Factoring Algorithms

Beyond trial division, there do exist some factoring algorithms that tend to work faster than trial division. Many of these are extremely complicated, but two of the more simple ideas are covered in this lecture.

Fermat Factoring

The Fermat Factoring Algorithm is hinged upon the following factoring fact:

$x^2 - y^2 = (x + y)(x - y)$.

Since we are assuming that our goal is to factor an integer that is the product of two large odd (prime) numbers, we know that the difference between the two factors will be even. Since this is the case, there MUST exist and x and y that satisfy the following equations:

$x + y = p$
$x - y = q$, $N = pq$, where N is the number of factor, and p and q are both large primes.

Namely, x is the "halfway" point in between p and q, and y is the distance from this halfway point to both p and q.

So the idea is as follows: Imagine you are trying to factor N = 26441.

The first thing we know is that x > sqrt(26441), since we must solve

$N = 26441 = x^2 - y^2 = (x + y)(x - y)$, and y^2 is positive.

sqrt(26441) is 162.6. Thus, our first possible value for x is 163. Try it out:

$26441 = 163^2 - y^2 \quad \rightarrow y^2 = 163^2 - 26441 = 128$ (Not a perfect square)

Now, keep on trying successive values of x:

$164^2 - 26441 = 455$ (Not a perfect square)
$165^2 - 26441 = 784$ (This is 28^2, so we are done. x = 165, y = 28, p = 193, q = 137.)

Thus, we find 26441 = 193 x 137.

This algorithm will ALWAYS succeed, but sometimes, its run time will be prohibitive. In particular, the algorithm does well when the two factors are very close to each other in magnitude. It does poorly if they are far apart. Keep in mind that two twenty digit numbers, such as 10^{19} and $9x10^{19}$ are very far apart. (The difference between the two is $8x10^{19}$.)

Pollard-Rho Factoring

The Pollard-Rho Factoring Algorithm works based on the notion of order previously described. But, unlike the Fermat algorithm, it may not succeed in all instances. The idea is to create a sequence of numbers that will eventually cycle, when considered mod n. If it cycles mod n, it must also cycle mod p, (where p is one of n's two distinct prime divisors.) Also, it is likely that the cycle mod p has a shorter length than the cycle mod n. If we can detect the cycle mod p (even though we don't know p), perhaps we can utilize that information to get p. Here's the algorithm:

1. let a = 2, b = 2.
2. while (true) {
    a = a^2 + 1 mod n
    b = b^2 + 1 mod n
    b = b^2 + 1 mod n
    d = GCD(a – b, n)
     if (1 < d && d < n) return d;
     if (d == n) return failed;
  }

The basic idea is that both a and b are taken from the following sequence of numbers: 5, 26, 677, ... In each iteration, a is the first number, then the second number, then the third number, etc and b is the second number, then the fourth numberm then the sixth number, etc. When we calculate a – b, we are calculating the difference between successive terms, then the second and fourth term, then the third and sixth term, etc. Essentially we are trying different cycle lengths in the sequence, it the very worst case, the sequence will cycle every n-1 values (which is an exceedingly long time), but in most instances it should cycle much more quickly. If the cycle for p is shorter, then the algorithm will work.