MAT 302: LECTURE SUMMARY

Both Alice and Bob have to compute large exponentials to implement RSA: Alice, $X^e \pmod N$, and Bob, $Y^d \pmod N$. For Bob, in particular, this is a pressing problem, since d is an enormous (say, 100-digit-long) number, so that *no computer* can evaluate Y^d. However, we're not interested in Y^d, but rather, only in $Y^d \pmod N$. So, rather than computing Y^d and then reducing $\pmod N$, we build the reductions into the exponentiation. For example, to evaluate $2^{50} \pmod{27}$, we start with 2 and multiply it by 2, then multiply the result by 2, etc. At each stage, if the answer is larger than 27 we reduce $\pmod{27}$. We have:

$$2 \to 4 \to 8 \to 16 \to 5 \to 10 \to 20 \to 13 \to \cdots$$

Thus, after around 50 steps,* we will arrive at the value of $2^{50} \pmod{27}$. More generally, if we apply the same approach to computing $Y^d \pmod N$ it will take $O(d)$ steps. This is totally impractical, since d is enormous – no computer could perform a task requiring d steps. Thus, if RSA is to work, we need a shortcut, a more efficient way to compute large exponentials modulo N.

There were several suggestions. Lance observed that $2^5 \equiv 5 \pmod{27}$, whence $2^{50} \equiv 5^{10} \pmod{27}$. Since $5^2 \equiv -2 \pmod{27}$, we have

$$2^{50} \equiv 5^{10} \equiv (-2)^5 \equiv -2^5 \equiv -5 \equiv 22 \pmod{27}.$$

Kiavash took this idea further by noting that $2^{\varphi(27)} \equiv 1 \pmod{27}$, whence $2^{54} \equiv 1 \pmod{27}$. This implies that $2^{50} \equiv 2^{-4} \pmod{27}$. It is easily seen that $4^{-1} \equiv 7 \pmod{27}$, which shows once more that $2^{50} \equiv 22 \pmod{27}$. Both these ideas are good ones, if somewhat ad hoc, and will certainly save some time.

A third idea, suggested by Serge, is known as the Square-and-Multiply algorithm, and is an extreme shortcut. To illustrate the idea before describing it in general, consider once again the problem of computing $2^{50} \pmod{27}$. Start with 2, and square it repeatedly (reducing module 27 when appropriate) to create the following list of numbers:

$$2 \to 2^2 \to 2^4 \to 2^8 \to 2^{16} \to 2^{32}$$

How does this help? Easy: $2^{50} = 2^{32} \times 2^{16} \times 2^2$, all three of which we computed above. More generally, to compute Y^d, start with Y, and square repeatedly to obtain the following list:

$$Y \to Y^2 \to Y^4 \to Y^8 \to Y^8 \to \cdots \to Y^{2^k}$$

where 2^k is the largest power of 2 such that $2^k \leq d$. We will show below that Y^d can then be expressed as the product of distinct elements from this list. Before doing so, we explored how efficient this algorithm is.

Date: March 10th, 2011.

*I use the word 'step' rather loosely throughout to mean a single, simple computation. More formally, one should think of addition or multiplication of one 1 digit number by another as a basic step.

The number of steps to compute the list is $O(k)$ (I'm counting each squaring as one step, and if you're reducing modulo N, that's an extra step). How big is k? Well, by definition, 2^k was the largest power of 2 which was $\leq d$, i.e. $2^k \leq d < 2^{k+1}$. Taking logarithms yields

$$k \leq \log_2 d < k + 1.$$

But this is exactly the same as saying that $k = [\log_2 d]$ (make sure you understand why). Thus, the number of steps in the Square-and-Multiply algorithm is $O([\log_2 d]) = O(\log_2 d) = O(\log d)$. This is pretty terrific compared to our original approach, which took $O(d)$ steps. For example, if d is a 100-digit number, then the original algorithm takes a ridiculously huge (undoable) number of steps, whereas Square-and-Multiply takes on the order of 100 steps, which is extremely doable.

OK, so Square-and-Multiply is super fast, but how do we know that it will always work? Sure, we happened to be able to write $2^{50} = 2^{32} \times 2^{16} \times 2^2$, but how do you know that it's always possible to write Y^d as the product of a bunch of distinct numbers of the form Y^{2^ℓ}? This is the same as asking whether d can be written as the sum of a bunch of distinct powers of 2. (Note that if we remove the word *distinct*, this becomes trivial: just write d as the sum of a bunch of 1's!) Inspired by ideas of Scott and Kiavash, we proved the following:

Proposition 1. *Any positive integer n can be written as the sum of distinct powers of 2.*

Proof. We proceed by induction. Base case: the proposition clearly holds for $n = 1$ and $n = 2$.

Suppose the claim holds for every positive integer $< n$. We show that it must hold for n as well, thus completing the proof. There are two cases:

(1) n is even

> Then $n = 2k$ for some integer k. Since $k < n$, we can write k as the sum of distinct powers of 2. But then $n = 2k$ is also the sum of distinct powers of 2!

(2) n is odd

> Then $n = 2k + 1$ for some integer k. Once again, since $k < n$, induction tells us that k can be written as the sum of distinct powers of 2. It follows that $n = 2k + 1$ can be, as well.

□

We finished off the class by trying to formalize the notion of an easy or a hard problem. One example of an easy problem is adding two D-digit numbers together; this takes $O(D)$ steps. Another is multiplying two D-digit numbers together; this takes $O(D^2)$ steps. Note that both these examples have very few steps involved: if the numbers we're adding (or multiplying) are 100 digits long, then we only need about 100 (or 10,000) steps, easily handled by a computer. By contrast, I've kept referring to factoring as a hard problem. Why is this? Suppose we wish to factor an RSA number $N = PQ$, where P and Q are (secret) primes. How could we do this? Note that one of P or Q must be less than $\sqrt{N}$, so it suffices to test whether N is divisible by every odd number up to $\sqrt{N}$. This approach takes $O(\sqrt{N})$ steps, which seems pretty good – just like the easy problems

above, the number of steps is a power of the size of the input. Actually, this is deceptive, as is easily seen by imagining that N is huge – say, 200 digits long – and observing that the algorithm requires a vast (and totally unrealistic) number of steps. To put this algorithm on the same footing as those for addition and multiplication, we calculated the number of steps in terms of the number of digits of N. Say N is D digits long. Then $N = O(e^D)$ – why is this? – whence the number of steps required to factor $N = PQ$ is $O(e^{D/2})$. We will return to this topic next lecture.