CS 2800: Discrete Math Sept. 26, 2011

Lecture

Lecturer: John Hopcroft *Scribe: June Andrews*

Review

See previous lecture notes for the Fundamental Theorem of Arithmetic, Euler's ϕ function, and Euler's Theorem. The cool part of Euler's Theorem was that we got 2 very useful corollaries from it: Fermat's Little Theorem and a test of primality.

And a small note, the exam will only cover today's lecture and previous lectures.

Module Tricks

We present yet another corollary from Euler's Theorem (useful theorem!).

Corollary 1. *If $gcd(a,n) = 1$, then $a^x \equiv a^{x \mod \phi(n)} \mod n$.*

This may seem a bit of a surprise. How did we get the mod $\phi(n)$ in the exponent? Let us remember our mod tricks and look at the proof.

Proof. We need two tools to prove the corollary:

1. Euler's theorem: if $gcd(a,n) = 1$, then $a^{\phi(n)} \equiv 1 \mod n$
2. The module of a product is the product of the modules:

$$\begin{aligned} z_1 &\equiv y_1 \mod n \\ z_2 &\equiv y_2 \mod n \\ \Rightarrow z_1 z_2 &\equiv y_1 y_2 \mod n. \end{aligned}$$

So if $x = (x \mod \phi(n)) + i\phi(n)$, then

$$\begin{aligned} a^x &\equiv \left(a^{x \mod \phi(n)}\right)\left(a^{i\phi(n)}\right) \mod n \\ &\equiv \left(a^{x \mod \phi(n)} \mod n\right)\left(a^{i\phi(n)} \mod n\right) \mod n && \text{by product of modules} \\ &\equiv \left(a^{x \mod \phi(n)} \mod n\right)(1 \mod n) \mod n && \text{by Euler's Theorem} \\ &\equiv a^{x \mod \phi(n)} \mod n && \text{Done!} \end{aligned}$$

□

Now how does this help us?

Example 2. *Let's say we wanted to calculate $2^{999} \mod 21$. It would be nice if we could avoid calculating 2^{999}. To use the previous corollary, we need to check $a = 2$ and $n = 21$ are relatively prime, $gcd(a,n) = gcd(2,21) = 1$. So we can!*

We now calculate $\phi(21)$. Instead of linearly determining(see definitions) $\phi(21)$, remember that $21 = 3 * 7$ and that $\phi(pq) = (p-1)(q-1)$. Hence, $\phi(21) = 12$.

Putting it altogether:

$$\begin{aligned} 2^{999} &\equiv 2^{999 \mod 12} \mod 21 \\ &\equiv 2^3 \mod 21 \\ &\equiv 8 \mod 21 \end{aligned}$$

That's how the corollary helps - it allows us to reduce the exponent dramatically.

Even using that trick, it may not reduce the exponent to a particularly small number. We may be faced with an irreducible $\mod$ calculation of 2^{100}. We still don't want to have to multiply $2(2(2(2(\dots))))$. But we can re-examine the order with which we multiply, like $2^8 = 2^4 * 2^4$. So if we already had 2^4, we could just square it to get 2^8. In an inductive manner of thought, the same logic can be applied to $2^4 = 2^2 * 2^2$. And this is how we get the method of repeated squaring.

Example 3. *To calculate 2^{64}, let us compute 2^2, then square that to get $2^2 * 2^2 = 2^4$, then square that to get $2^4 * 2^4 = 2^8$, then square that to get $2^8 * 2^8 = 2^{16}$, then square that to get $2^{16} * 2^{16} = 2^{32}$. And finally, square that to get $2^{32} * 2^{32} = 2^{64}$.*

So, in 6 mulitplications we computed 2^{64}. In fact, to compute 2^n, we only need to do $\log(n)$ multiplications. So, by using repeated squaring we can compute a multiplication in logarithmic time (see definitions).

We finish repeated squaring by noting if you are computing a^{2^n+y}, you can use repeated squaring to get a^{2^n} and then multiply by a^y.

RSA

For the last few lectures we have covered the setup of encryption/decryption. (see previous lecture notes for a review).

Now we add the math! It will be alot of math, and we'll build it in the sense of a puzzle. We present the pieces, but it isn't until all the pieces are put together that you get the entire picture. This section may require a couple of read throughs.

The Math of RSA

To run RSA we need:

- Two large primes: p and q
- $N = pq$
- And $e \in (1, N)$, such that e is relatively prime to $\phi(N)$ (which means $gcd(e, \phi(N)) = 1$)

We make a small sidestep to note all of our previous corollaries and theorems allow us to find values for these variables very quickly. We can find p and q quickly with Rabin-Miller. We can compute $\phi(N)$ quickly with $\phi(N) = (p-1)(q-1)$. We can compute $gcd(e, \phi(N))$ quickly with Euclid's Algorithm.

Now back to the encryption/decryption schema. We make public (N, e), known as our *public key*, but we keep private (p, q), known as our *private key.*

Somebody wishing to send message m to us would lookup our public (N,e) and encrypt like this:

$$E(m) = m^e \mod N = c$$

Then, for us to decrypt c and get message m back, we'll need to decrypt like this:

$$D(c) = c^d \mod N = m,$$

where $d \equiv e^{-1} \mod \phi(N)$. That is, d is e's multiplicative inverse under mod $\phi(N)$ and we can find it by using Euclid's Extended Algorithm for e and $\phi(N)$. Notice that finding d requires knowledge of p and q, since they hold the only realistic way of calculating $\phi(N)$. This fact (that finding d requires knowledge of p,q and that p,q is only known by us) is the heart of RSA's security.

For this scheme to be sound, we must be sure that D(E(m)) = m for all valid m. We will use Euler's Theorem and the fact that $ed \equiv 1 \mod \phi(N) \implies ed = 1 + i\phi(N)$ to show this now:

$$\begin{aligned}
D(E(m)) &= D(m^e \mod N) \\
&= (m^e \mod N)^d \mod N \\
&= m^{ed} \mod N \\
&= m^{1+i\phi(N)} \mod N \\
&= m^1 * m^{i\phi(N)} \mod N \\
&= m \mod N * m^{i\phi(N)} \mod N \\
&= m \mod N * 1^i \mod N \\
&= m
\end{aligned}$$

To repeat, the required conditions on m are:

- $m < N$ (which we can guarentee by breaking up larger messages into packets)
- gcd(N, m) = 1 (so Euler's Theorem applies when decrypting)

In a similar way you can show $E(D(m))$ recovers m. And that's all the math that makes RSA work!

Hacking RSA

Now security is a chess game between hackers and coders. Coders create an encryption/decryption. Hackers hack that encryption. Coders patch or change that encryption. Hackers .... you get the picture.

With the previous section we have presented the coder's creation of encryption/decryption.

Play Hackers: If a hacker intercepts a message from Alice to Bob. The Hacker can generate messages he thinks Alice is sending to Bob, encrypt them with Bob's public key(just like Alice did) and check which encoded message matches the intercepted message.

Play Coders: A coder can pad a message, say send Bob $m+$ *hey Bob, the rest of this is just random to thawart hackers 9235013524...*, so that even if the hacker guesses what Alice is sending, he can't match messages.

Play Hackers: A hacker can guess you padded the message with a timestamp, knows at what time you sent the message, and can guess you padded the message with time x, $x + \delta x$, etc.

It's a game. It's a game that never ends.

Birthday Paradox

The Birthday Paradox is a reference to the fact that in a group of n people, there is a high probability that at least two people will share the same birthday - even for surprisingly small n. For example, if the group has 30 people this probability is over 70%, with 50 people it goes to 97%! To understand this phenomenon, it is better to think of the group as a set of pairs of people. With 30 people, the first person has 29 other people she could possible match with. The second person only has 28 (because the pairing of the first and second person was already done). This pattern continues, and we'll find that there are 29+28+...+2+1 = 435 distinct pairs. Now, assuming each day of the year is equally likely to be someone's birthday, finding one matching pair out of 435 seems less surprising! It is worth noting here that to be 100% certain that there is at least one matching pair, we would need n=367. If this isn't immediately obvious, try distributing 367 marbles over 366 jars as evenly as possible. If all goes well, one jar better have 2 marbles!

We can generalize this phenomenon and gain some insight into hash function collisions. Both birthdays and hashes can be thought of as random (or close enough to random) integers. But now instead of going around a room and having everybody shout their birthday, we have random hashes being generated one by one. The same principle must apply. In general, expected value for the number of hashes that can be generated before the first collision is $\sqrt{N}$. This means that collisions can never be forgotten about, even for large N. We leave the calculation of the expected value for the probability section of the course.

Euclid's Extended Algorithm

We finish off with a review of Euclid's Extended Algorithm.

Example Consider finding the inverse of 281 under mod 131. We begin by calculating out the $gcd(281, 131)$ with Euclid's Algorithm. Recall this is a good way to start, because if $gcd(a, n) \neq 1$, then an inverse does not exist.

a	b	quotient	remainer
281	131	2	19
131	19	6	17
19	17	1	2
17	2	8	1

And with 1 more step you'll see the *gcd* is indeed 1. But why did we bother to keep the quotients and remainders around? Notice that for a given row of the table $remainder = a - b * quotient$. Explicitly the table is:

row	$remainder = a - b * quotient$
1	$19 = 281 - 2 * 131$
2	$17 = 131 - 6 * 19$
3	$2 = 19 - 17$
4	$1 = 17 - 8 * 2$

And we have a nice ladder climbing from the bottom row to the top row (that contains 281 and 131). Hence, let us try some substitution:

$$\begin{aligned}
1 &= 17 - 8 * 2 \\
1 &= 17 - 8(19 - 17) && \text{sub from row 3} \\
1 &= 9 * 17 - 8 * 19 \\
1 &= 9(131 - 6 * 19) - 8 * 19 && \text{sub from row 2} \\
1 &= 9 * 131 - 62 * 19 \\
1 &= 9 * 131 - 62(281 - 2 * 131) && \text{sub from row 1} \\
1 &= 133 * 131 - 62 * 281
\end{aligned}$$

Now, where is that inverse?

$$\begin{aligned}
1 &= 133 * 131 - 62 * 281 \\
1 \mod 131 &= 133 * 131 \mod 131 - 62 * 281 \mod 131 \\
1 \mod 131 &= -62 * 281 \mod 131
\end{aligned}$$

Hence, -62 is the multiplicative inverse of 281 under $\mod 131$. But for convention, we want to multiplicative inverse in $(0, 130)$. We can do this by applying $-62 + 131$, to find that the conventional answer to give as the inverse is 69! Try it! Calculate $69 * 281 \mod 131$.