Theoretical Cryptography, Lecture 13

Instructor: Manuel Blum
Scribe: Ryan Williams

March 1, 2006

1 Today

- Proof that $\mathbb{Z}_p^*$ has a generator
- Overview of Integer Factoring
- Discrete Logarithm and Quadratic Residues

2 Generators

We'll prove the following fact stated in the previous lecture.

Fact 2.1 *For all primes p, $\mathbb{Z}_p^*$ has a generator,* i.e. *an element g such that for every $i \in Z_p$, there is a k such that $g^k = i$.*

Before we do this, we state a very useful and old algebraic lemma.

Lemma 2.1 *Let F be a field. Let $f(x) \neq 0$ be a polynomial of degree at most d, with coefficients in F. Then $f(x)$ has at most d roots.*

Proof. By induction on d. When $d = 0$, $f(x) = a$ for some $a \neq 0$, so f has no roots. In the induction step, suppose $f(x)$ has a as a root. Dividing $f(x)$ by $(x - a)$, we can express f as $f(x) = (x - a)g(x) + r$ for some polynomial g of degree $d - 1$, and $r \in F$. Since a is a root, it must be that $r = 0$. By induction hypothesis, g has at most $d - 1$ roots, so we're done. □

Proof of Fact. We first show that for every $d = 2, \ldots, p - 1$, if there is an element of order d, then there are exactly $\phi(d)$ elements in $\mathbb{Z}_p^*$ of order d.

Suppose a has order d. Consider the polynomial $f(x) = x^{d-1} - 1$ in the field $GF(p)$ (which is $\mathbb{Z}_p^*$ with 0 and addition). By the above lemma, we know that $f(x)$ has at most $d - 1$ roots. But since a has order d, each of $1, a, a^2, \ldots, a^{d-1}$ are roots, so there are exactly $d - 1$ roots and these are the roots.

We claim that a^i has order d iff $gcd(d, i) = 1$: if $a^{di} = 1$ and $a^{ki} \neq 1$ for all k s.t. $1 < k < d$, then d and i have no common factors. Conversely, if a^i had order $d' < d$, then $(a^{d'})^i = 1$ and $(a^{d'})^d = 1$ (since $a^d = 1$), so $(a^{d'})^{gcd(i,d)} = 1$ as well, but this implies that a has order $d' < d$, a contradiction.

But the number of i such that $gcd(d, i) = 1$ is $\phi(d)$. This proves that, if there is an element of order d, then there are exactly $\phi(d)$ elements in $\mathbb{Z}_p^*$ of order d.

Let $\mathcal{O}(d)$ be the total number of elements of order d. From the above, we know that every element has some order d that divides $p-1$, and that either $\mathcal{O}(d) = 0$ or $\mathcal{O}(d) = \phi(d)$.

Now we show

$$\sum_{d|(p-1)} \phi(d) = p - 1. \quad (*)$$

That is, the sum of $\phi(d)$'s such that d divides $p-1$ is precisely $p-1$. Then, it must be that for every d, there are $\phi(d)$ elements of order d, not 0. Why? Every element has an order d that divides $p-1$, and the total number of elements in $\mathbb{Z}_p^*$ is $p-1$. Adding up all the elements gives

$$\sum_{d|(p-1)} \mathcal{O}(d) = p - 1.$$

Thus, provided $(*)$ is true, it must be that $\mathcal{O}(d) = \phi(d)$ for all d. This implies not only that $\mathbb{Z}_p^*$ has a generator, but also that it has $\phi(p-1)$ generators (recall that an element of order $p-1$ is a generator).

We proceed to proving $(*)$:

$$\begin{aligned}
p-1 &= \sum_{d|(p-1)} |\{a \in \{1, \ldots, p-1\} \ : \ gcd(a, p-1) = d\}|, \quad \text{by counting} \\
&= \sum_{d|(p-1)} |\{b \in \{1, \ldots, (p-1)/d\} \ : \ gcd(b, (p-1)/d) = 1\}|, \quad \text{by properties of } gcd \\
&= \sum_{d|(p-1)} \phi\left(\frac{p-1}{d}\right), \quad \text{by definition of } \phi \\
&= \sum_{q|(p-1)} \phi(q), \quad \text{by rearrangement of terms in the sum}
\end{aligned}$$

□

3 Overview of Integer Factoring

Cryptographers often need problem instances that can be easily randomly generated, but difficult to solve. The problem of integer factorization is useful in this regard. We can generate a random n-bit prime easily, as mentioned before: pick a random n-bit number and test for primality. The prime number theorem implies that, with an efficient primality test, this randomized procedure outputs a prime number in expected polynomial time. However, we do not know how difficult it is to factor. The evidence that it is indeed hard is getting weaker and weaker.

Before Cryptography was interested in factoring, the best known factoring algorithm (taken from Knuth, Volume 2) could factor in $e^{O(sqrtn \ln n)}$ steps, assuming various plausible number-theoretic

conjectures. The current best is due to Lenstra (more can be found on Wikipedia), which runs in $e^{O(\sqrt[3]{n \ln^2 n})}$, again under natural number-theoretic assumptions. Lenstra's algorithm is currently used in practice, and can factor 200-bit numbers within a few weeks. We will just mention as an aside that Shor proved that a quantum computer, if realized, could factor an n-bit number in $O(n^3)$ steps.

Question: Are there other schemes that do not use multiplication as a one-way function?

Answer: Yes, others exist, but for the next ten years, we will probably still be using multiplication as the primary one-way function of choice.

4 Discrete Logarithm

Discrete logarithm is another problem that cryptosystems are based upon. Nobody has yet shown that discrete log and factoring are actually equivalent, but whenever a faster algorithm for one problem is found, then a faster algorithm for the other is found. It would be a good PhD thesis to show that they are indeed computationally equivalent (*i.e.* given a black box for one of the two problems, you can solve the other efficiently).

Let p be a prime, let $a \in \mathbb{Z}_p^*$, and let g be a generator of $\mathbb{Z}_p^*$. The discrete logarithm problem is to find integer x such that $g^x \equiv a \bmod p$. That is, since g is a generator, there is some power that g can be raised to that yields a, and the problem is to simply find that power.

As alluded to above, discrete log seems as hard as factoring. Currently, we know that a quantum computer can also solve it in $O(n^3)$ steps, and that it takes the same number of steps to solve as factoring on a deterministic computer as well. The particular choice of n-bit prime does not seem to affect the time complexity as a function of n, and the problem seems difficult for randomly chosen a (*i.e.* all but a tiny fraction of a's).

4.1 Quadratic Residues

We will look at a special case of the discrete log problem: the logarithms that are even numbers. For g a generator of $\mathbb{Z}_p^*$, elements of the form g^{2k} are called *quadratic residues modulo p*. Half of the elements in $\mathbb{Z}_p^*$ are quadratic residues, the other half are called *non-quadratic residues.*[1]

Quadratic residues have several interesting properties. For one, the order of a quadratic residue is at most $(p-1)/2$.

Theorem 4.1 *If x is a quadratic residue modulo p,* i.e. *$x \equiv a^2 \bmod p$ for some a, then $x^{\frac{p-1}{2}} \bmod p \equiv 1$.*

Proof. If $x \equiv a^2 \bmod p$ then $x^{\frac{p-1}{2}} \equiv (a^2)^{\frac{p-1}{2}} \equiv a^{p-1} \equiv 1 \bmod p$, by Fermat's little theorem. □

Theorem 4.2 *Exactly half of the elements in $\mathbb{Z}_p^*$ are quadratic residues.*

Proof. There are at least $(p-1)/2$ residues, given by g^2, g^4, *etc.* Define $f(x) = x^{\frac{p-1}{2}} - 1$. By

[1] Some mathematicians bafflingly call them quadratic non-residues.

an earlier lemma, the polynomial $f(x)$ has at most $(p-1)/2$ roots. Therefore, at most $(p-1)/2$ elements of $\mathbb{Z}_p^*$ satisfy $x^{\frac{p-1}{2}} \bmod p \equiv 1$, so there can be at most this many quadratic residues. □

Note that for non-quadratic residues, $a^{(p-1)/2} \bmod p \neq 1$. But $a^{p-1} \bmod p \equiv 1$ by Fermat, so $a^{(p-1)/2} \bmod p \equiv -1$. This gives an efficient test to determine if a is a quadratic residue mod p: just compute $a^{(p-1)/2} \bmod p$.

Example. Consider $\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$. 3 is a generator of $\mathbb{Z}_7^*$, since $3^2 = 2$, $3^3 = 6$, $3^4 = 4$, $3^5 = 5$, $3^6 = 1$. In this case, $(p-1)/2 = 3$. The quadratic residues are 1, 2, and 4, as $2^3 \equiv 8 \equiv 1 \bmod 7$ and $4^3 \equiv 64 \equiv 1 \bmod 7$. The non-quadratic residues are the rest: 3, 5, 6.

4.2 Computing Quadratic Residues

We will find it convenient to work with primes p satisfying $p \equiv 3 \bmod 4$. Call p a *good* prime if it has this property. Good primes have the property that square roots of them can be computed very easily.

Theorem 4.3 *Let p be a good prime. One can efficiently compute $\sqrt{a} \bmod p$, for every quadratic residue a.*

Proof. Let a be a quadratic residue. By Fermat,

$$a^p \equiv a \bmod p \Longrightarrow a^{p+1} \equiv a^2 \bmod p.$$

Note that $(p+1)/4$ is an integer since $p \equiv 3 \bmod 4$. Therefore $a^{(p+1)/4} \equiv \sqrt{a} \bmod p$, so computing $\sqrt{a}$ is equivalent to computing $a^{(p+1)/4}$, which can be done efficiently. □

Example. Let $p = 7$. Note $7 \equiv 3 \bmod 4$. As mentioned above, 2 is a quadratic residue. To get a square root of it, we can compute $2^{(p+1)/4} = 2^2 \equiv 4 \bmod 7$. The above method can also be used to prove that a number is not a non-quadratic residue. For example, 3 is not a quadratic residue, since $3^{(p+1)/4} = 3^2 \equiv 2 \bmod 7$, but $2^2 = 4$.

4.3 An (Incorrect) Algorithm for Discrete Log

We'll now discuss an algorithm that *almost* computes the discrete logarithm when p is a good prime, for any generator g. The algorithm's intent is to slowly manipulate the equation $g^x \equiv a \bmod p$, recovering the bits of x one-by-one.

In the below, the "current equation" is a variable that is initially defined as $g^x \equiv a \bmod p$.

First, compute g^{-1} using GCD. Then, repeat the following steps until $x = 0$ or $x = 1$ (that is, the LHS of the current equation is either 1 or g).

1. Check if a is a quadratic residue.
2. If yes, then take the square root of both sides of the equation as described above.
   Write down 0 as the next bit of x.
   (*E.g.* if $x = 1100$ in binary, then the equation $g^{1100} \equiv a \bmod p$ becomes $g^{110} \equiv \sqrt{a} \bmod p$.)

3. If no, then multiply both sides of the current equation by g^{-1}.
   Write down a 1 as the next bit of x.

   (*E.g.* if $x = 1101$ in binary, then the equation $g^{1101} \equiv a \bmod p$ becomes $g^{1100} \equiv a \cdot g^{-1} \bmod p$.

If the above algorithm is correct, it would mean that discrete logarithm can be computed in polynomial time! (At least, it can be done in the case of good primes.) What's wrong with the above?

The problem is that there are actually *two* possible square roots of a quadratic residue. For example, in $\mathbb{Z}_7^*$, 4 is a square root of 2, and $(-4) \equiv 3 \bmod 7$ is a square root of 2. Only one of these roots is "correct", and we don't necessarily obtain it from using the above algorithm. This motivates the problem of finding a *principal* square root, which we will discuss in subsequent lectures.