Math 55: Discrete Mathematics, Fall 2008
Supplementary notes on Algorithms for Factoring

Introduction

The RSA public-key cryptography system relies on the idea that while it is easy to test whether a large number is prime, and therefore easy to construct a number n that is the product of two large primes $n = pq$, there is no way known to efficiently find the factors of such a number, and thus break the encryption scheme, if n is large.

What constitutes "large?" It is quite practical to construct a product $n = pq$ where the primes p and q have 200 digits each, so n is a 400 digit number. It is also practical to perform RSA encryption and decryption with such a large number n. The computations involved take fractions of a second on current computers.

On the other hand, the current record for the largest number not of a special form factored by a general purpose factoring algorithm is 200 digits. This record was achieved by German researchers Bahr, Boehm, Franke, and Kleinjung in 2005 using a cluster of 80 computers running for over a year. Because the running times of the known algorithms are roughly exponential in the square root of the number of digits, factoring a 400 digit number is currently many orders of magnitude more difficult than the record 200 digit number.

In these notes we will discuss *Pollard's algorithm*, a simple factoring algorithm based on the Chinese remainder theorem, which usually takes a number of steps roughly proportional to the square root of the smallest factor to be found. In particular, when the number to be factored is a product $n = pq$ of two nearly equal primes, the number of steps required is typically proportional to the fourth root of n. Pollard's algorithm is thus much better than trial division, which takes $\sqrt{n}$ steps. Using it on an current workstation computer, one can typically factor a number of 20 digits or so in a few seconds. The algorithm is also practical to use for hand computation with the aid of a calculator to factor numbers of about 6 digits.

Pollard's algorithm

Let N denote the number we want to factor.

We can first eliminate the possibility that N is prime using one of the standard primality tests such as Miller's test.

We can also eliminate the case that N is a power of a prime by just checking for each possible power whether $N = p^2$, whether $N = p^3$, and so forth. This need only be done for a small number of possible powers (proportional to the logarithm of N).

After these preliminary steps, we can now assume that N is composite and can be factored as a product of two relatively prime integers, $N = pq$. Without loss of generality we can and will assume that p is the smaller of the two factors.

The basic principle behind Pollard's algorithm (and also behind some of the more sophisticated algorithms) is that when we perform any sequence of arithmetic operations on remainders (mod N), the Chinese Remainder Theorem implies that this "secretly" has the effect of performing the same operations (mod p) and (mod q) separately, even though we do not yet know the factors p and q. Our plan will be to start with a random remainder x_0 (mod N) and from it compute a sequence of additional remainders x_k (mod N) in such a

way that there is a high likelihood that after about $\sqrt{p}$ steps, one of these remainders will, by chance, be divisible by p. At each step we will also compute $\gcd(x_k, N)$. As soon as x_k is divisible by p, then, unless x_k is zero, our gcd computation will discover the factor p of N.

The other key ingredient in Pollard's algorithm is the following. Suppose P is a finite set with p elements and $f\colon P \to P$ is a random function from P to itself. Starting with any element $y_0 \in P$, we can compute a sequence $y_1 = f(y_0)$, $y_2 = f(y_1)$, and so on. Since P is finite, this sequence must eventually repeat, with some $y_j = y_k$. In fact, it can be shown that for a randomly chosen function f, it is highly likely that the first k for which y_k repeats a prior value will be roughly proportional to $\sqrt{p}$ (for now, we will take this fact for granted, but we may justify it later when we discuss probability theory).

An obvious but inefficient way to recognize when the sequence (y_i) repeats is to store all values and compare each new one to the previous ones. However, there is a cleverer way: along with the y_i's, construct another sequence $z_0 = y_0$, $z_1 = f(f(z_0)) = y_2$, $z_2 = f(f(z_1)) = y_4$ that goes 'two hops at a time' through our original sequence y_i. At each step we compare y_k with $z_k = y_{2k}$. This works because once the sequence (y_i) enters a loop, of length m, let's say, we will have $y_k = y_{2k}$ as soon as m divides $2k - k = k$.

In Pollard's algorithm, we assume that the function $f(x) = x^2 + a$, where a is a randomly chosen remainder, is a sufficiently random function that it is likely to repeat after about $\sqrt{p}$ steps (it has not been proven that such a simple function must work well, but in practice people have observed that it seems to).

The algorithm has the following steps.

0. Eliminate the cases where N is prime or a power of a prime by checking for this separately.
1. Choose random remainders a and $y_0 \pmod N$; set $z_0 = y_0$.
2. For $i = 1, 2, \ldots$, compute

$$\begin{aligned} y_i &= f(y_{i-1}); \\ z_i &= f(f(z_{i-1})); \\ x_i &= y_i - z_i; \\ d_i &= \gcd(x_i, N); \end{aligned}$$

where $f(x) = x^2 + a$. If $x_i \not\equiv 0 \pmod N$ and $d_i \neq 1$, then d_i is a non-trivial factor of N, so stop and return d_i. If $x_i \equiv 0 \pmod N$, it means the sequence (y_i) has gone into a loop modulo N without first finding a factor of N [this might happen, although it is unlikely]. In that case, stop and return failure.

Let's notice a few things about the above algorithm. First of all, for clarity, I have labelled the values computed at each step with a subscript i, but at step i the algorithm only needs to use the values y_{i-1} and z_{i-1}. Therefore we need not store all the preceding values, but instead we can recycle a single pair of variables y and z.

Another thing to notice is that the algorithm can fail. If this happens, we can just run it again with new random starting values. It is extremely unlikely to fail repeatedly when run again and again.

A final thing to notice is that we did not set any limit in the algorithm on how many steps i to take. It can't run forever, since the y_i's must eventually repeat modulo N, but at worst, it might run for N steps (and then fail, besides). A desirable improvement over the algorithm described above would be to set a limit in advance of something like $100\sqrt[4]{n}$ on the number of steps to be computed. Most of the time, the algorithm should find a factor before reaching this limit, but if it hits the limit we would be better off having it declare failure, so we could run it again with new starting values that might find a factor more quickly.

Example

To make clear how Pollard's algorithm works, we'll now proceed through it step by step to factor the integer $N = 4020649$.

Instead of choosing a at random, we'll take $a = 2$ for simplicity. Then our function f is

$$f(x) = x^2 + 2 \pmod{N}$$

Now we choose our random starting value

$$y_0 = z_0 = 3878711.$$

Next we compute $y_1 = y_0^2 + 2 \pmod{N}$, and $z_1 = (z_0^2 + 2)^2 + 2 \pmod{N}$ to be

$$y_1 = 2944356, \quad z_1 = 1355865.$$

The difference is $x_1 = 1588491$, and using the Euclidean algorithm we find $\gcd(x_1, N) = 1$. Therefore we continue to the next step. Here is a table of the values of y_i, z_i and $\gcd(y_i - z_i, N)$ found at the first 9 steps.

i	y_i	z_i	$\gcd(y_i - z_i, N)$
1	2944356	1355865	1
2	1355865	1571461	1
3	514659	3330800	1
4	1571461	2994483	1
5	3038074	1779175	1
6	3330800	3675486	1
7	3606514	3990798	1
8	2994483	199488	1
9	645160	3247459	1493

Observe how the sequence of z_i values is the same as that of the y_i values, skipping every other step.

The algorithm terminates with the discovery of the factor 1493. Dividing it into N, we find that

$$4020649 = 1493 \cdot 2693.$$

Both these factors happen to be prime, so we have completely factored our number N.

For the sake of illustration, I chose an example in which the algorithm takes fewer steps than would be typical for factoring a 7-digit number. Nevertheless, the chosen example is

not all that misleading. I discovered it simply by testing the algorithm (with the help of a computer) on several products of two random primes less than 3000, until I found one that took relatively few steps to factor. I only had to test a handful of examples before I found this one, and all the ones I tested factored within 40 steps or less.