Pollard's Rho Method

Introduction

Suppose you had some large number that you knew was not a prime number and you needed to find out what its factors are. How would you go about doing that? You could try dividing it by ***2***, ***3***, ***4***, ***5***, etc. until you find one that works. You could even optimize that a bit if you by only trying to divide by prime numbers. But, if the smallest divisor of your number is pretty large, you've got a great deal of work ahead of you.

John Pollard published his Rho Method of Factorization in 1974. It takes advantage of some properties of divisors of numbers to zoom in on a factor quite quickly. Once you have one factor, you can readily break the original number into two smaller numbers to factor.

This is a brief description of the Pollard's Rho Algorithm. If you're already familiar with modulo arithmetic and greatest common divisors, you can skip ahead to the actual algorithm.

Modulo Arithmetic

If you're already comfortable with addition, subtraction, multiplication, and exponentiation modulo a number, feel free to skip over this whole section.

Definition Modulo

Two numbers ***x*** and ***y*** are defined to be **congruent modulo *n*** if their difference ***(x-y)*** is an integer multiple of ***n***. Another way of stating this is to say that ***x*** and ***y*** both have the same remainder when divided by ***n***.

For example, suppose that $x = q_x * n + r_x$ where $0 <= r_x < n$. And, suppose that $y = q_y * n + r_y$ where $0 <= r_y < n$. Then, ***x*** is congruent to ***y*** modulo ***n*** if and only if $r_x = r_y$. You can see that if $r_x = r_y$, then ***(x-y)*** is just

$$q_x * n - q_y * n + r_x - r_y = (q_x - q_y) * n$$

For a concrete example, suppose that $x = 37$, $y = -14$ and $n = 17$. ***x*** is congruent to ***y*** modulo ***n***. We know this because

$$(x-y) = 37 + 14 = 51 = 3*17$$

We could also tell this because $x = 2*17 + 3$ and $y = (-1)*17 + 3$—

both have a remainder of ***3*** when divided by ***17***. By the same logic, it is also easy to see that both ***x*** and ***y*** are congruent to ***3*** since ***3 = 0*17 + 3***.

Modulo Operations

We often speak of doing some operation (addition, subtraction, multiplication, etc.) "modulo ***n***". This simply means, do the operation and then rewrite the answer to be the number that's at least ***0***, is less than ***n***, and is congruent modulo ***n*** with the answer.

For example, ***37 + 15 = 1*** modulo ***n***. This is often abbreviated like this:

$$37 + 15 = 1 \pmod{n}$$

Conveniently, one can take any number in a modulo equation and replace it with any number which is congruent to it. It is usually convenient to pick the smallest positive number which foots the bill. So, we could redo the equation ***37 + 15 = 1*** without having to add those huge numbers ***37*** and ***15*** or to divide that huge sum of ***52*** by ***17***. Instead, we could replace the ***37*** with ***3*** because they are congruent with each other modulo ***17***. So,

$$37 + 15 = 3 + 15 = 18 = 1 \pmod{n}$$

The same thing holds for subtraction and for multiplication and for exponentiation. So, it is easy to see that $37^4 = 13$ modulo ***17***. We simply replace the ***37*** by ***3***. Then, we break up the exponentiation a bit.

$$37^4 = 3^4 = (3^3)*3 = 27 * 3 = 10 * 3 = 30 = 13$$

because ***27 = 10*** and ***30 = 13*** modulo ***17***.

Greatest Common Divisor (Euclidean Algorithm)

For Pollard's Rho Method, one needs to be able to find the Greatest Common Divisor of two numbers. The **Greatest Common Divisor** is the largest number which divides evenly into each of the original numbers. The Greatest Common Divisor of ***a*** and ***b*** is often written "gcd(***a***,***b***)". Sometimes, you will see it written simply as "(***a***,***b***)".

The Greatest Common Divisor is symmetric. This is

$$\gcd(a,b) = \gcd(b,a)$$

The usual method for finding the Greatest Common Divisor is the Euclidean

Algorithm. The **Euclidean Algorithm** goes like this.... Start with the numbers ***a*** and ***b***. Express ***a*** as a multiple of ***b*** plus a remainder ***r*** which is greater than or equal to zero and less than ***b***. If ***r*** is greater than zero, set ***a*** equal to ***b*** and set ***b*** equal to ***r***. Lather. Rinse. Repeat.

As you can see, ***r*** decreases every iteration until it reaches zero. On the first pass, it cannot be as big as ***b***. On the second pass, it cannot be as big as it was on the first pass. On the ***n***-th pass, it cannot be as big as it was on the previous pass. Eventually, ***r*** has to get to zero. When it does, then ***b*** (which was the previous ***r***) is the Greatest Common Divisor of the original ***a*** and ***b***. [Actually, if the original ***b*** is some multiple of the original ***a***, then the first ***r*** will be zero. In that case, the Greatest Common Divisor will actually be ***a*** instead of ***b***. You can avoid this problem by always starting with ***a*** being the number which has the highest absolute value.]

For example, let us find the Greatest Common Divisor of ***658*** and ***154***. This leads to the following sequence of equations.

$$658 = 4 * 154 + 42$$

$$154 = 3 * 42 + 28$$

$$42 = 1 * 28 + 14$$

$$28 = 2 * 14 + 0$$

Which means that ***14*** is the greatest common divisor of ***658*** and ***154***.

You can see that ***14*** divides evenly into ***154*** and ***168*** by propagating back up the that list of equations. The last equation shows that ***14*** divides into ***28***. The equation before that shows that ***42*** is some multiple of something which is a multiple of ***14*** with an extra ***14*** tacked on the end. This means that ***42*** is a multiple of ***14*** since it is the sum of two things which are multiples of ***14***. The equation above that shows that ***154*** is the sum of a multiple of ***42*** and ***28***. Both ***42*** and ***28*** are multiples of ***14***, so ***154*** is also a multiple of ***14***. And, the last equation, by similar logic, shows that ***658*** is divisible by ***14***.

Unfortunately, in the preceding paragraph, we only managed to show that ***14*** was a common divisor of ***658*** and ***154***. We didn't show that it was necessarily the largest divisor common to both. That part is more complicated. At the time of writing here, I don't feel like getting into that. You can find ample documentation of the Euclidean Algorithm in text books and on the web if you're interested in that part of the proof.

Pollard's Rho Method

Now, on to the actual algorithm.

The Algorithm

Say, for example, that you have a big number ***n*** and you want to know the factors of ***n***. Let's use ***16843009***. And, say, for example, that we know that ***n*** is a not a prime number. In this case, I know it isn't because I multiplied two prime numbers together to make ***n***. (For the crypto weenies out there, you know that there a lot of numbers lying around which were made by multiplying two prime numbers together. And, you probably wouldn't mind finding the factors of some of them.) In cases where you don't know, a priori, that the number is composite, there are a variety of methods to test for compositeness.

Let's assume that ***n*** has a factor ***d***. Since we know ***n*** is composite, we know that there must be one. We just don't know what its value happens to be. But, there are some things that we do know about ***d***. First of all, ***d*** is smaller than ***n***. In fact, there are at least some ***d*** which are no bigger than the square root of ***n***.

So, how does this help? If you start picking numbers at random (keeping your numbers greater or equal to zero and strictly less than ***n***), then the only time you will get $a = b$ modulo ***n*** is when ***a*** and ***b*** are identical. However, since ***d*** is smaller than ***n***, there is a good chance that $a = b$ modulo ***d*** sometimes when ***a*** and ***b*** are not identical.

Well, if $a = b \pmod d$, that means that $(a-b)$ is a multiple of ***d***. Since ***n*** is also a multiple of ***d***, the greatest common divisor of $(a-b)$ and ***n*** is a positive, integer multiple of ***d***. So, now, we can divide ***n*** by whatever this greatest common divisor turned out to be. In doing so, we have broken down ***n*** into two factors. If we suspect that the factors may be composite, we can continue trying to break them down further by doing the algorithm again on each half.

The amazing thing here is that through all of this, we just knew there had to be some divisor of ***n***. We were able to use properies of that divisor to our advantage ***before*** we even knew what the divisor was!

This is at the heart of Pollard's rho method. Pick a random number ***a***. Pick another random number ***b***. See if the greatest common divisor of $(a-b)$ and ***n*** is greater than one. If not, pick another random number ***c***. Now, check the greatest common divisor of $(c-b)$ and ***n***. If that is not greater than one, check the greatest common divisor of $(c-a)$ and ***n***. If that doesn't work, pick another random number ***d***. Check $(d-c)$, $(d-b)$, and $(d-a)$. Continue in this way until you find a factor.

As you can see from the above paragraph, this could get quite cumbersome quite quickly. By the ***k***-th iteration, you will have to do $(k-$

1) greatest common divisor checks. Fortunately, there is way around that. By structuring the way in which you pick "random" numbers, you can avoid this buildup.

Let's say we have some polynomial $f(x)$ that we can use to pick "random" numbers. Because we're only concerned with numbers from zero up to (but not including) n, we will take all of the values of $f(x)$ modulo n. We start with some x_1. We then choose to pick our random numbers by $x_{k+1} = f(x_k)$.

Now, say for example we get to some point k where $x_k = x_j$ modulo d with $k < j$. Then, because of the way that modulo arithmetic works, $f(x_k)$ will be congruent to $f(x_j)$ modulo d. So, once we hit upon x_k and x_j, then each element in the sequence starting with x_k will be congruent modulo d to the corresponding element in the sequence starting at x_j. Thus, once the sequence gets to x_k it has looped back upon itself to match up with x_j (when considering them modulo d).

This looping is what gives the Rho method its name. If you go back through (once you determine d) and look at the sequence of random numbers that you used (looking at them modulo d), you will see that they start off just going along by themselves for a bit. Then, they start to come back upon themselves. They don't typically loop the whole way back to the first number of your sequence. So, they have a bit of a tail and a loop---just like the greek letter "rho".

Before we see why that looping helps, we will first speak to why it has to happen. When we consider a number modulo d, we are only considering the numbers greater than or equal to zero and strictly less than d. This is a very finite set of numbers. Your random sequence cannot possibly go on for more than d numbers without having some number repeat modulo d. And, if the function $f(x)$ is well-chosen, you can probably loop back a great deal sooner.

The looping helps because it means that we can get away without piling up the number of greatest common divisor steps we need to perform. In fact, it makes it so that we only need to do one greatest common divisor check for every second random number that we pick.

Now, why is that? Let's assume that the loop is of length t and starts at the j-th random number. Say that we are on the k-th element of our random sequence. Furthermore, say that k is greater than or equal to j and t divides k. Because k is greater than j we know it is inside the looping part of the Rho. We also know that if t divides k, then t also divides $2*k$. What this means is that x_{2*k} and x_k will be congruent

modulo ***d*** because they correspond to the same point on the loop. Because they are congruent modulo ***d***, their difference is a multiple of ***d***. So, if we check the greatest common divisor of $(x_k - x_{k/2})$ with ***n*** every time we get to an even ***k***, we will find some factor of ***n*** without having to do ***k-1*** greatest common divisor calculations every time we come up with a new random number. Instead, we only have to do one greatest common divisor calculation for every second random number.

The only open question is what to use for a polynomial ***f(x)*** to get some random numbers which don't have too many choices modulo ***d***. Since we don't usually know much about ***d***, we really can't tailor the polynomial too much. A typical choice of polynomial is

$$f(x) = x^2 + a$$

where ***a*** is some constant which isn't congruent to ***0*** or ***-2*** modulo ***n***. If you don't place those restrictions on ***a***, then you will end up degenerating into the sequence ***{ 1, 1, 1, 1, ... }*** or ***{ -1, -1, -1, -1, ... }*** as soon as you hit upon some ***x*** which is congruent to either ***1*** or ***-1*** modulo ***n***.

Examples

Let's use the algorithm now to factor our number ***16843009***. We will use the sequence $x_1 = 1$ with $x_{n+1} = 1024 * x_n^2 + 32767$ (where the function is calculated modulo ***n***). [I also tried it with the very basic polynomial $f(x) = x^2 + 1$, but that one went 80 rounds before stopping so I didn't include the table here.]

k	x_k	gcd$(n, x_k - x_{k/2})$
1	1	
2	33791	1
3	10832340	
4	12473782	1
5	4239855	
6	309274	0
7	11965503	
8	15903688	1
9	3345998	
10	2476108	0
11	11948879	
12	9350010	1

13	4540646	
14	858249	0
15	14246641	
16	4073290	0
17	4451768	
18	14770419	257

Let's try to factor again with a different random number schema. We will use the sequence $x_1=1$ with $x_{n+1} = 2048*x_n^2 + 32767$ (where the function is calculated modulo n).

k	x_k	gcd$(n, x_k-x_{k/2})$
1	1	
2	34815	1
3	9016138	
4	4752700	1
5	1678844	
6	14535213	257

There is an art to picking the polynomial. I don't know that art at all. I tried a couple of polynomials until I found one that zoomed in relatively quickly. If I had to factor something with this method, I would generate a few small polynomials at random and try them out in parallel until one of them found a factor or I got tired of waiting.