Integer Factorisation

Sean Morrell

March 2022

Abstract

Discussion of how common integer factorisation algorithms work, and an analysis of how efficient they are.

Contents

1 Introduction

I wrote this paper as an analysis of my research into different integer factorisation algorithms. I coded these programs to improve my understanding of how they worked, and also to figure out which method is the fastest. All code is available on my GitHub page: https://github.com/Riptide684/IntegerFactorisation.

As of yet, no algorithm exists to factorise an integer in polynomial time on a classical computer. Shor's algorithm runs in polynomial time on a quantum computer with time complexity $O(b^3)$, but is beyond the scope of this paper. RSA cryptography relies on the fact that it is very hard to factor large semiprimes, but it is quick to check whether two primes supplied are indeed the two prime factors. Integer factorisation algorithms first came to my attention while doing a cybersecurity challenge, in which a small semiprime was used as the key, and a fast algorithm such as Pollard's Rho enabled me to factorise it and break the encryption. If quantum computers became stable, the existence of Shor's algorithm would effectively render RSA useless, and another encryption method would have to be used. However, the largest number factored so far was 829 bits (RSA-250) in February 2020, which is significantly smaller than any numbers used in modern RSA encryption.

The algorithms are listed in a rough order of increasing complexity, or at least my perception of how difficult they are to understand and program. There are two kinds of integer factorisation algorithms: special purpose and general purpose. The algorithms described in this paper cover both. Special purpose algorithms have a run time which depends on the structure of the number to be factorised, and particular factors differ between algorithms. General purpose algorithms have a run time which depends entirely on the size of the number to be factorised. When used in practice, general purpose algorithms are preferred because they are much faster, and have an approximate run time, but special purpose algorithms are often used to remove small prime factors quickly.

Each chapter will consist of a breakdown of the algorithm, the mathematical logic behind why they work and a summary of the efficiency of the algorithm.

2 Trial Division

Trial division is the most simple but intuitive way to factorise a number, and is incredibly effective when the prime factors are small. However, for larger prime factors it becomes far slower than any other method, and also requires a very large list of every prime, meaning it's time and space complexity are very large.

The algorithm consists of attempting to divide the number, n by small primes p from some database of primes. If $\frac{n}{p} \in \mathbb{Z}$ then $p|n$, and so we have found a prime factor of n.

3 Pollard's Rho Algorithm

Pollard's Rho algorithm is the most simple yet efficient algorithm that I have come across. The core idea behind the algorithm is to define a pseudo-random number generator from 1 to $n-1$, for example $g(x) = x^2 + 1 \pmod{n}$. We then define two sequences, x and y, which iterate through this sequence at different rates. Since the set of least residues modulo n is finite, we know that these sequences must eventually repeat, which actually occurs much more frequently than anticipated because of the birthday paradox.

Once we have found terms of the sequence such that $x \equiv y \pmod{n}$, then we know $n|abs(x-y)$. If $x \neq y$, then $abs(x-y)$ must have a non trivial factor of n, and so $gcd(abs(x-y), n)$ will be a factor of n.

4 Pollard's $p-1$ Algorithm

Pollard's $p-1$ algorithm is based off of the observation that if $a^k \equiv 1 \pmod{n}$ then $gcd(a^k - 1, n)|n$. In order to do this, we look for k which have $p-1$ as a factor for some $p|n$, which would satisfy the above equation by Fermat's little theorem.

In order to find such a k, we increment it as the product of the first B numbers, because it is likely to have many prime factors, and so increases the chance that $p-1|k$.

In practice, Lenstra elliptic curve factorisation method is much faster than Pollard's $p-1$ algorithm once the prime factors grow large, since the algorithm becomes much slower when B is increased.

5 Lenstra Elliptic Curve Factorisation Method

6 Rational Sieve

This is the first of many algorithms which rely on the concept of congruence of squares. If we can somehow find $x^2 \equiv y^2 \pmod{n}$ where $x \not\equiv y \pmod{n}$ then we can compute two factors of n as follows.

$x^2 - y^2 \equiv (x-y)(x+y) \equiv 0 \pmod{n}$. And so $n|(x-y)(x+y)$, so $(x-y)(x+y)$ must contain all of the factors of n. If we are lucky, then these factors are non-trivial and can be computed as $gcd(x+y, n)$ and $gcd(x-y, n)$. This algorithm (along with many others) hinges on finding such a congruence of squares.

If our goal is to set up a congruence of squares, it would be helpful to know the factors of our numbers, but this is in general very hard. However, if we limit our factors to a small set of primes - called the factor base - then we can factorise these very easily. We define a B-smooth number to be any n such that $n = \prod_{p \in P}^{B} p^{e_i}$, where P is the set of all primes less than B. Generally we choose $B = \lfloor e^{\frac{1}{2}\sqrt{\ln n \ln \ln n}} \rfloor$.

We aim to find a collection of z such that both z and $z+n$ are B-smooth, because $z \equiv z + n \pmod{n}$, and so we can write this as,

$$\prod_{p \in P}^{B} p^{e_i} \equiv \prod_{p \in P}^{B} p^{e_j} \pmod{n}$$

Once we have built up enough of these relations - this is known as the data collection phase - we can use linear algebra to create a congruence of squares. Generally, any number of relations greater than the cardinality of the factor base is sufficient, because a system of $k+1$ equations in k variables must have a linear dependency. We construct the congruence of squares as follows,

1. Store the powers in each of the products as a vector.
2. Since we are only concerned with even powers (because we want a perfect square), write all of the vectors modulo 2.
3. Store each vector as a row in a matrix.
4. Use Gaussian elimination to force one of the rows to be zeroed out.

Once we have a congruence with all even powers, we have our congruence of squares, and we can attempt to compute a factor as described above.

This method is the slowest out of all the ones relying on the congruence of squares. I was unable to find an algorithm to quickly multiply the congruence equations to get all even exponents, and for large n there are very few z which satisfy the conditions above. Much faster methods exist which build on the same principle, and the rational sieve really only serves as a startpoint to understand these more complicated algorithms

7 Dixon's Method

Dixon's method is the first general purpose algorithm listed in this paper, and is much easier to conceptualise than the rational sieve. The overarching goal is the same - to create a congruence of squares - but the way in which we do it is much more efficient.

We start off by building the factor base as described in the rational sieve chapter. We now start at $z = \lceil \sqrt{n} \rceil$ and increment z up to n, recording any cases where both z and $z^2 \pmod{n}$ are B-smooth. As before we store the exponents of z^2 in a vector modulo 2, which form the rows of our matrix.

It is much simpler now to construct a congruence of squares, because the z^2 is already a perfect square, and so we only have to worry about one side of the congruence equations having even exponents. We construct a congruence of squares as follows,

1. Order the rows such that the most significant 1 bits are at the top of the matrix.
2. Add the topmost row to all other rows which have the same most significant 1 bit set.
3. Discard this row. If any of the other rows had this bit set then store the z value of the row we just added.
4. Repeat this process until you run out of rows or one of the rows is the zero vector.
5. If you run out of rows then you need more congruence equations from the data collection phase.

Multiplying the tabulated values of z together yields our congruence of squares, from which we can attempt to factorise n. It is very common at this point to find only trivial factors, and in this case the zero vector row should be discarded, and another congruence of squares should be found by continuing the process described above.

This method has a significant speed up to the rational sieve mentioned before. The idea from this algorithm is the basis for both the much more optimised quadratic sieve and general number field sieve.

8 Quadratic Sieve

The quadratic sieve is almost identical to Dixon's method, except in the way that we find z such that $z^2 \pmod{n}$ is B-smooth. The quadratic sieve uses polynomials and a sieving technique similar to the sieve of Eratosthenes, which is where the algorithm gets its name from.

9 General Number Field Sieve

The general number field sieve is the fastest classical integer factorisation algorithm to date, and has been used to factor many of the largest numbers from the Cunningham project.