CS124	Lecture 13

Hopefully the ideas we saw in our hashing problems have convinced you that randomness is a useful tool in the design and analysis of algorithms. Just to make sure, we will consider several more example of how to use randomness to design algorithms.

13.1 Primality testing

A great deal of modern cryptography is based on the fact that factoring is apparently hard. At least nobody has published a fast way to factor yet. (It is rumored the NSA knows how to factor, and is keeping it a secret. Some of you might well have worked or will work for the NSA, at which point you will be required to keep this secret. Shame on you.) Of course, certain numbers are easy to factor– numbers with small prime factors, for example. So often, for cryptographic purposes, we may want to generate very large prime numbers and multiply them together. How can we find large prime numbers?

We are fortunate to find that prime numbers are pretty dense. That is, there's an awful lot of them. Let $\pi(x)$ be the number of primes less than or equal to x. Then

$$\pi(x) \approx \frac{x}{\ln x},$$

or more exactly,

$$\lim_{x\to\infty} \frac{\pi(x)}{\frac{x}{\ln x}} = 1.$$

This means that on average about one out of every $\ln x$ numbers is prime, if we are looking for primes about the size of x. So if we want to find prime numbers of say 250 digits, we would have to check about $\ln 10^{250} \approx 576$ numbers on average before finding a prime. (We can search smarter, too, throwing out multiples of 2,3,5, etc. in order to check fewer numbers.) Hence, all we need is a good method for *testing* if a number is prime. With such a test, we can generate large primes easily– just keep generating random large numbers, and test them for primality until we find a suitable prime number.

How can we test if a number n is prime? The pedantic way is to try dividing n by all smaller numbers. Alternatively, we can try to divide n by all primes up to $\sqrt{n}$. Of course, both of these approaches are quite slow; when n is about 10^{250}, the value of $\sqrt{n}$ is still huge! The point is that 10^{250} has only 250 (or more generally $O(\log n)$) digits, so we'd like the running time of the algorithm to be based on the size 250, not 10^{250}!

How can we quickly test if a number is prime? Let's start by looking at some ways that work pretty well, but have a few problems. We will use the following result from number theory:

Theorem 13.1 *If p is a prime and $1 \le a < p$, then*

$$a^{p-1} = 1 \bmod p.$$

Proof: There are two nice proofs for this fact. One uses a simple induction to prove the equivalent statement that $a^p = a \bmod p$. This is clearly true when $a = 1$. Now

$$(a+1)^p = \sum_{i=0}^{p} \binom{p}{i} a^{p-i}.$$

The coefficient $\binom{p}{i}$ is divisible by p, unless $i = 0$ or $i = p$. Hence

$$(a+1)^p = a^p + 1 \bmod p = a + 1 \bmod p,$$

where the last step follows by the induction hypothesis.

An alternative proof uses the following idea. Consider the numbers $1, 2, \ldots, p-1$. Multiply them all by a, so now we have $a, 2a, \ldots, (p-1)a$. Each of these number is distinct $\bmod p$, and there are $p-1$ such numbers, so in fact the sequence $a, 2a, \ldots, (p-1)a$ is the same as the sequence $1, 2, \ldots, p-1$ when considered modulo p, except for the order. Hence

$$1 \cdot 2 \cdot \ldots \cdot (p-1) = a \cdot 2a \cdot \ldots \cdot (p-1)a \bmod p = a^{p-1} \cdot 1 \cdot 2 \cdot \ldots \cdot (p-1) \bmod p.$$

Thus we have $a^{p-1} = 1 \bmod p$. ■

This immediately suggests one way to check if a number n is prime. Compute $2^{n-1} \bmod n$. If it is not 1, then n is certainly not prime! Note that we can compute $2^{n-1} \bmod n$ quite efficiently, using our previously discussed methods for exponentiation, which require only $O(\log n)$ multiplications! Thus this test is efficient.

But so far this test is just one-way; if n is composite, we may have that $2^{n-1} = 1 \bmod n$, so we cannot assume that n is prime just because it passes the test. For example, $2^{340} = 1 \bmod 341$, and 341 is not prime. Such a number is called a *2-psuedoprime*, and unfortunately there are infinitely many of them. (Of course, even though there are infinitely many 2-pseudoprimes, they are not as dense as the primes– that is, there are relatively very few of them. So if we generate a large number n randomly, and see if $2^{n-1} = 1 \bmod n$, we will most likely be right if we then say n is prime if it passes this test. In practice, this might be good enough! This is not a good primality test, however, if an NSA official you know gives you a number to test for primality, and you think they might be trying to fool you. The NSA might be purposely giving you a 2-pseudoprime. They can be tricky that way.)

You might think to try a different base, other than 2. For example, you might choose 3, or a random value of a. Unfortunately, there are infinitely many 3-pseudoprimes. In fact, there are infinitely many composite numbers n such that $a^{n-1} = 1 \bmod n$ for all a that do not share a factor with n. (That is, for all a such that the greatest common divisor of a and n is 1.) Such n are called *Carmichael numbers*– the smallest such number is 561. So a test based on this approach is destined to fail for some numbers.

There is a way around this problem, due to Rabin. Let $n - 1 = 2^t u$. Suppose we choose a random base a and compute a^{n-1} by first computing a^u and then repeatedly squaring. Along the way, we will check to see for the values $a^u, a^{2u}, \ldots$ whether they have the following property:

$$a^{2^{i-1}u} \neq \pm 1 \bmod n, a^{2^i u} = 1 \bmod n.$$

That is, suppose we find a *non-trivial square root* of 1 modulo n. It turns out that only composite numbers have non-trivial square roots – prime numbers don't. In fact, if we choose a randomly, and n is composite, for at least $3/4$ of the values of a, one of two things will happen: we will either find a non-trivial square root of 1 using this process, or we will find that $a^{n-1} \neq 1 \bmod n$. In either case, we know that n is composite!

A value of a for which either $a^{n-1} \neq 1 \bmod n$ or the computation of a^{n-1} yields a non-trivial square root is called a *witness* to the compositeness of n. We have said that $3/4$ of the possible values of a are witnesses (we will not prove this here!). So if we pick a single value of a randomly, and n is composite, we will determine that n is composite with probability at least 3/4. How can we improve the probability of catching when n is composite?

The simplest way is just to repeat the test several times, each time choosing a value of a randomly. (Note that we do not even have to go to the trouble of making sure we try different values of a each time; we can choose values with replacement!) Each time we try this we have a probability of at least $3/4$ of catching that n is composite, so if

we try the test k times, we will return the wrong answer in the case where n is composite with probability $(1/4)^k$. For $k = 25$, the probability of the algorithm itself making an error is thus $(1/2)^{50}$; the probability that a random cosmic ray affected your arithmetic unit is probably higher!

This trick comes up again and again with randomized algorithms. If the probability of catching an error on a single trial is p, the probability of failing to catch an error after t trials is $(1-p)^t$, assuming each trial is independent. By making t sufficiently large, the probability of error can be reduced. Since the probability shrinks exponentially in t, few trials can produce a great deal of security in the answer.