Implementing Two General Purpose Factoring Algorithms

Brendon Go
Stanford University

Abstract

This project examines two general purpose factoring algorithms and implements both. The first algorithm examined is Dixon's factoring method while the second is the Quadratic Sieve Factoring Algorithm.

1 Introduction

Integer factorization is the decomposition of a composite number into a product of smaller numbers. With certain classes of intergers, no efficient integer factoring algorithm is known. The difficulty of integer factorization is at the heart of many cryptographic systems such as RSA. An algorithm that could efficiently factor arbritrary numbers would compromise many cryptographic systems.

2 Background

Fermat's Factorization Algorithm: Many state of the art factoring algorithms are based on Fermat's factorization algorithm which is itself based on congruence of squares. If we know of some x and y such that $x^2 \equiv y^2 \mod N$ for some integer we want to factor, N. Then as long as $x \not\equiv y \mod N$ then we know that $(x-y)(x+y) \equiv 0 \mod N$ and so $gcd(x-y,N)$ is a non-trivial factor of N.
The factoring algorithms we will examine work by more efficiently finding an x and y that statisfy $x^2 \equiv y^2 \mod N$.

3 Dixon's Algorithm

3.1 Algorithm Overview

Dixon's algorithm [1] finds the squares by first finding a number of relations:

$$z^2 = \prod_{p_i \in P} p_i^{a_i} \mod N$$

For some set of prime factors P otherwise known as the factor base.

Once we find a sufficient number of these relations we use linear algebra to find a multiciplative combination of these relations in such a way that both sides are squares. That is we find:

$$z_1^{2b_1} z_2^{2b_2} \ldots z_k^{2b_k} = \prod_{p_i \in P} p_i^{a_{i,1}b_1 + a_{i,2}b_2 + \ldots + a_{i,k}b_k} \mod N$$

such that $a_{i,1}b_1 + a_{i,2}b_2 + \ldots + a_{i,k}b_k \equiv 0 \mod 2$
We can then proceed to find the non trivial factors using Fermat's Factoring Algorithm.

3.2 Implementation Details

3.2.1 Choosing a Bound

The optimal bound for factoring a number n is about $e^{\frac{1}{2}\sqrt{\ln n \ln \ln n}}$. So we set bound to this.

3.2.2 Generating Relations

We find relations by iterating through all the numbers x from $\sqrt{n}$ to n and checking to see if x^2 is b-smooth. Once we find a number that is b-smooth we note the number and the exponents we need to raise each factor base. We will be using these relations to find a linear dependency. For such a dependency to exist we need one more relation than there are factors in the factor base. So once that many relations have been found we can stop.

We check b-smoothness of a number x by using repeated trial division for each prime in the factor base. If after all the repeated division we are left with 1 then it means that the number has all its factors in the factor base and so is b-smooth.

3.2.3 Finding Congruence

It would be best to explain the procedure to find a congruence of squares by going through an example. So suppose we are factoring 16850989. The algorithm obtains a factor base of $[2,3,5,7,11,13,17,19,23,29]$ and the following relations (expressed as a list of z, exponent list pairs):

```
4105, [2, 2, 0, 0, 0, 0, 0, 0, 0, 0]
4113, [2, 0, 1, 0, 1, 1, 0, 0, 1, 0]
4136, [0, 1, 0, 1, 0, 0, 0, 0, 3, 0]
4159, [2, 2, 0, 2, 1, 0, 0, 0, 1, 0]
4168, [0, 6, 1, 0, 1, 1, 0, 0, 0, 0]
4192, [0, 1, 5, 1, 1, 0, 0, 0, 0, 0]
4269, [2, 0, 0, 4, 1, 1, 0, 0, 0, 0]
4558, [0, 1, 4, 1, 0, 1, 0, 0, 1, 0]
4633, [2, 1, 2, 1, 0, 3, 0, 0, 0, 0]
4742, [0, 4, 2, 0, 2, 0, 0, 0, 1, 0]
4817, [2, 1, 4, 1, 2, 0, 0, 0, 0, 0]
```

Note that the first row is already all even but for the sake of example we will ignore the fact that that row has all even exponents. We can express the the exponents as a large matrix:

$$\begin{bmatrix}
2 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 3 & 0 \\
2 & 2 & 0 & 2 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 6 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 5 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 4 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 4 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
2 & 1 & 2 & 1 & 0 & 3 & 0 & 0 & 0 & 0 \\
0 & 4 & 2 & 0 & 2 & 0 & 0 & 0 & 1 & 0 \\
2 & 1 & 4 & 1 & 2 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

Given the relations, we use linear algebra to find a congruence of squares. Our general goal is to find a set of relations so that the sum of each exponent a factor must be raised to is even. Since only parity matters here, we take all the exponent lists and take them mod 2:

$$\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

To find a set of rows that add up to even exponents, we can find a set of rows that add up to one other row (mod 2) and so all these rows combined would have to have each entry be even. This is equivalent to finding which linearly independent rows could combine to a linearly dependent row. To find the linearly indepentent rows (or the row space) of the matrix we can get the reduced row echeolon form of the exponent matrix transposed:

$$\begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 2 & 1 & 2 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 2 & 2 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 3 & 2 & 3 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

The index of a leading 1 tells us the index of a linearly independent row in the exponent matrix. So in our example, the rows at index 1, 2, 3, 4, and 5 (0 indexed) are linearly independent. All other rows are dependent. Now we select a dependent row. Let us say we select the last row. Now we must find a linear combination of the independent rows to get this dependent row. That is we want to solve for v such that:

$$v * \begin{bmatrix}
0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

$$= \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Solving for v using linear algebra we get $\begin{bmatrix} 1 & 1 & 0 & 1 & 0 \end{bmatrix}$. In python this can be done using the numpy linalg library. Some minor details that

needed considering are the fact that matrix solvers assume that the left hand side is a square matrix so we just pad with zeroes. Another consideration is that since there isn't a unique solution and we don't need one, we can use lstsq which gives us an approximate solution and we can just round the numbers to the nearest integer and then we are only concerned with the result modulo 2 since include a row twice gives us the same result parity wise as just not including the row at all. That is we take the mod 2 of the result v.

So that means combining the rows at index 1, 2, and 4 get us the last row. So now we have enough information to find the congruence:

```
4113, [2, 0, 1, 0, 1, 1, 0, 0, 1, 0]
4136, [0, 1, 0, 1, 0, 0, 0, 0, 3, 0]
4168, [0, 6, 1, 0, 1, 1, 0, 0, 0, 0]
4817, [2, 1, 4, 1, 2, 0, 0, 0, 0, 0]
------------------------------------
      [4, 8, 6, 2, 4, 2, 0, 0, 4, 0]
```

So $(4113 * 4136 * 4168 * 4817)^2$ is congruent modulo n to the factor base raised to exponents above which are all even so is a square so we are done.

4 Quadratic Sieve

4.1 Algorithm Overview

The quadratic sieve factoring algorithm [2] builds on Dixon's Algorithm and makes a number of improvements.

First instead of just using x^2 in finding relations, quadratic sieve checks if $f(x)$ is b-smooth for some quadratic function on x. Normally this function will be of the form $(x - \sqrt{(n)})^2 - n$. This keeps $f(x)$ small which increases the chance of it being smooth.

Instead of using trial division to find relations, quadratic sieve uses a sieve. By solving $f(x)$ and properties of the factor base, we can find which indexes into the sieve are divisible by a factor. Once enough candidate relations have been found in the sieve we can then use trial division on a smaller set of numbers to find the exponents for the relations.

The rest of the algorithm follows just as Dixon's algorithm once the relations have been found

4.2 Implementation Details

4.2.1 Factor Base

Instead of simply getting all prime numbers less than a calculated bound like in Dixon's Algorithm, Quadratic Sieve requires that each for each factor in the factor base p, the number we want to factor n must be a quadratic residue modulo p. That is n has a squareroot modulo p.

If p is 2, then any n is a quadratic residue. If p is a factor of n then it is also a quadratic residue. Finally we check if n is a quadratic residue modulo p with Euler's Criterion: $n^{\frac{p-1}{2}} mod p$ is 1 iff n is a quadratic residue modulo p

4.2.2 Generating Relations

First we initialize the sieve to be $f(x)$ for x from 0 to $\sqrt{cn}$ for some constant c. To perform the sieve, we iterate through the primes in the factor base. For each prime, we solve $f(x) = 0 mod p$ to find what the modulo of x is to make the statement true. Suppose the solution is w. We can then know that each element at index $w + cp$ for any c is divisible by p. Since $f(x)$ is quadratic there will be two solutions for p not equal to 2. Once we finish going through the factors, any element in the sieve that has value 1 is necessarily b-smooth.

4.2.3 Finding Congruences

Once the relations have been found, the quadratic sieve factoring algorithm follows Dixon's Algorithm.

5 Code

You can access the open source code here:

`https://github.com/brendongo/factoring`

References

[1] DIXON, J. D. Asymptotically fast factorization of integers. *Mathematics of Computation 36*, 153 (1981), 255–260.

[2] POMERANCE, C. Analysis and comparison of some integer factoring algorithms. *Computational Methods in Number Theory* (1982), 89–141.