On the Quadratic Sieve

Using the best publicly known algorithm, it takes a modern, 2.2 GHz CPU about 1000 years to factor a 795-bit integer of the form $N = pq$, where and p and q are prime numbers.[1] On the other hand, generating such an integer takes merely a few seconds. The asymmetry of these two problems – factoring a large integer that is the product of two primes and generating such an integer – is widely used in modern cryptography, particularly in RSA encryption and signature schemes.

Therefore, it is worthwhile to examine the efficiency, implementation, and theory of algorithms used to factor integers. This paper presents an implementation of the quadratic sieve algorithm – the fastest known method for factoring integers with around 350 bits – with a particular emphasis on the seiving step of the algorithm.

Motivation for the Quadratic Sieve

Suppose that for some integer a, $N + a^2$ is a perfect square. Then it follows that $N = b^2 - a^2$ for some integer b. Since $N = b^2 - a^2 = (b + a)(b - a)$, a non-trivial factorization of N has been found so long as $b - a \neq 1$. This method is known as factoring by squares.

If an oracle which returned integers a and b of this form for any N existed, then factoring large integers would be trivial in terms of computational complexity. Unfortunately, no such oracle exists, and algorithms that directly search for integers a and b of this form are inefficient.

For this reason, it is necessary to relax the assumptions on a and b, and search for them indirectly. More specifically, we indirectly search for values a and b such that

$$b^2 \equiv a^2 \bmod N. \tag{1.1}$$

If equation (1.1) holds, then $kN = (b + a)(b - a)$ and there is a decent chance that $d = \gcd(N, b - a)$ is a non-trivial factor of N.

[1] Boudot pg. 1

Factoring Using the Quadratic Sieve

The indirect method for factoring N by finding values of a and b satisfying equation (1.1) is known as the quadratic sieve. At at high level, the quadratic sieve can be broken down into the following steps, each of which will be discussed at more length in its own section:

(1) Let $B = \lceil (L(N)^{\frac{1}{\sqrt{2}}}) \rceil$ where $L(x) = e^{\sqrt{\ln(x)\ln(\ln(x))}}$.

(2) Find at least $\pi(B) + 1$ integers $\{a_i\}_{i=0}^{n}$ so that $c_i \equiv a_i^2 \pmod{N}$ factors as a product of primes $p < B$.[2]

(3) Take a product $c_{i_1} \dots c_{i_k}$ so that each prime power appearing in the product is even. Then, $c_{i_1} \dots c_{i_k} = b^2$ for some integer b.

(4) Compute a and b by letting $a^2 = (a_{i_1} \dots a_{i_k})^2 \equiv a_{i_1}^2 \dots a_{i_k}^2 \equiv c_{i_1} \dots c_{i_k} = b^2$ and taking the square roots of a^2 and b^2 to recover a and b.

(5) Let $d = \gcd(N, b - a)$. If $d \neq N$ and $d \neq 1$, then return d, a non-trivial factor of N. Else, repeat step (3).

The proceeding process hints at a number of important properties of this algorithm.

Firstly, the algorithm is probabilistic. It is not guaranteed to return a correct factorization of an integer N, although it does so with high probability since the chance of success in step (5) is non-negligible and it can be repeated many times.

Secondly, as the functions in step (1) and (2) indicate, the runtime of this algorithm is somewhat complicated. The general quadratic sieve algorithm is $O(e^{(1+\epsilon)\sqrt{\ln(N)\ln\ln(N)}})$ for some positive ϵ where N is the integer being factored. Thus, the runtime is sub-exponential but super-polynomial in the bit length of an integer N. Therefore, the algorithm is a vast improvement on the naive trial division factorization algorithm, but still impractical for arbitrarily large integers.

Lastly, the computational complexity of the problem arises primarily from finding the values a_i so that $c_i \equiv a_i^2 \pmod{N}$ factors as a product of primes $p < B$ in step (2). This step is known as sieving, and, of all the preceding steps, it will be covered in the most detail due to its technically complex nature. Other steps, such as (3) and (4) can be solved by algorithms that are less conceptually challenging.

[2] Here π denotes the prime counting function.

Choosing a bound B

At first glance, it is somewhat odd to assign to B the value $\lceil L(N)^{1/\sqrt{2}} \rceil$. This value is an optimization of the algorithm. Choosing it is not strictly necessary for the algorithm to run correctly. If B is taken to be a larger value, the algorithm is more likely to return a factor of N, but it takes longer to run. On the other hand, if a smaller value of B is taken the chance of factoring N successfully decrease, but so does the runtime. This value of B happens to be "just right" in the sense that it gives the algorithm a good runtime, while still ensuring that there is a good chance that it factors N successfully.[3] The code for assigning B to this value as in step (1) is trivial.

The Seive

Recall that in step (2) we wish to find many integers a_i such that $c_i \equiv a_i^2 \pmod{N}$ factors as a product of primes $p < B$. Step (2) is fairly complicated and can be considered an algorithm unto itself. It can be broken into a number of sub tasks:

(i) Make a list of primes $p < B$ for which there is a solution to $t^2 = N$ (mod p) – i.e., a list of possible prime divisors of the potential c_i generated in step (2).

(ii) Make a list of values $[F(x), F(x+1), \ldots, F(x+y)]$, where F is the function such that $F(t) = t^2 - N$, $x = \lceil \sqrt{N} \rceil$, and $y = \lceil L(N) \rceil$.

(iii) For each value in step (ii), if it can be factored using primes from step (i) – i.e., primes less than B which divide the value – then return the value.

The first two steps are fairly simple. Step (i) can be solved by generating a list of primes and then checking if a solution to $t^2 = N$ (mod p) exists by seeing if $N^{(p-1)/2} \equiv 1$ (mod p). This works with the exception of when $p = 2$.[4] In this case there is always a solution and the case where $p = 2$ is treated separately. Step (ii) is simply a direct computation.

Step (iii) is more interesting conceptually. Although a brute force algorithm certainly exists, there are *multiple* solutions which are far more elegant.

Notice that if $p \mid F(t)$ then it follows that $p \mid F(t + \alpha p) = (t^2 - N) + 2\alpha p + \alpha^2 p^2$ since $p \mid 2\alpha p + \alpha^2 p^2$. Hence, starting from a solution t to $t^2 \equiv N$ (mod p), every pth value in the list from step (ii) is divisible p. This gives a more efficient way of canceling small prime powers: for each prime p, for each starting solution t, divide every pth value in step (ii) by the largest p^n such that p^n divides the value. At the end of this process, those values which have been reduced to 1 can be expressed as products of small prime powers[5]. In doing so,

[3]The derivation of this optimization is somewhat lengthy, and can be found in Hoffstien pgs. 151-155

[4]If $t^2 = N$ then $N^{(p-1)/2} = t^{p-1} = 1$. If not, notice $x^{(p-1)/2} = 1$ has at most $(p-1)/2$ solutions in $\mathbb{Z}/p\mathbb{Z}$ and there are $(p-1)/2$ quadratic residues, indicating a non residue cannot be a solution

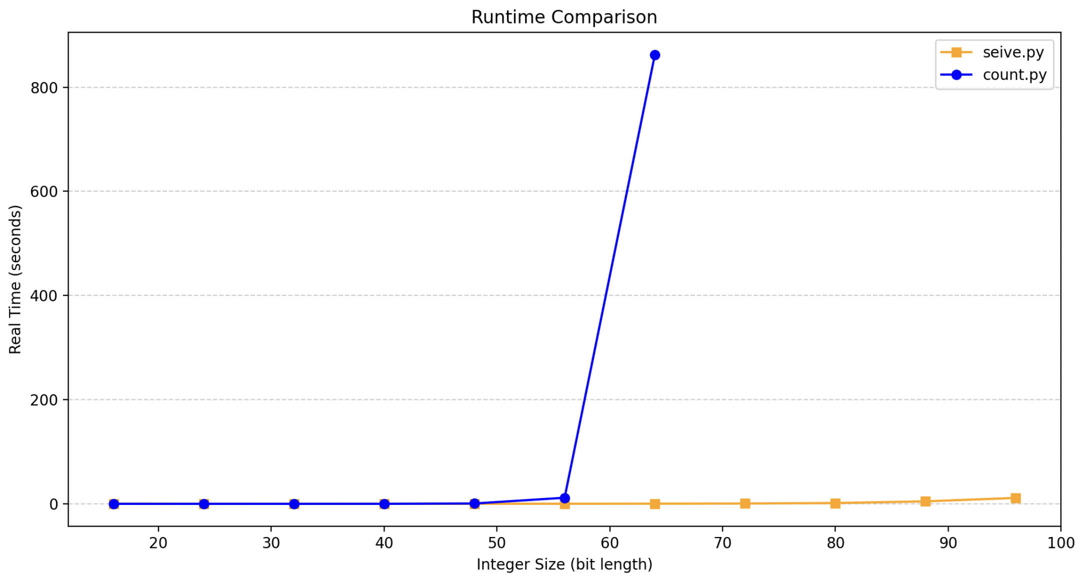
[5]That is, with base primes $p_i < B$

the desired c_i in step (2) have been found. More specifically, the algorithm for step (2) can be implemented in python as follows:

```
# step (2)
def seive(N: int, B: int) -> tuple[list[int], list[int]]:
    # step (i)
    primes = list(primerange(3, B))
    prime_base = []
    for prime in primes:
        if pow(N, ((prime - 1) // 2), prime) == 1:
            prime_base.append(prime)

    # step (ii)
    x = floor(sqrt(N) + 1)
    y = ceil(exp(sqrt(log(N) * log(log(N)))))
    values = []
    for t in range(x, x + y):
        values.append((t * t) - N)
    saved_values = values.copy()

    # handle the case p = 2 separately
    k = values[0] % 2
    for i in range(k, len(values), 2):
        while values[i] % 2 == 0:
            values[i] = values[i] // 2

    # step (iii)
    for p in prime_base:
        roots = mod_sqrt(N, p)
        for root in roots:
            for i in range((root - x) % p, len(values), p):
                while values[i] % p == 0:
                    values[i] = values[i] // p
    c_i_list = []
    for i in range(0, len(values)):
        if values[i] == 1:
            c_i_list.append(saved_values[i])
    return c_i_list
```

Where mod_sqrt is implemented using the Tonelli-Shanks algorithm, and the starting solution's index is set to (root - x) % p in step (iii) since the first element of the values list is $F(x)$. The x's can be thought of as "canceling" mod p leaving the index of $F(x + \text{root} - x\%p) = F(\text{root}\%p)$ as needed.

The runtime for the sieving step compared to a lightweight $O(\sqrt{N})$ algorithm[6] is graphed below



giving a good heuristic for the superior runtime of the sieve in comparison to the $O(\sqrt{N})$ runtime of trial division factoring.

Computing a and b

In step (3) we aim transform the list c_i we obtained in step (2) into many a and b such that $b^2 \equiv a^2 \pmod{N}$.

Since each c_i is the product of small primes $p < B$, it is fairly efficient to find a prime factorization of each of these c_i and record a vector $v_i = (l_1, \dots, l_n) \in \mathbb{F}_2^m$ where l_t is the power modulo 2 of the tth prime in the prime base obtained from step (2).[7] Then, the product $c_{i_1} \dots c_{i_k}$ has an even prime power for every prime if and only if

$$\sum_{j=1}^{k} v_{i_j} = 0 \in \mathbb{F}_2^m.$$

Finding sums of this form is the same problem as solving a system of m equations with n variables.[8] Gaussian reduction and other well known algorithms are well suited for this task. Steps (4) and (5) can then be easily completed using the modular square root algorithm from before and computing a greatest common divisor.

[6]The algorithm simply counts to $O(\sqrt{N})$

[7]If a time for space trade off is desirable, it is possible to do this in step (2) part (iii) with very little overhead by counting the number of times values[i] is divided by p

[8]Which is why $\pi(B) + 1$ elements c_i are desired, so that there are more variables than equations