Math 189H Joy of Numbers Activity Log

Tuesday, November 15, 2011

Jason Love: "The microwave oven is the consolation prize in our struggle to understand physics."

John von Neumann: "In mathematics you don't understand things. You just get used to them."

Today we continued to talk about the mechanics of encrypting and decrypting messages, and how hard it might be to decrypt a message without prior knowledge of the decryption process. Last time we saw shift ciphers, which are relatively insecure; trying every single possible shift (since to decrypt the shift $x \mapsto x+a$ mod n we just shift back, $y \mapsto y-a$ mod n) would not take very long and would eventually recover the correct value of a (instantly enabling you to encrypt your own messages, as well!). The problem is that the amount of work required to test all possible decryption procedures (assuming that you know the basic structure of the process) is not high enough; a 'brute force' attack will succeed in a reasonable amount of time.

To combat this, we can build encryption mechanisms involve a larger 'space' of built-in choices. For example, we can move from just adding a constant $x \mapsto x+a$ to a more involved function, like an *affine* function $x \mapsto ax+b$ mod n [what you would probably call a linear function?]. This is called an *affine shift cipher*, and gives us greater flexibility in choosing parameters, increasing the number of possible decryption procedures. But in order to have a decryption procedure, we need to know that we can 'undo' the affine shift; that is, recover x from $ax+b$. For this we can turn to a little algebra: we learn that if $y = ax+b$, then we can figure out how to write x in terms of y by 'inverting' the equation. $ax = y-b$, so $x = (1/a)(y-b)$ allowing us to recover x from y, i.e., decrypt the encrypted message y. Except for one little thing; we are working modulo n, and division might not make sense? What is $1/a$, anyway? It is the number c so that $ca = 1$. What we really what to do is to find c so that $ca \underset{n}{\equiv} 1$. And not every a has such an inverse modulo n. As we saw, $ca \underset{n}{\equiv} 1$ means that $n|ca-1$, so $ca-1 = nk$ for some integer k, and so $1 = ca-nk$. This took us all the way back to our first explorations of common factors; that fact that 1 can be expressed as a combination of a and n means that a and n are relatively prime. And if they <u>are</u> relatively prime, then we can find such a c, using the Euclidean algorithm! So, for example, since 7 and 47 are relatively prime, if we encrypt using the affine shift cipher $x \mapsto 7x+24$ mod 47, then we can build the decryption function by inverting 7 modulo 47, using the Euclidean algorithm:

$47 = 6\cdot 7+5$, $7 = 1\cdot 5+2$, $5 = 2\cdot 2+1$, so

$1 = 5-2\cdot 2 = 5-2\cdot(7-1\cdot 5) = 3\cdot 5-2\cdot 7 = 3(47-6\cdot 7)-2\cdot 7 = 3\cdot 47-20\cdot 7$, so

$20\cdot 7 = 3\cdot 47-1$, so $(-20)\cdot 7 = 1-3\cdot 47$ and mutiplication by -20 (hm, not by 20 as asserted in class!?) will 'undo' mutiplication by 7, modulo 47. If you don't like negative numbers, add another 47, since this won't change anything mod 47, to get $c = 27$. So $27\cdot 7 \equiv 1$ mod 47, and so $y \mapsto 27(y-24)$ mod 47 is the decription function for $x \mapsto 7x+24$ mod 47. It is worth noting that the decryption is another affine map!

$27(y-24) = 27y - 27\cdot 24 \equiv 27y + 20\cdot 24 = 27y + 480 \equiv 27y + 10 \bmod 47$, so if $e(x) = 7x + 24 \bmod 47$, then $d(y) = 27y + 10 \bmod 47$.

How many affine shift functions are there? Since we need $\gcd(a,n) = 1$ in order to have a decryption function, there are $\phi(n)$ choices for a, and n choices for b (although we agreed that $a = 1$ and $b = 0$, so $e(x) = 1\cdot x + 0 = x$ was probably a bad choice of encryption!), so there are $n\phi(n)-1$ choices of affine cipher, and so the same number of choices of decryption function. For $n = 26 = 2\cdot 13$, $\phi(n) = (2-1)(13-1) = 12$, giving us $12\cdot 26 - 1 = 285$ possible shift ciphers for a 26-letter alphabet. Better th an 25, for shift ciphers, but still not very great!

In general, what we require out of a cipher that encrypts one letter/character at a time, like the shifts and the affine shifts, is really a function, which you can picture as an input/output table, which is a column of the symbols to encrypt together with the values each symbol will be assigned listed to the right of it. Encryption really just reads the table from left to right, and decryption reads it from right to left! [Although we usually reorder the list so that the things on the right come in some sensible order, to make decryption faster.] In the above we were using an affine map mudulo n to determine the values to fill in to the right of our symbols, but really, any function will do just as well. Or almost any function, anyway. We need to know that we can decrypt the message, which really means that we cannot assign the same value to two different symbols we want to encrypt. That means that the function from symbols to encrypt to symbols used in the encrypted message (usually the same set of symbols) cannot take the same value twice, meaning that if we treat the symbols used in encryption as buckets, there is no more than one symbol in each bucket. By the pigeonhole principle, this would mean that every bucket has exactly one symbol in each bucket. A map from {symbols} $\rightarrow$ {symbols} that is like this - every symbol is the image of exactly one symbol under the function - is called a *permutation*. There are precisely $n!$ permutations of n symbols. [Why? Induction!] If we use permutations of the 26 letters as our encryption, then decryption is also (the inverse) permutation; so a brute force attack to recover a message by trying all possible permutations will (using the notion that, on 'average', you will have to try half of the possiblilties before you expect to hit upon the right one) requires, typically, $26!/2 = 201,645,730,563,302,817,792,000,000 \approx 2\cdot 10^{2}6$ attempts. Which is a lot! Unfortunately, since under a scheme like this every letter would get encrypted to the exact same letter every single time it appeared, such a scheme is vulnerable to 'frequency analysis' attacks (which we will discuss later if time permits).

At this point it was asked if we could make things more secure by stringing together several encryption functions. The answer was, sort of, yes and no. Combining two affine shifts, really only just gives us an affine shift: $a_1(a_2x + b_2) + b_1 = (a_1a_2)x + (a_1b_2 + b_1)$; the composition of two linear functions is a linear function. And if you string together two permutations, you get another permutation. If you have two affine ciphers that use <u>different</u> moduli, you can improve things; the result is (I think!) not an affine function anymore. And combining together several different encryption methods into one big one <u>is</u> a very common method for building a cryptosystem (as an encryption/decryption procedure is often called), especially if the component systems have very different characteristics; combining them is often thought of as taking advantage of each systems strengths (or rather, compensating for one system's weakness by 'hiding' it under another's strength).

Another more important worry, common with all 'permutation ciphers' (and many others), is that knowing the encryption function tells you precisely how to decrypt messages, as well: to encrypt you need the table, to read left to right, but to decrypt you only need to read the table backwards, from right left, instead. So everybody who has a knowledge of how to encrypt messages to send to you holds the information on how to decrypt messages, and must therefore keep that information secret and protected, otherwise your encryption method is not secure. This was for a long time viewed as a major flaw to such encryption schemes. It was not until 1978, however, that Rivest, Shamir, and Adelman [although read the Wikipedia entry of the RSA algorithm; it was reported that a British intelligence analyst, Clifford Cocks, had found a roughly equivalent procedure in 1973] reported an encryption/decryption scheme for which, essentially, complete knowledge of how a message was to be encrypted did not reveal how to build the decryption function. That is, you can tell the entire world how you are going to encrypt your messages, and your decryption method (which you alone know) will remain secure. And it is all based on Euler's Theorem!

The basic idea is that since, if $\gcd(a,n)=1$ we know that $a^{\phi(n)} \underset{n}{\equiv} 1$, we can [as some of you discovered and used on the most recent exam!] carry out computations of exponents a^k mod n by modifying k. The basic idea, for us, is that since $a^1 \underset{n}{\equiv} a$, $a^{\phi(n)+1} = a^{\phi(n)}a^1 \equiv 1\cdot a^1 = a$, as well. And we can continue to add multiple of $\phi(n)$ to the exponent without changing things, as well: $a^{k\phi(n)+1} \underset{n}{\equiv} a$ for every (well, OK, positive) integer k.

What Rivest, Shamir, and Adelman did was to say, 'Well, what if we write $k\phi(n)+1$ as a product, $k\phi(n)+1 = ed$. Then $a \underset{n}{\equiv} a^{k\phi(n)+1} = a^{ed} = (a^e)^d = b^d$ (for $b \underset{n}{\equiv} a^e$), so $b^d \underset{n}{\equiv} a$ when $b \underset{n}{\equiv} a^e$. The idea is that we can then 'hide' a, by taking a power of it, mod n, as $b = a^e$ mod n, and then <u>recover</u> a, from the 'encrypted' number b as $a \underset{n}{\equiv} b^d$. That is, raising to the power d, mod n, is the decryption function for raising to the power e, mod n. [That is true, at least, when the 'message', a, is relatively prime to n (oh!, and smaller than n (!)).]

And what does it take to do all of this? To create the pair e, d, we need to know $\phi(n)$. To encrypt, you need to know e and n. And to decrypt, you need to know d and n. The question is, does the person who knows how to encrypt have the information to build the decryption function? That is, knowing e and n, can you recover d? As we shall see, since $\phi(n)$ is used to build e and d (or, as we look a little deeper, to build d from e), the question boils down to, can someone who knows n determine the value of $\phi(n)$? To which the answer, of course, is 'YES!', because we gave a formula for $\phi(n)$ (!). The fun part comes when we look deeper at what we used to <u>compute</u> $\phi(n)$....