Lecture 27: Randomized Algorithms: Primality Testing and Fingerprinting

Anup Rao

June 5, 2019

So far in this course we have been learning the basic concepts of probability and counting. We have seen some examples of how these concepts can be used for several practical applications like load balancing, polling, building a classifier, hashing, and so on. In the remaining part of the course, we explore how these ideas can help with the design of randomized algorithms.

A randomized algorithm is any algorithm that uses randomization in its computation. Often, randomized algorithms are far simpler than their deterministic counterparts, and far more efficient. You may have already seen the *quicksort* algorithm for sorting. This is a randomized algorithm that works by picking a random pivot element in each step. I will not describe it here, because there is actually a deterministic algorithm that performs as well as quicksort.

Primality Testing

Suppose you are given a number and want to know whether the number is a prime number or not. This is a very basic algorithmic primitive. Your phone does something like this every time it sets up a secure connection to a website you are visiting. So, it is important that the test is fast!

Here is a simple algorithm to accomplish this, called the Miller-Rabin primality test. Suppose we are given the number n and want to understand whether it is prime or not.

The algorithm is simple to implement, but it is not so easy to see why it works.

1. Compute s, d such that $n - 1 = 2^s d$, and d is not divisible by 2. This can be done by repeatedly dividing $n - 1$ by 2 until you cannot anymore.

2. Pick a uniformly random number $b \in \{1, 2, \ldots, n-1\}$.

3. Compute

   $$(b^d \mod n), (b^{2d} \mod n), (b^{2^2 d} \mod n), \ldots, (b^{2^{s-1} d} \mod n).$$

   Each of these numbers can be computed from the previous one by squaring the previous one and then taking the remainder modulo n.

4. If any of these s numbers is $n-1$, conclude that n is prime. Otherwise, conclude that n is not prime.

The correctness of this algorithm involves understanding some group theory, so we will not discuss it here. The theorem that can be proved is:

Theorem 1. *If n is prime, then the algorithm always concludes that n is prime, no matter b is chosen. If n is not prime, then the algorithm concludes that n is prime with probability at most $1/4$.*

It was a major result when a group of researchers came up with deterministic primality test in 2006. The algorithm is, however, much slower than the randomized algorithm discussed above. See here: `https://en.wikipedia.org/wiki/AKS_primality_test`.

The main selling point of the Rabin-Miller test is that it is extremely fast—much faster than any known algorithm that does not use randomness in some way! So, somehow the use of randomness translates into raw speed.

The theorem above ensures that whenever the algorithm concludes that the number is not prime, it must be correct. So, we can run the algortihm multiple times, using multiple independent choices of b, and output that the number is not prime if any one run concludes that n is not prime. Since each run is independent, repeating this algorithm k times decreases the probability of error to $1/4^k$. For example, if we repeat it 10 times, the probability of error becomes about 1 in a million.

Fingerprinting

Suppose dropbox has a file x and a user has a file y, and both files are of size at most ℓ. Dropbox wants to know whether $x = y$, to decide whether to update their local copy or not. What is the best way to to do this? The obvious way is to send the users computer x, and have the computer respond with whether or not $x = y$. But there is a much more efficient way using a randomized algorithm.

Here we give a simple randomized algorithm for this problem. The algorithm relies on the famous prime number theorem:

Throughout this discussion, log is computed with base 2.

Theorem 2. *The number of primes between 1 and t is asymptotically $t/\log t$.*

More formally, the prime number theorem says that if $\pi(t)$ is the number of primes, then

$$\lim_{n \to infty} \frac{\pi(t)}{t/\ln t} = 1$$

.

1. x, y are converted to viewed as bit strings that represent non-negative integers. Since x is a binary string of length at most ℓ, we have $x \leq 2^\ell$.
2. Dropbox picks a uniformly random prime number p between 1 and ℓ^2.
3. Dropbox sends the user p and $x \mod p$.

4. The user responds with whether or not $x = y \mod p$.

The algorithm is very efficient—since $p \leq \ell^2$, it only takes about $2 \log \ell$ bits to encode p as a binary string. The length of the file x is exponentially longer, it ℓ bits. To pick a uniformly random prime number, the algorithm just samples a uniformly random number and checks if the number is prime or not using the primality test described in the last section. If the number is not prime, it resamples a new random number, and repeates this process until it finds a prime number. By the prime number theorem, this algorithm succeeds in each step with probability $1/\ln \ell$, so the number of repetitions needed is a geometric random variable. The expected number of repetitions is $\ln \ell$.

If I were to implement this in practice, I would precompute a list of $10,000$ prime numbers to sample from every time I wanted to check that two files are equal.

More importantly we have saved a massive amount of communication by using this method. For example, if our files are of size 1 gigabyte, which are 8×10^9 bits long. Then instead of exchanging such a huge number of bits, the solution above will exchange only $2 \log p \leq 4 \log \ell$ bits, which is at most 40 bits.

Let us show that the probability that the algorithm makes an error is small. The claim below implies that in the above example, the probability of error is at most 1 in a million.

Claim 3. *The probability that the algorithm makes an error is at most* $2(\ln \ell)/\ell$.

Proof. If $x = y$, then certainly $x = y \mod p$. On the other hand, if $x \neq y$, then the algorithm makes an error only if $x - y = 0 \mod p$, which happens only if p divides $|x - y|$. If $|x - y|$ has k prime factors, then $|x - y| \geq 2^k$, which means that $k \leq \log |x - y| \leq \log(2 \cdot 2^\ell) = \ell + 1$. However, the number of primes in between 1 and ℓ^2 is about $\ell^2/\log \ell$, so the algorithm only makes an error when it picks one of the at most $\ell + 1$ bad primes out of these. The probability of making an error is thus at most $\frac{\ell+1}{\ell^2/\ln \ell} \leq \frac{2 \ln \ell}{\ell}$. □

Again, repeating the algorithm multiple times drives down the probability of making an error.

Error Reduction For Randomized Algorithms

A RANDOMIZED ALGORITHM is an algorithm that uses randomness and computees the correct answer the majority of the time. A key property of these algorithms is that the probability of making an error can be made extremely small just by repeatedly running the algorithm a few times.

To illustrate this, suppose we have an algorithm that always computes the correct answer with probability at least $2/3$, where the probability is taken over the random coin tosses of the algorithm. Suppose we run the algorithm t times, and output the answer that the algorithm gives us most often. What is the probability that new algorithm makes an error?

To reason about this, let $X_1, \ldots, X_t$ be random variables with

$$X_i = \begin{cases} 1 & \text{if the } i\text{'th run produces the correct answer,} \\ 0 & \text{otherwise.} \end{cases}$$

$X_1, X_2, \ldots, X_t$ are mutually independent. Let $X = X_1 + X_2 + \ldots + X_t$. The expected value of X is at least $2t/3$. The new algorithm makes an error exactly when $X \leq t/2$. By the Chernoff bound, the probability that this happens can be bounded by

$$\begin{aligned} p(X \leq t/2) &= p(X \leq (1 - 1/4)2t/3) \\ &\leq e^{-\frac{(1/4)^2 2t/3}{2}} = e^{-\frac{t}{48}}, \end{aligned}$$

so the probability that the new algorithm makes an error is exponentially small in t.