is no known algorithm for factoring very large numbers in a computationally feasible time frame. However, with the dawn of quantum computers, the time complexity of factorization algorithms can drastically improve, causing most cryptosystems, like RSA, to become vulnerable to attacks.

Throughout this paper we will analyze the most famous factorization algorithms, starting from the oldest and easiest ones and ending with the state of the art General Field Number Sieve algorithm. We will delve into the historical context of each algorithm and try to provide a basic understanding for each method. Furthermore, we will present small examples to aid understanding, we will provide time complexity analysis and we will also provide pseudocode or implementations of the algorithms in Sage or Python.

2 Fermat's Method

2.1 History and Context

Fermat was a French lawyer and mathematician who made vast contributions to the field of number theory. He is best known for Fermat's last theorem, which was posthumously found in the margin of his copy of Diophantus' Arithmetica.

2.2 Description and Mathematical Motivation

Any integer can be written as the difference of two squares. Fermat's factorization method allows us to factor an integer into prime factors. However, with this method, the farther away the two factors are from each other the longer it takes to factor.

2.3 Algorithm

2.3.1 Pseudocode

To find a factorization $n = ab$ where a and b are prime factors of n

- **Step 1:** Let y = 1.
- **Step 2:** Compute $x = \sqrt{n + y^2}$
- **Step 3:** If the integer part of x is equal to x, then we found $a = (x - y) and b = (x + y)$ for which $n = ab$. Otherwise increment x and move back to step 2.

2.3.2 Sage Code

```
def Fermat(n):
    x = int(sqrt(n))+1
    while x<n:
        y = sqrt(x * x - n)
        if y == int(y):
            break
        x += 1
    return (x-y,x+y)
```

2.4 Example

Let $n = 187$ and $y = 1$.

Begin iterating on the value of y, until the value of $x = \sqrt{187 + y^2}$ is an integer:

- $x = \sqrt{187 + 1^2} = 13.7113$
- $x = \sqrt{187 + 2^2} = 13.8202$
- $x = \sqrt{187 + 3^2} = 14$

Since $y = 3$ yielded an integer for x, we use the pair $(x, y) = (14, 3)$ to compute $a = (x + y) = 17; b = (x - y) = 11$. Thus, the factorization of n in $n = ab$

3 Euler's Method

3.1 History and Context

Leonhard Euler was a renowned Swiss mathematician who made influential contributions to the field of number theory. Euler based a lot of his earlier works on that of Fermat, Newton, Euclid, and Mersenne. In fact, Mersenne did much of the ground work that made it possible for Euler to come up with his prime factorization method by introducing the idea that two distinct representations of an odd positive integer may lead to a factorization. Using this methodology, Euler was able to come up with a method for prime factorization using the sums of squares. Euler's method is also similar to that of Fermat's – who was almost certainly a motivator. The main difference, however, is that Fermat's method requires the two prime factors to be relatively close to each other.

3.2 Description and Mathematical Motivation

Euler's factorization method is a technique used for factoring an integer by writing it as a sum of two different sets of two squares. As said earlier, Euler's method is more efficient than Fermat's due to its ability to be used as long as you can find representations of the integers as sums of two squares relatively easily. Euler's method relies on the Brahmagupta-Fibonacci identity which states that the product of two squares is a sum of two squares. The famous example of Euler's method is 1000009 because of its ability to be written as the sum of 1000^2 and 3^2 and 972^2 and 235^2.

3.3 Algorithm

3.3.1 Pseudcode

- **Step 1:** Let n be the number we want to factor.
- **Step 2:** Compute a, b, c, d such that $n = a^2 + b^2 = c^2 + d^2$ and observe that $(a - c)(a + c) = (d - b)(d + b)$
- **Step 3:** Let $k = \gcd(a - c, d - b)$ and $h = gcd(a + c, d + b)$
- **Step 4:** Define constants l, l', k, k' such that $(a - c) = kl, (d - b) = km$ and the $gcd(l, m) = 1$; $(a+c) = hm'$, $(d+b) = hl'$ and the $gcd(l', m') = 1$. Thus, replace in $(a-c)(a+c) = (d-b)(d+b)$ and obtain $klhm' = kmhl'$, followed by $lm' = l'm$
- **Step 5:** Because each factor is a sum of two squares, one of them must contain both even numbers: Either (k, h) or (l, m) will be even. If we assume that (k, h) is even. Then, the formula becomes $n = ((k/2)^2 + (h/2)^2)(l^2+m^2)$, where the factors of n are $((k/2)^2+(h/2)^2)$ and (l^2+m^2).

3.3.2 Python Code

```
import numpy as np
from fractions import gcd

def euler(n):
    # The method is valid only if n is odd
    # Check if n is odd
    if n%2==0:
        return ((n/2, 2) if n>2 else (2, 1))

    factors = (n, 1)
    end = n
    a = 0
    solutionsFound = []
    firstb = -1
    while a < end and len(solutionsFound) < 2:
        b_square = n - a**2
        if (b_square > 0):
            b = np.sqrt(b_square)
            #so, a2 + b2 = n
            if np.trunc(b) == b and a != firstb and b != firstb: # If b is a pe
                firstb = b
                solutionsFound.append([int(b), a])
        a+=1
    if len(solutionsFound) < 2:
        return str(n) + " is of the form 4k+3"
    print("Solutions Found: " + str(solutionsFound))
    a = solutionsFound[0][0]
    b = solutionsFound[0][1]
    c = solutionsFound[1][0]
    d = solutionsFound[1][1]
    print("a^2 + b^2: " + str(a**2+b**2) + " = c^2 + d^2: " + str(c**2+d**2))
    #so, a2 - c2 = d2 - b2, (a-c)(a+c) = (d-b)(d+b)
    k = gcd(a-c, d-b)
    h = gcd(a+c, d+b)
    #so, a-c = kl, d-b = km, m,l in Z
    m = gcd(a+c, d-b)
    l = gcd(a-c, d+b)

    n = (k**2 + h**2)*(l**2 + m**2)
    print(n/4)
    print(k, h, m, l)

    return [(k**2 + h**2)/2, (l**2 + m**2)/2]


for i in range(10, 10000, 81):
    print("\nFactors of " + str(i))
    print(euler(i))
    print("\n")
```

3.4 Example

Let $n = 100009$

Compute $n = 1000^2 + 3^2 = 972^2 + 235^2$

Then we have $a = 1000; b = 3; c = 972; d = 235$ and $a - c = 28; a + c = 1972; d - b = 232; d + b = 238$

Compute $\gcd(28, 282) = k = 4$ and $\gcd(1972, 238) = h = 34; l = 7; m = 58$

So, we can use the formula $((k/2)^2 + (h/2)^2)$ and $(l^2 + m^2)$ to compute the factorization $n = 100009 = ((4/2)^2 + (34/2)^2) * (7^2 + 58^2) = (22 + 172) * (72 + 582) = (4 + 289) * (49 + 3364) = (293) * (3413)$.

Thus, 293 and 3413 are the two prime factors of 1000009.

4 Pollard's Rho Algorithm

4.1 History and Context

The original reference for this algorithm is a 1975 paper by John M. Pollard called "A Monte Carlo method for factorization". Richard Brent published an improved, quicker algorithm, which build on Pollard's Rho in 1980. His new, faster method is called Brent-Pollard rho factorization.

4.2 Description and Mathematical Motivation

Pollard's Rho is a heuristic, randomized algorithm, whose expected time complexity is $O(N^{1/2})$ operations, and uses only $O(1)$ space.

The way the algorithm works is by picking random integer and test potential factors for our number n that we wish to factor. However, we use the following trick to improve our odds of arriving at a factor of n:

- Suppose we pick a number at random from 1 to 1000 (let's say we want to pick the number 7). We see that there is a 1/1000 chance of picking 7. No random pick is more probable than another.
- Now, suppose we modify the problem a little. We pick two random numbers $i, j \in [1, 1000]$. What are the odds that $i - j = 7$? Note that i, j need not be different. There are roughly 958*2 possible values of i, j that ensure that $i - j = 42$ and the probability reduces to $\frac{2*958}{1000*1000}$, which is roughly $\frac{1}{500}$ probability.
- Rather than insist that we pick one number and it have it be exactly 42, if we are allowed to pick two and just ask that their difference be 42, the odds improve.
- Now, What if we pick k numbers $x_1, ..., x_k$ in the range $[1, 1000]$ and ask whether any two numbers x_i, x_j satisfy $x_i - x_j = 42$? How do the odds relate to k? This is the idea that Pollard gets at in his Rho algorithm - picking random numbers within a range in order to increase the chances of happening across a factor of N.

4.3 Algorithm

4.3.1 Python Code

```
from fractions import gcd
from random import randint

#Input : A number N to be factorized
#Output : A divisor of N
#If x mod 2 is 0, return 2


def pollardRho(N):
        if N%2==0:
                return 2
        x = randint(1, N-1)
        y = x
        c = randint(1, N-1)
        g = 1
        while g==1:
            x = ((x*x)%N+c)%N
            y = ((y*y)%N+c)%N
            y = ((y*y)%N+c)%N
            g = gcd(abs(x-y),N)
        return g

print("A factor of 1000009 according to pollard is:")
print(pollardRho(1000009))
print("1000009/293 = " + str(1000009/293))
```

4.4 Example

Let $n = 8051$ and let our increment function be $f(x) = (x^2 + 1) \mod 8051$. Let us iterate on a variable i until we find a factor.

When i = 1: $x_i = 5, y_i = 26, \gcd(|x_i - y_i|, 8051) = 1$.

When i = 2: $x_i = 5^2 + 1 = 26, y_i = 7474, \gcd(|x_i - y_i|, 8051) = 1$.

When i = 3: $x_i = 26^2 + 1 = 677, y_i = 871, \gcd(|x_i - y_i|, 8051) = 97$.

Thus, 97 is a factor of 8051, as 97*83 = 8051.

5 Dixon's Algorithm

5.1 History and Context

Dixon's Algorithm was introduced by John D. Dixon in 1981 in his paper "Asymptotically Fast Factorization of Integers". However, as he later declared, the ideas used in conceiving the algorithm are not his own creation, the mathematician only being responsible for collating previously formulated concepts into a formal algorithm and providing time complexity analysis for it: "...the method goes back much further than my paper in 1981[...]. The fact is that the idea in one form or another had been around for a much longer time, but no-one had been able to give a rigorous analysis of the time complexity of the versions which were used."

Dixon's Algorithm represents an improvement to Fermat's method that stems from the idea that instead of looking for pairs (x, y) such that $x^2 - y^2 = n$, it suffices to look for pairs for which $x^2 \equiv y^2 \mod n$. This idea stands at the foundation of most modern time factorization algorithms. Although this fact was introduced by Gauss and Seelhoff in the 19th century, it was only brought into the context of factorization algorithms in 1920 by Kraitchik. He introduced the function $Q(x) = x^2 \mod n$ and used the property of some $Q(x)$'s to factor easily over a basis of primes to form pairs of squares $\mod n$. In 1931, Lehmer and Powers, suggested a faster method of finding pairs of the form $(x, Q(x))$, by considering the sequence $\frac{x_i}{z_i} \to \sqrt{n}$ and computing $Q(x_i) = {x_i}^2 - n{z_i}^2$.[Tale of Two Sieves]

In 1975, in a paper by Morrison and Brillhart, a systematic method of combining pairs of the form $(x, Q(x))$ such that they yield perfect squares was presented. In the late 1970s, Schroeppel was the first to provide time complexity analysis of the algorithm in some of his unpublished correspondence. However, it was only in Dixon's 1981 paper that a rigorous complexity analysis was carried out. Finally, in the same year, Carl Pomerance devised the Quadratic Sieve Algorithm, an algorithm that is similar to Dixon's method, but is almost twice as fast, and represents the foundation of the best known factorization algorithm currently in existence: the General Number Field Sieve.

5.2 Description and Mathematical Motivation

Let n be an odd integer that is divisible by at least two primes. We ignore the case when n is even, since $n = 2k$ is a valid factorization, and the case when $n = p^a$ where p is a prime, since we can take the first $\log n$ roots of n and check if they are integers to quickly solve this possibility. Using Fermat's algorithm, we would look for a pair (x, y) such that $x^2 - y^2 = n$. However, this method is very inefficient for large integers that don't have factors close to $\sqrt{n}$ (in fact, the integers n that have factors close to $\sqrt{n}$ represent a very small portion of $\mathbb{Z}$). Thus, in order to make the problem computationally easier, we content ourselves to finding pairs (x, y) such that $x^2 \equiv y^2 \mod n$ and $x \not\equiv \pm y$ $\mod n$(pairs for which $x \equiv \pm y \mod n$ yield trivial factorizations of the form $n = n * 1$. It can be easily shown that for a composite odd integer n that has l prime factors, there are 2^l such pairs, out of which 2/3 are pairs (x, y) for which $x \not\equiv \pm y \mod n$.

Now, suppose we have found a pair (x, y) such that $x^2 \equiv y^2 \mod n$ and $x \not\equiv \pm y \mod n$, then $(x-y)(x+y) \equiv 0 \mod n$. However since $x \not\equiv \pm y \mod n$, it follows that $(x \pm y) \not\equiv 0 \mod n$. Thus, $n|(x - y)(x + y)$, but $n \not| (x \pm y)$. This implies that the $\gcd(x - y, n)$ and the $\gcd(x + y, n)$ are proper divisors n. It should be noted that computing the gcd of two integers is easily done via Euclid's algorithm. So, our final objective using Dixon's Algorithm is to find the $\gcd(x - y, n)$.

It is easy to see that our greatest challenge is to find pairs (x, y) efficiently. To do so we will divide this problem into multiple smaller problems. So, instead of computing (x, y) directly, we will try to find k pairs (x_i, y_i) such that $x^2 \equiv {x_1}^2 ... {x_k}^2 \equiv {y_1}^2 ... {y_k}^2 \equiv y^2 \mod n$ and computing the pairs (x_i, y_i) is an easier problem than computing (x, y).

Let us first highlight how the difficulty of computing pairs (x, y) ensues from using Fermat's Method. Using this method, we define $Q(x)$ as $Q(x) = x^2 - n$. So, we are looking for x's such that $Q(x)$ is a perfect square. Since the initial value of x is ceil($\sqrt{n}$), if n is very large, the function $Q(x)$ will grow very quickly towards integers where perfect squares are harder and harder to find, provided that we don't get lucky using the very first values of x. Thus, to make the problem easier, we will do two things. First, we will make the size of $Q(x)$ more manageable, and second we will relax the condition that $Q(x)$ has to be a perfect square, to $Q(x)$ has to be "easy to factor".

The first step is to contain the size of $Q(x)$, while ensuring that $Q(x) \equiv x^2$ $\mod n$(why this condition is necessary, will become apparent later on). The best we can do is to have $1 \leq Q(x) \leq n$ by defining $Q(x)$ as one of the following functions:

- $Q(x) = x^2 \mod n$. This is the function used by Kraitchik in his factorization method.
- $Q(x_i) = {x_i}^2 - n{z_i}^2$, where $\frac{x_i}{z_i} \rightarrow \sqrt{n}$ represents the continued fraction sequence to $\sqrt{n}$. This function due to Lehmer and Powers does a slightly better job at keeping $Q(x)$ as small as possible.

The second step is to replace "perfect square" with "easy to factor". But what does this mean? As it turns out we let the user of the algorithm decide this by allowing him to choose a number B, and say that an integer is "easy to factor"(or B-smooth) if all its prime factors are in the prime basis $\{p_1, p_2, ..., p_t\}$ comprising of the prime numbers less than B. While checking if a number is B-smooth might seem akin to finding its factorization, this is far from being the case. To determine the B-smoothness of an integer a, we successively divide a by all the p_i's, starting from p_1, and moving to the next element in the basis, p_{i+1}, only if p_i does not divide the current value of a. If at the end of this procedure which takes approximately $\pi(a)$(number of primes less than a) steps $a = 1$, then a is B-smooth. So, using the relaxed condition, we are now looking for values of x such that $Q(x)$ is B-smooth for some value of B. While the value of B is directly proportional to the number of integers that are B-smooth (and, as we will see later, the choice of B relative to n impacts the speed of the algorithm), it should be obvious to the reader that irrespective of the value of B, it is much easier to find B-smooth numbers than perfect squares.

Now, using the fact that $Q(x_i) \equiv {x_i}^2 \mod n$ and that we can quickly find pairs $(x_i, Q(x_i))$ such that $Q(x_i)$ is B-smooth, we compute multiple such pairs until we are able to form a subset of them for which $Q(x_{i_1})...Q(x_{i_j})$ is a perfect square. Thus, suppose such a subset is found, then we can compute a desired (x, y) pair by letting $x = x_{i_1}...x_{i_j}$ and $y = \sqrt{Q(x_{i_1})...Q(x_{i_j})}$ since $x^2 = (x_{i_1}...x_{i_j})^2 = {x_{i_1}}^2...{x_{i_j}}^2 \equiv Q(x_{i_1})...Q(x_{i_j}) = y^2 \mod n$(in practice, we compute y by storing the decomposition with respect to the prime basis of each $Q(x_{i_j})$'s, because this allows us to add the exponents, and compute the square root just by halving their sum).

Thus, if we can find pairs $(x_i, Q(x_i))$ such that the product of $Q(x_i)$'s yields a perfect square, we can compute $x^2 \equiv y^2 \mod n$ as shown above. However, there are a couple of pieces missing from the puzzle. First of all, how do we check if the product of $Q(x_i)$'s forms a perfect square, and more importantly, how can we be sure that a perfect square can always be formed. Second of all, we have to remember that in order to have a proper factorization of n, we need to have $x^2 \equiv y^2 \mod n$ and $x \not\equiv \pm y \mod n$. So, how can we be sure that we didn't arrive at a trivial solution for which $x \not\equiv \pm y \mod n$.

To solve the first problem we turn to linear algebra and use a method proposed by Morrison and Brillhart. Suppose that the prime factorization of a B-smooth integer q is $q = {p_1}^{a_1}...{p_t}^{a_t}$ where the p_i's are elements of the prime basis of size t derived from B and a_i's are the exponents of each prime, with the possibility of $a_i = 0$. Morrison and Brillhart define a function v_B as $v_B(q) = \begin{bmatrix} a_1 \mod 2 \\ . \\ . \\ a_t \mod 2 \end{bmatrix}$, where the resulting vector has the same dimension t as the prime basis derived from B. Since q is a perfect square if and only if all of the exponents of its prime factors are even, it follows that q is a perfect square if and only if $v_B(q) = \begin{bmatrix} 0 \\ . \\ . \\ 0 \end{bmatrix}$. Furthermore, if $q = {p_1}^{a_1}...{p_t}^{a_t}$ and $q' = {p'_1}^{a'_1}...{p'_t}^{a'_t}$, then $v_B(qq') = v_B(q) \oplus v_B(q')$. Thus, given a set of $Q(x_i)$'s, we would like to find a subset of it such that $v_B(Q(x_{i_1}) \oplus ... \oplus v_B(Q(x_{i_j})) = \begin{bmatrix} 0 \\ . \\ . \\ 0 \end{bmatrix}$. To show that a perfect square can always be formed from a set of $Q(x_i)$'s, we use the property that given $t+1$ vectors of dimension t, say $\{b_1, ..., b_{t+1}\}$, then we can write 0 as a linear combination of the vectors in this set. For our case specifically, this property says that we can find a subset of the set of B-smooth integers $\{b_1, ..., b_{t+1}\}$ such that $v_B(b_{i1}) \oplus ... \oplus v_B(b_{ij}) = \begin{bmatrix} 0 \\ . \\ . \\ 0 \end{bmatrix}$. Thus, for any value of B, if we can find $t+1$ $Q(x)$'s, that are B-smooth(and it is not hard to show that we can), then we can certainly find a subset of them that form a perfect square when multiplied together. Algorithmically, to find this subset, we pick at least $t+1$ pairs of the form $(x_i, Q(x_i))$ where $Q(x)$ is B-smooth and we form a matrix $A = \begin{bmatrix} v_B(Q(x_1)) & v_B(Q(x_2)) & ... & v_B(Q(x_{t+1})) \end{bmatrix}$. Then we solve for vectors $q = \begin{bmatrix} q_1 \\ . \\ . \\ q_t \end{bmatrix} \neq \begin{bmatrix} 0 \\ . \\ . \\ 0 \end{bmatrix}$ the equality $Aq = 0$. The resulting vector will have a $q_i = 1$ if $Q(x_i)$ needs to be included in the product that will form a square, and 0 otherwise. This way we can compute the set of B-smooth $Q(x_i)$'s needed to obtain the desired (x, y) pairs.

The final problem is whether the pairs $(x, Q(x))$ computed as above yield a pair (x, y) for which $x^2 \equiv y^2 \mod n$ and $x \not\equiv \pm y \mod n$. The answer is that in general they don't. However, by adding enough pairs $(x, Q(x))$ and recomputing $Aq = 0$ we are ensured to eventually find a subset of $(x, Q(x))$'s such that $x^2 \equiv y^2 \mod n$ where $x \not\equiv \pm y \mod n$. Once a proper pair (x, y) is found, we can compute $d = \gcd(x - y, n)$ and output the factorization n = d * (n/d).

5.3 Algorithm

5.3.1 Pseudocode

- **Step 1:** Let $x_i = ceil(\sqrt{n})$, let $P = \{p_1, ...p_t\}$ be the prime basis of primes less than the parameter B. Initialize X, the list of x_i's, V, the list of vectors $v_B(Q(x_i))$, and Q_{exp}, the list of the exponents in the prime decomposition of $Q(x_i)$'s with respect to P, to empty lists. The lists X, V and Q_{exp} will be indexed in such a way that $X[i] = x_i$, $V[i] = v_B(Q(x_i))$ and $Q_{exp}[i] = [a_1, a_2, ..., a_t]$, where $Q(x_i) = {p_1}^{a_1}...{p_t}^{a_t}$.

- **Step 2:** If $x_i \geq n$, exit the algorithm unsuccessfully. Otherwise, compute $Q(x_i) = {x_i}^2 \mod n$(we are using Kraitchik's variation of the algorithm) and move on to step 3.

- **Step 3:** Determine if $Q(x_i)$ is B-smooth by succesively dividing $Q(x_i)$ by all the p_j's, starting from p_1, and moving to the next element in the basis, p_{j+1}, only if p_j does not divide the current value of $Q(x_i)$. If at the end of this procedure $Q(x_i) = 1$, then $Q(x_i)$ is B-smooth. If $Q(x_i) = {p_1}^{a_1}...{p_t}^{a_t}$

is B-smooth, append x_i to X, $v_B(Q(x_i))$ to V and $[a_1, a_2, ..., a_t]$ to Q_{exp}. If X, V and Q_{exp} have size at least $t+1$, where t is the dimension of the prime basis, move to step 4. Otherwise, increment x and go back to step 2.

- **Step 4:** Form the matrix $A = \begin{bmatrix} V[1] & V[2] & ... & V[t'] \end{bmatrix}$, where $t' \geq t+1$, and compute the kernel of this matrix(i.e. the set of vectors q such that $Aq = 0$). Move to step 5.

- **Step 5:** For each vector $q \neq 0$ in the kernel of A, initialize x to 1 and y_{exp} to a list of 0's of size t'. If an entry of the vector q, say $q_j = 1$, perform $x = (x * X[j]) \mod n$ and $y_{exp} = y_{exp} + V[j]$. In other words, if the j'th entry of the vector q is 1, then we aggregate the pair $(x_j, Q(x_j))$ to the computation of the pair x and y. We have the value of x, and to compute y we perform $y = y * {p_j}^{y_{exp}[j]/2} \mod n$. If any of the vectors in the kernel of A yield a non-trivial solution for x and y, exit the program successfully with the output $d_1 = \gcd(x+y, n)$ and $d_2 = \frac{n}{d_1}$ where $n = d_1 d_2$. Otherwise, increment x and move back to step 2.

5.3.2 Sage Code

```
def Dixon(n,B):
    prime_base = list(primes(1,B))
    prime_base_size = len(prime_base)
    X = []
    Q_x_exponents = []
    V = []
    x_i = int(sqrt(n)) + 1

    while x_i<=n:
        while True:
            vect = [0] * prime_base_size
            Q_x_i = (x_i * x_i) % n
            Q_x_i_exponents = [0] * prime_base_size
            for j in range(0, prime_base_size):
                p_j = prime_base[j]
                while Q_x_i % p_j == 0:
                    Q_x_i = Q_x_i // p_j
                    vect[j] = (vect[j] + 1)%2
                    Q_x_i_exponents[j] = Q_x_i_exponents[j] + 1
                if Q_x_i == 1:
                    X.append(x_i)
                    V.append(vect)
                    Q_x_exponents.append(Q_x_i_exponents)
                    break
            if len(X)>=prime_base_size:
                break
            x_i = x_i + 1

        A = matrix(Zmod(2),V)
        K = A.kernel()
        K_size = len(K)

        for j in range(1,K_size):
            x = 1
            y = 1
            K_vect = K[j]
            y_exponent_vect = [0] * prime_base_size
            for k in range(0,len(X)):
                if K_vect[k] == 1:
                    x_factor = X[k]
                    for l in range(0,prime_base_size):
                        y_exponent_vect[l] = y_exponent_vect[l] + Q_x_exponents
                    x = (x * x_factor)%n
            for l in range(0,prime_base_size):
                y = (y * pow(prime_base[l],y_exponent_vect[l]//2) )%n
        if x != y and x != n-y:
            print x,y,y_exponent_vect
            d_1 = gcd(x+y,n)
            d_2 = n//d_1
            return (d_1,d_2)
        x_i += 1

    return None
```

5.4 Example

Let $n = 248775$. Initialize $X = [], V = []$ and $Q_{exp} = []$.

Let $B = 10$ which gives the prime basis $\{2, 3, 5, 7\}$.

Initialize $x_i = \text{ceil}(\sqrt{n}) = 499$.

For this value of n, it turns out that we need the first 6 B-smooth $Q(x_i)$'s greater than 499 to form a product of $Q(x_i)$'s that yield a perfect square. These values are:

- $Q(500) = (500^2 - 248775) \mod n = 1225 \mod n = (2^0 * 3^0 * 5^2 * 7^2) \mod n$
- $Q(505) = (505^2 - 248775) \mod n = 6250 \mod n = (2^1 * 3^0 * 5^5 * 7^0) \mod n$
- $Q(1000) = (1000^2 - 248775) \mod n = 4900 \mod n = (2^2 * 3^0 * 5^2 * 7^2) \mod n$
- $Q(1010) = (1010^2 - 248775) \mod n = 25000 \mod n = (2^3 * 3^0 * 5^5 * 7^0) \mod n$
- $Q(1230) = (1230^2 - 248775) \mod n = 20250 \mod n = (2^1 * 3^4 * 5^3 * 7^0) \mod n$
- $Q(1500) = (1500^2 - 248775) \mod n = 11025 \mod n = (2^0 * 3^2 * 5^2 * 7^2) \mod n$

From these values of $Q(x_i)$ we compute $V_B(Q(x_i))$ and the exponent vectors(denoted $E_B(Q(x_i))$), and obtain:

- $V_B(Q(500)) = \begin{bmatrix}0\\0\\0\\0\end{bmatrix}, V_B(Q(505)) = \begin{bmatrix}1\\0\\1\\0\end{bmatrix}, V_B(Q(1000)) = \begin{bmatrix}0\\0\\0\\0\end{bmatrix}, V_B(Q(1010)) = \begin{bmatrix}1\\0\\1\\0\end{bmatrix}, V_B(Q(1230)) = \begin{bmatrix}1\\0\\1\\0\end{bmatrix}, V_B(Q(1500)) = \begin{bmatrix}0\\0\\0\\0\end{bmatrix}$

- $E_B(Q(500)) = \begin{bmatrix}0\\0\\2\\2\end{bmatrix}, E_B(Q(505)) = \begin{bmatrix}1\\0\\5\\0\end{bmatrix}, E_B(Q(1000)) = \begin{bmatrix}2\\0\\2\\2\end{bmatrix}, E_B(Q(1010)) = \begin{bmatrix}3\\0\\5\\0\end{bmatrix}, E_B(Q(1230)) = \begin{bmatrix}1\\4\\3\\0\end{bmatrix}, E_B(Q(1500)) = \begin{bmatrix}0\\2\\2\\2\end{bmatrix}$

So, at this stage we have $X = [500, 505, 1000, 1010, 1230, 1500]$, $V = \left[\begin{bmatrix}0\\0\\0\\0\end{bmatrix}, \begin{bmatrix}1\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\0\end{bmatrix}, \begin{bmatrix}1\\0\\1\\0\end{bmatrix}, \begin{bmatrix}1\\0\\1\\0\end{bmatrix},\right.$ and $Q_{exp} = \left[\begin{bmatrix}0\\0\\2\\2\end{bmatrix}, \begin{bmatrix}1\\0\\5\\0\end{bmatrix}, \begin{bmatrix}2\\0\\2\\2\end{bmatrix}, \begin{bmatrix}3\\0\\5\\0\end{bmatrix}, \begin{bmatrix}1\\4\\3\\0\end{bmatrix}, \begin{bmatrix}0\\2\\2\\2\end{bmatrix}\right]$. We take $A = \begin{bmatrix}0&1&0&1&1&0\\0&0&0&0&0&0\\0&1&0&1&1&0\\0&0&0&0&0&0\end{bmatrix}$ and compute the kernel of this matrix, where we find the vector $q = \begin{bmatrix}1\\1\\1\\1\\0\\1\end{bmatrix}$ (note that since the matrix A has 6 rows, it took a while to find this vector q for which the decomposition will work, since we first tried a matrix A with 5 rows for which all the vectors in the kernel yielded trivial pairs (x, y)).

With this information, we can now compute:

- $x = X[1]^{q_1}X[2]^{q_2}X[3]^{q_3}X[4]^{q_4}X[5]^{q_5}X[6]^{q_6} \mod n = 500^1 * 505^1 * 1000^1 * 1010^1 * 1230^0 * 1500^1 \mod n = 382537500000000 \mod n \equiv 201150 \mod n$(in practice we do this multiplication iteratively, by taking $\mod n$ after multiplying two factors to keep the values as small as possible).

- $y_{exp} = q_1 * Q_{exp}[1] + q_2 * Q_{exp}[2] + q_2 * Q_{exp}[2] + q_3 * Q_{exp}[3] + q_4 * Q_{exp}[4] + q_5 * Q_{exp}[5] + q_6 * Q_{exp}[6] = 1 * \begin{bmatrix}0\\0\\2\\2\end{bmatrix} + 1 * \begin{bmatrix}1\\0\\5\\0\end{bmatrix} + 1 * \begin{bmatrix}2\\0\\2\\2\end{bmatrix} + 1 * \begin{bmatrix}3\\0\\5\\0\end{bmatrix} + 0 * \begin{bmatrix}1\\4\\3\\0\end{bmatrix} + 1 * \begin{bmatrix}0\\2\\2\\2\end{bmatrix} =$

$$\begin{bmatrix} 6 \\ 2 \\ 16 \\ 6 \end{bmatrix}$$

- $y = {p_1}^{y_{exp}[1]/2}{p_2}^{y_{exp}[2]/2}{p_3}^{y_{exp}[3]/2}{p_4}^{y_{exp}[4]/2} \mod n = 2^3 * 3^1 * 5^8 * 7^3 \mod n = 643125000 \mod n \equiv 208125 \mod n$(again we do this iteratively taking $\mod n$ at each step).

Finally, we obtain $d_1 = \gcd(x+y, n) = \gcd(409275, 248775) = 8025$ and $d_2 = \frac{n}{d_1} = \frac{248775}{8025} = 31$. This yields the factorization $n = d_1 d_2 = 8025 * 31$.

5.5 Time Complexity and Improvements

Dixon's algorithm consists of two main processes: finding enough B-smooth $Q(x)$'s and solving the linear equation $Aq = 0$ such that we can ascertain which pairs $(x, Q(x))$ to combine in order to obtain perfect squares. Matrix multiplication and linear equation solving are very important areas of research in Computer Science, and improvements in software and hardware implementations of these algorithms allow for a very fast computation of a solution to the equality $Aq = 0$. Using modern time algorithms such as the Coppersmith–Winograd Algorithm or the Strassen Algorithm, as well as algorithms that take advantage of the sparseness of A, we can solve for q in N^2 - N^3 time, where N is the greater dimension of A. Thus, although this part of the algorithm might have been problematic in terms of efficiency earlier in history, it has now become a routine operation that doesn't impact the efficiency of Dixon's algorithm.

However, the problem of finding B-smooth integers remains a tough problem, and although advancements were made, the time it takes to find such numbers in a given integer interval is still exponential with respect to the size of the numbers in the interval.

Let us explore why this problem is computationally hard. The objective is to identify B-smooth numbers in the interval $[1, L]$, where $L = \{\max Q(x) | \text{ceil}(\sqrt{n}) \leq x \leq n\}$. It has been conjectured, but not proven, that if we pick unrelated x's, then whether $Q(x)$ is a B-smooth number can be treated as a random event. If we take this conjecture for granted and observe that B-smooth numbers in $[1, L]$ become less dense as L grows larger, then the computation time is directly proportional to L and to the median value of $Q(x)$'s which we will denote by X. Using Kraitchik's method, X can be estimated by the value $n^{\frac{1}{2}+\epsilon}$, while using Lehmer and Powers method of continued fractions to $\sqrt{n}$, the value of X can be approximated by a smaller value, namely $2\sqrt{n}$. However, using an optimal value for B, the algorithms can be shown to both run in approximately $\exp\left(\sqrt{2\log n \log\log n}\right)$ steps.

But what is the optimal value for B. Let us first remark that there is an interesting dynamic between the choice of B and the time complexity of the algorithm, because, the greater B is, the easier it is to find B-smooth values, but the longer it will take to verify if an integer is B-smooth. Using the method we defined in the previous sections to determine whether a number is B-smooth takes on average $\pi(n)$ steps, where π is the prime counting function. Balancing out the tradeoff between the sparsity of B-smooth integers and the verification time, was an open problem during the 1970s, but it was Dixon's "Asymptotically

Fast Factorization of Integers" paper published in 1981 that offered a proof that the optimal choice for B is $\exp\left(\sqrt{2 \log X \log \log X}\right)$. While this value has suffered modifications since, it has remained in the same range.

While it seems that the nature of the problem imposes the time complexity bounds due to X and that there is not a lot we can do to improve the algorithm by modifying B, we redirect our attention to the B-smoothness identification and verification function. This is the direction in which improvements were made by the sieving functions (which we will cover later on). In its simplest form, the Quadratic Sieve algorithm, proposed by Carl Pomerance in 1981, looks at the values of the function $Q(x) = x^2 - n$ around $\sqrt{n}$ and optimizes the search of B-smooth integers by only factoring those integers that "have a chance" of being B-smooth. To do so, it uses some symmetry properties of the function $Q(x)$ to quickly factor out primes p from only the integers around $\sqrt{n}$ that are divisible by p, thus saving time by ignoring the values that are not divisible by p. Using Quadratic Sieving to compute the factorization of an integer n, the complexity of the computation halves to $\exp\left(\sqrt{\log n \log \log n}\right)$

6 Shor's Algorithm

6.1 History and Context

The algorithm was created by Peter Shor in 1995 and it can be used to quickly factorize large numbers on a quantum computer. If it will ever be implemented, it will have a profound effect on cryptography, as it would compromise the security provided by public key encryption (such as RSA).

In 2001, Shor's algorithm was demonstrated by a group at IBM, who factored 15 into 3×5, using an NMR implementation of a quantum computer with 7 qubits. After IBM's implementation, two independent groups, one at the University of Science and Technology of China, and the other one at the University of Queensland, have implemented Shor's algorithm using photonic qubits, emphasizing that multi-qubit entanglement was observed when running the Shor's algorithm circuits. In 2012, the factorization of 15 was repeated. Also in 2012, the factorization of 21 was achieved, setting the record for the largest number factored with a quantum computer. In April 2012, the factorization of 143 was achieved, although this used adiabatic quantum computation rather than Shor's algorithm. In November 2014, it was discovered that this 2012 adiabatic quantum computation had also factored larger numbers, the largest being 56153. Since April 2016, the largest integer factored on a quantum device is 200099, using D-Wave 2X quantum processor.

6.2 Description and Mathematical Motivation

Imagine two sine waves, one of length P and the other of length Q. Assuming that P and Q are co-prime, then we can rephrase the factorization question as: "At what point does the harmony of P overlapped with Q repeat itself?" We can determine the answer quickly, using quantum computers to figure out the phase of each wave at any given point on the number line. For quantum computers, phase estimation is a fairly simple computation.

If we look at a point N then the phase of P and Q must be 0 if they are factors of N(or else there would be a remainder of the division $\frac{N}{Q}$ or $\frac{N}{P}$). Shor's Algorithm just tests whether the phase of P is equal to Q is equal to 0 at point N.

6.3 Algorithm

A Python implementation of Shor's Algorithm simulating a quantum computer can be found at:

- https://github.com/toddwildey/shors-python/blob/master/shors.py

6.4 Example

An example of Shor's algorithm is provided on page 18:

- http://www.uwyo.edu/moorhouse/slides/talk2.pdf-python/blob/master/shors.py

7 General Number Field Sieve

7.1 History and Context

The General Number Field Sieve is the fastest known algorithm for factoring very large integers(due to its complex set up, other algorithms might be faster for small integers, the crossover being somewhere around 100 digits). Starting from Fermat's difference of squares idea, continuing with Kraitchik's insight that allowed for Dixon's Algorithm to be created, algorithm which was further improved by Pomerance's Quadratic Sieve, the General Number Field Sieve is the current tip of the spear in this series of algorithms. Although the fundamental objective of finding a difference of squares mod the number n to be factored has remained the same, the methods of reaching the end result have expanded from a simple iteration on x to find $x^2 - n = y^2$, to an algorithm that incorporates concepts of algebraic number theory, abstract algebra, linear algebra and real and complex analysis. The outcome of the collaboration of some of the most brilliant mathematical minds of modern times, the GNFS is a beautiful and fascinating concept that currently sets the bar for the size of the integers that can be factored in a feasible amount of time. Upon its discovery in 1996, the GNFS was tested against the Quadratic Sieve to factor a 130-digit RSA number. Although it took the same time for both algorithms to factor the number, the GNFS used 15% less computational power. Today, using advancements in computational power, and small tweaks to the original algorithm, the GNFS was able to successfully factor a 232-digit RSA number. However, until the efficiency of quantum computers drastically improves(the largest integer factored by a quantum computer being the "mighty" 200099), or an amazing insight into the factorization problem(which has not yet been proven to be an NP hard problem) is discovered, our bank accounts can still rely on a large enough RSA key to be resistant to factoring.

Prime bases, smoothness, solving linear equations of exponent vectors, are ideas that we found in the description Dixon's Algorithm and in the Quadratic Sieve algorithm and that also stand at the core of the GNFS. Since it is a sieving algorithm, the GNFS uses the constant spacing of multiples of primes and properties of polynomials to quickly identify smooth values without wasting time performing unnecessary divisions by elements in the factor base. Furthermore, the purpose of finding pairs (x, y) such that $x^2 \equiv y^2 \mod n$ and $x \not\equiv \pm y \mod n$ by aggregating pairs of smooth numbers using a function $Q(x_i)$ to relate x_i's and y_i's remains the same (to be in sync with the literature, we will later change notation from Q to ϕ). However, there are two major insights that enabled the GNFS to come to existence. Before we look at the two ideas, let us first observe that $Q(x)$ takes values from $\mathbb{Z}$ to $\mathbb{Z}/n\mathbb{Z}$ and that it maps perfect squares in $\mathbb{Z}$ to perfect squares in $\mathbb{Z}/n\mathbb{Z}$. Furthermore, notice that we can write $Q(x) = (\phi \circ f)(x)$ where $f(x) = x^2$ and $\phi(x) = x - n$. The first insight is that in the previous methods, we restrict our attention to only using $f(x) = x^2$ to generate perfect squares over $\mathbb{Z}$. This means leaving out almost all other polynomials without exploring the possibility that another $f(x)$ might be better at producing smooth values in $\mathbb{Z}/n\mathbb{Z}$ after ϕ is applied. While the first insight seems a logical extension to the algorithm, the second one is definitely a feat of human imagination. So, assume that we are using a polynomial of some sort, say $f(x)$, to generate perfect squares in $\mathbb{Z}$ and that we use a function ϕ to map these squares to $\mathbb{Z}/n\mathbb{Z}$. The amazing insight is the realization that there is nothing special about $\mathbb{Z}$, and that if we would be able to define a new space were we can impose the notions of factorization, perfect squares and smoothness, and if we could find a map ϕ to take perfect squares in the new space to perfect squares in $\mathbb{Z}/n\mathbb{Z}$, we might find it that we can use the polynomial $f(x)$ more efficiently to generate squares and smooth values(mathematically this means that we have to come up with a field and with a ring homomorphism from the new field to $\mathbb{Z}/n\mathbb{Z}$).

Inspired by an algorithm due to Coppersmith, Odlyzko and Schroeppel that used quadratic number fields to solve the discrete logarithm problem, John Pollard informally proposed in 1988 to a couple of fellow number theorists a method of factoring large numbers using algebraic number fields. However, his method was working for only a couple of "special" large composites that are of the form $p^e \pm r$, where p and r are small. While his method seemed promising, Hendrik Lenstra and Carl Pomerance proving that the algorithm takes $\exp\left(\sqrt{c((\log n)^{1/3}(\log\log n)^{2/3})}\right)$ steps to completion(a significant improvement in the exponent of the leading term), its lack of generality discouraged many mathematicians from pursuing the idea. However, a couple of mathematicians, namely Lenstra and the Manasse brothers, saw more potential in Pollard's idea and implemented the Special Field Number Sieve to factor very large numbers of the special form $p^e \pm r$. This displayed to the scientific community the power of Pollard's idea, but the mathematical roadblocks that prevented this method from being applied to all odd composite integers seemed insurmountable. Nevertheless, due to the work of Carl Pomerance, Joe Buhler, Hendrik Lenstra, Len Adleman and many others, the obstacles began falling one by one, and in the early 1990s, the General Number Field Sieve was born.

7.2 Description and Mathematical Motivation

In this section we will attempt to present the GNFS and the mathematical concepts behind it. We will only present the main ideas of the algorithm taking most of its underlying mathematical concepts for granted.

Let us start by discussing the simpler Quadrating Sieve, an algorithm coined by Carl Pommerance in 1981. This is an improved version of Dixon's algorithm that is quicker than the GNFS at factoring numbers under 100 digits, due to its simplicity. Just like Dixon's algorithm, the method begins by fixing a number B and considering a factor base $P = \{p_1, p_2, ..., p_t\}$ consisting of the prime numbers less than B. We replace the function $Q(x)$ from Dixon's method with a new slightly different one $Q(x) = x^2 - n$. The objective remains the same: first find enough x_i's for which $Q(x_i)$ is smooth over P. Then select a subset x_{ij} such that $\prod_j Q(x_{ij}) = {p_1}^{2e_1}{p_2}^{2e_2}...{p_t}^{2e_t}$. Then, by having $x = \prod_j x_{ij}$ and $y = {p_1}^{e_1}{p_2}^{e_2}...{p_t}^{e_t}$, we can obtain the desired pair of (x, y) for which $x^2 \equiv y^2 \mod n$, and hope that $x \not\equiv \pm y \mod n$, so that we can obtain a factor of n by calculating the $\gcd(x - y, n)$.

The improvement in time complexity of the QS to only half of the one of Dixon's Algorithm, stems from a very clever insight. The way we identify smooth values in Dixon's Algorithm is by plugging in random or incremental values for the x_i, and then test whether $Q(x_i)$ is smooth by dividing $Q(x_i)$ with elements of P until we can hopefully decompose $Q(x_i)$. This is clearly inefficient, as we will often waste time performing divisions (not the cheapest operation for a computer) of $Q(x_i)$'s by $p_j \in P$, even though $p_j \not| Q(x_i)$'s. We can prevent the useless divisions, by turning around the way we think about the problem. Thus, in the QS algorithm we begin by selecting a bound L. Then, for $x_i \in [\sqrt{n} - L, \sqrt{n} + L]$, we store the corresponding $Q(x_i)$ values in memory. Foreach, $p \in P$ we solve the congruence ${x_i}^2 \equiv n \mod p$ with $x_i \in [\sqrt{n} - L, \sqrt{n} + L]$(quick operation for a computer). So, for each prime p_i in P we find an anchor $x_{0_{p_i}}$ in our search space $[\sqrt{n} - L, \sqrt{n} + L]$ at which we know $p_i | Q(x_{0_{p_i}})$ as $Q(x_{0_{p_i}}) = {x_{0_{p_i}}}^2 - n = kp_i + n - n = kp_i$. Using this anchor point we can determine all $x_j \in [\sqrt{n} - L, \sqrt{n} + L]$ for which $p_i | Q(x_j)$. This is possible since $\forall k \in \mathbb{Z} : Q(x_{0_{p_i}} + kp_i) = {x_{0_{p_i}}}^2 + 2x_{0_{p_i}}kp_i + k^2{p_i}^2 - n \equiv {x_{0_{p_i}}}^2 - n \equiv 0 \mod p_i$, which means that all the x_j that yield $p_i | Q(x_j)$ are a multiple of p_i away from the anchor point. Having figured out an anchor point for each $p_i \in P$ we move through the array of $Q(x_j)$'s using p_i incremental steps from the anchor point, and divide the values stored by p_i. Note that we have to divide $Q(x_j)$ by p_i until what is left of $Q(x_j)$ is no longer divisible by p_i. At the end of this process, the array will have the value 1 stored at all the places where $Q(x_j)$'s array stored B-smooth values.

The QS is a neat algorithm that performs very well for reasonably sized integers due to its simplicity. But, can we do better? There are two places where we might look for improvement. The first and more obvious one is at the function $Q(x)$. First of all, the size of the outputs of $Q(x)$ get dangerously large as x moves away from $\sqrt{n}$, and, as we showed in our analysis of Dixon's algorithm, the time complexity increases with the size of $Q(x)$. To address this issue, Peter Montgomery proposed changing the function to $Q_{a,b}(x) = (ax - b)^2 - n$ where $a, b \in \mathbb{Z}, |b| \leq a/2$ and a is a perfect square, and showed that using this new function, $\frac{Q_{a,b}(x)}{a}$ is smooth implies $Q_{a,b}(x)$ being smooth. This function scales the values of $Q_{a,b}(x)$ by a factor of $1/a$ and allows the user to control the "density of the sieve". This solution partially addresses a second problem related to the function $Q(x)$, which is not adapting the function $Q(x)$ to the size of the integer n we want to factor. For very large n's the function $Q(x)$ requires large inputs x which will cause $Q(x)$ to change rapidly. In turn, this will make $Q(x)$'s more unlikely to be smooth, which means that a larger input interval

$[\sqrt{n}-L, \sqrt{n}+L]$ has to be considered. However, by considering a polynomial $f(x)$ of a degree that is related to the size of n and defining $Q(x) = f(x) - n$, it could be possible to bring the output values closer together due to the smaller x's, which wouldn't require the size of the input interval to grow too much. This is possible since the sieving idea works well for any polynomial.

The second improvement is a much more elusive one and builds upon the idea of using a general polynomial $f(x)$ to define $Q(x) = f(x) - n$. By plugging in integer values for x, we can definitely come up with a lot of $f(x)$'s which are smooth over $\mathbb{Z}$, values which we would group together into a perfect square that would then be mapped to a perfect square in $\mathbb{Z}/n\mathbb{Z}$ by $Q(x)$. However, $f(x)$ grows quickly, making the algorithm less efficient. Pollard conjectured that if we can define a new space along with the concepts of factorization, smoothness and factor base, and if we could come up with a function ϕ that maps perfect squares from this new space to perfect squares in $\mathbb{Z}/n\mathbb{Z}$, then we could produce differences of squares in exactly the same way as before. Furthermore, if we could have this new space use the polynomial $f(x)$ to generate more conservative values, we could significantly improve the execution time of the Quadratic Sieve algorithm.

Let us define this new space and the ϕ function. Suppose $f(x)$ is an irreducible monic polynomial over $\mathbb{Z}$ of degree d, and let $m \in \mathbb{Z}$ be such that $f(m) \equiv 0 \mod n$. Furthermore, let $\alpha \in \mathbb{C}$ be a complex root of $f(x)$. We define our new space as the set of complex numbers that can be formed as a linear combination of the elements in $\{1, \alpha, \alpha^2, ...\alpha^{d-1}\}$, which we will denote by $\mathbb{Z}[\alpha]$. Since our objective is still finding $x^2 \equiv y^2 \mod n$ and $x \not\equiv \pm y \mod n$ by aggregating multiple smooth values, we would like factoring, smoothness and perfect squares to work in about the same way in $\mathbb{Z}[\alpha]$ as they do in $\mathbb{Z}$. As it turns out, if we pick as our factoring base a subset of the form $a + \alpha b$ of the set of algebraic numbers (the set of all roots of polynomials with rational coefficients) contained in $\mathbb{Z}[\alpha]$, then all the properties work out nicely over $\mathbb{Z}[\alpha]$. Having these very convenient tools over $\mathbb{Z}[\alpha]$ we can identify smooth values with respect to an algebraic number factor basis $R = \{r_1, r_2, ..., r_q\}$ where $r_i = a_i + \alpha b_i$, where $a_i, b_i \in \mathbb{Z}$, and multiply them to obtain a perfect square in $\mathbb{Z}[\alpha]$, which, as in the case of $\mathbb{Z}$, has to have even exponents for all of the factor base components in its decomposition. Now, let us define the function ϕ. For our plan to work, we want this function to map perfect squares in $\beta \in \mathbb{Z}[\alpha]$ to perfect squares $\mathbb{Z}/n\mathbb{Z}$. Another, more elusive requirement, is for ϕ to be surjective. All of these requirements are fulfilled by a what is called a ring homomorphism $\phi : \mathbb{Z}[\alpha] \to \mathbb{Z}/n\mathbb{Z}$. To see how such a function might help us, suppose we have $\beta \in \mathbb{Z}[\alpha]$. By the property that ϕ maps squares to squares, $\phi(\beta^2) = y^2 \mod n$ for some $y \in \mathbb{Z}/n\mathbb{Z}$ and since ϕ is surjective, $x = \phi(\beta) \mod n$ for some $x \in \mathbb{Z}/n\mathbb{Z}$. Then we can obtain a pair (x, y) that can be used to factor n, as $x^2 \equiv \phi(\beta)^2 \equiv \phi(\beta^2) \equiv y^2 \mod n$, where the second congruence is due to the properties of a ring homomorphism. Enough of abstract algebra! How do we define ϕ in practice? Well, since $f(\alpha) = 0$ in $\mathbb{Z}[\alpha]$ and $f(m) = 0 \mod n$ in $\mathbb{Z}/n\mathbb{Z}$ it would make sense to have $\phi(\alpha) = m$. However, after working out the algebra, it turns out that this restriction on ϕ is enough to create the desired map.

There is one more piece missing from the big picture of the GNFS: the sieving procedure. Let $P = \{p_1, p_2, ..., p_t\}$ be an integer factoring base over $\mathbb{Z}$, and $R = \{r_1, r_2, ..., r_q\}$ where $r_i = a_i + \alpha b_i$, where $a_i, b_i \in \mathbb{Z}$ be an algebraic factoring base over $\mathbb{Z}[\alpha]$. As opposed to the QS algorithm, we are no longer guaranteed to obtain a perfect square on the x's side, since we are not using the polynomial $f(x) = x^2$ over $\mathbb{Z}$ anymore. Thus, to ensure that we obtain perfect squares on both sides of the congruence, we have to apply the sieving procedure both on $\mathbb{Z}[\alpha]$ and $\mathbb{Z}/n\mathbb{Z}$. Furthermore, since a "prime" in $\mathbb{Z}[\alpha]$ is defined as an algebraic number $r = a + \alpha b$, then we have to sieve by varying both a and b. We do so by first solving for anchor points for each element of the factor bases of both P and R. Then we fix b and vary a over an interval $[-L, L]$, where L is defined by the user. Then we increment b which will have values in the interval $[1, M]$, where M is also a value selected by the user, and repeat the iteration of a over $[-L, L]$ until all the values of b are exhausted. As it turns out, we can still use the fact that multiples of a prime have constant spacing between them over $\mathbb{Z}[\alpha]$. After the sieving procedure, we should be left with a set U of integer pairs (a, b) such that $a + b\alpha$ is smooth over R, and $a + bm$ is smooth over P. Our final objective is then to find a subset U' of U such that the both the product of algebraic integers $\prod_{U'} a + b\alpha = \beta^2$ and the product of integers $\mod n \prod_{U'} a + bm = y^2$ are perfect squares. If we can find such a subset U', we can let $\phi(\beta) \equiv x \mod n$ and thus we would be able to obtain a pair (x, y) which we can use to factor n as: $x^2 \equiv \phi(\beta)^2 \equiv \phi(\beta^2) \equiv \phi(\prod_{U'}(a+b\alpha)) \equiv \prod_{U'}(\phi(a+b\alpha)) \equiv \prod_{U'}(\phi(a+bm)) \equiv y^2 \mod n$

7.3 Algorithm and Example

Here is a great source providing pseudocode for the algorithm, as well as a "small" example:

http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.219.2389

An implementation of the algorithm is pretty hard to find online. Here is the link to a C/C++ version:

http://cado-nfs.gforge.inria.fr/

8 Conclusion

Integer factorization is a fascinating problem that hides its complexity and toughness behind the simple mathematical objects it involves. In this paper we investigated some of the most important factorization algorithms due to their present day relevance or due to their historical influence. We were able to identify two main approaches to tackling the problem of factoring integers. The first approach started with Fermat's algorithm and is based on the idea of finding pairs of squares that can be used to factor an integer n. Although simple at its core, this method incorporated more and more mathematical principles and insights, culminating nowadays with the General Number Field Sieve. The second approach to factoring is the interpretation of factors as signals that when in sync can be used to compose another integer. This method is used in Pollard's Rho algorithm and in Shor's algorithm. Nevertheless, in spite of the numerous attempts for a fast solution, the problem of factoring remains unsolvable in polynomial time. However, quantum computers offer great promises with regards to reducing the time needed to factor integers.

9 References

1. Lehman, R. Sherman (1974). ”Factoring Large Integers” Mathematica of Computation
2. ”Fermat’s factorization method” Wikipedia: The Free Encyclopedia
3. Ore, Oystein. ”Euler’s Factorization Method”. Number Theory and Its History. pp. 59–64
4. ”Euler’s factorization method” Wikipedia: The Free Encyclopedia
5. Pollard, J. M. (1975), “A Monte Carlo method for factorization”, BIT Numerical Mathematics 15 (3): 331–334
6. ”Pollard’s rho algorithm” Wikipedia: The Free Encyclopedia
7. http://www.scottaaronson.com/blog/?p=208
8. Bauer, P. Craig ”Secret History: The Story of Cryptology”
9. Dixon, J. D. (1981). ”Asymptotically fast factorization of integers”. Math. Comp. 36 (153): 255–260
10. ”Dixons’s factorization method” Wikipedia: The Free Encyclopedia
11. David Beckman, Amalavoyal N. Chari, Srikrishna Devabhaktuni, and John Preskill “Efficient networks for quantum factoring”
12. ”Pollard’s rho algorithm” Wikipedia: The Free Encyclopedia
13. Pomerance, Carl (December 1996). ”A Tale of Two Sieves”
14. Case, Michael ”A Beginner’s Guide To The General Number Field Sieve”
15. Brigs, E. Matthew ”An Introduction to the General Number Field Sieve”
16. Pomerance, Carl (December 1996). ”The number field sieve”
17. ”General number field sieve” Wikipedia: The Free Encyclopedia