Theoretical Cryptography, Lecture 10

Instructor: Manuel Blum
Scribe: Ryan Williams

Feb 20, 2006

1 Introduction

Today we will look at:

- The String Equality problem, revisited
- What does a random permutation look like?
- What does a random n-bit factored number look like?

2 String Equality Revisited

In the last lecture, we gave a randomized protocol for two parties (Alice and Bob) to determining if their two n-bit strings (A and B, respectively) are equal. The protocol is summarized as follows: Alice chooses a random prime $p \in [1, 2n\ln(2n)]$, sends $A \bmod p$ and p to Bob, and Bob says "equal" if and only if $A \bmod p = B \bmod p$. Since $p \leq O(n \log n)$, it takes only $O(\log n)$ bits to send p and a modulus of p.

We argued about the difference $C = |A - B|$ to prove the correctness of this protocol. A slightly different proof of correctness can be gotten from applying the Chinese Remainder Theorem.

Corollary 2.1 (To the Chinese Remainder Theorem) *Let A and B be* distinct *positive n-bit integers. Then $A \equiv B \bmod p$ for at most n distinct primes p.*

Proof. Suppose we listed all the primes from 2 up to $\max\{A, B\}$, and marked all primes p such that $A \equiv B \bmod p$. Let $\{p_1, \ldots, p_k\}$ be the set of all marked primes. We claim that $\prod_{i=1}^{k} p_i < 2^n$. Otherwise, by the Chinese Remainder Theorem, the map

$$a \mapsto \langle a \bmod p_1, a \bmod p_2, \ldots, a \bmod p_k \rangle$$

is a bijection on the integers from 1 to 2^n, so

$$\langle A \bmod p_1, A \bmod p_2, \ldots, A \bmod p_k \rangle = \langle B \bmod p_1, B \bmod p_2, \ldots, B \bmod p_k \rangle \Longrightarrow A = B,$$

contradicting our hypothesis that $A \neq B$. Therefore $\prod_{i=1}^{k} p_i < 2^n$.

Since each $p_i \geq 2$, it follows that $k \leq n$. □

This corollary implies that executing k rounds of the randomized protocol works with $1 - 1/2^k$ probability: there are at least $2n$ primes in $[1, 2n \ln(2n)]$, so if we choose a prime p uniformly at random from the interval $[1, 2n \ln(2n)]$, there is at most a $1/2$ chance that $A \equiv B \bmod p$.

3 Random Permutations

The primary objective of this lecture is to develop an understanding for what the factorization of a random n-bit number looks like: how many expected factors it has, what are the expected sizes of such factors, and so on. Not only would we like to understand expected properties of random numbers, but we would also like to efficiently *generate* random numbers with their factorizations.

The ability to generate random factored numbers is a useful primitive in cryptography. Eric Bach presented an efficient method for this in his PhD thesis. In this lecture, we outline how his method works. As a sanity check, one should notice that if we are not required to be efficient, then generating a random factored number is trivial! (Pick a random n-bit number uniformly. Factor it. Output that.)

We will approach the question of understanding random integer factorizations by first looking at what a random permutation looks like. We will give two procedures for generating a random permutation: the *sane* method, and the *brain-addled* method. Both methods work, and generate random permutations uniformly at random. The sane method is efficient, and the brain-addled method is ridiculously inefficient. However, the *brain-addled* method, while ridiculous on an algorithmic level, will actually be useful in a mathematical sense. Then we will attempt to extend this approach to generating random factored numbers.

3.0.1 The sane method of generating a permutation

Let $[n] = \{1, \ldots, n\}$ The "sane method" of generating a permutation on $[n]$ is what you think it should be: to get a random permutation π, roll a fair n-sided die, get a number i_1, and define $\pi(1) := i_1$. Then roll an $(n-1)$-sided die that has all numbers except for i_1, get a number i_2, and define $\pi(2) := i_2$. This procedure continues, picking the first $n-2$ integers of the permutation, until we are down to a two-sided coin with integers i_{n-1} and i_n on each side. We toss the coin; say we obtain i_{n-1}. Then we set $\pi(n-1) := i_{n-1}$, $\pi(n) := i_n$. Finally, we output π as the random permutation.

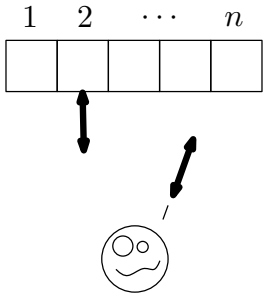
Consider a desired permutation π^*. It is easy to show that the probability that π^* is output is $\frac{1}{n!}$. Thus the "sane method" generates permutations uniformly at random. Note the sane method needs only $n \log_2 n + O(n)$ random bits: it takes at most $\log_2 i + O(1)$ random bits to "roll" a fair i-sided die, and $\sum_{i=1}^{n} \log_2 i + O(1) \leq \log_2(n!) + O(n) \leq n \log_2 n + O(n)$.

3.0.2 The brain-addled method of generating a permutation

We will now present an alternative way to generate a random permutation on $[n]$, that requires *more* random bits and takes *longer* to execute.

Why on earth might such a brain-addled method be useful? The mere fact that the method also generates permutations uniformly at random will give us some additional insight into what a random permutation looks like.

The one-dimensional dartboard, and a drunken dart-thrower. Let's imagine a one-dimensional dartboard with n cells, with a drunken dart-thrower who is very good at missing any particular cell he aims at, with high probability. That is, the dart-thrower has a $1/n$ chance of hitting each cell.



Another way to generate a random permutation is to run the following "brain-addled method":

Pick a random arrangement of $[n]$, and place the elements of $[n]$ on a linked list L.

Repeat forever:

Feed the dart-thrower a beer.

Have him throw a dart into one of n random cells; suppose he hits the ith cell.

Pull the first i elements from the head of L, in order.

Construct a cycle C of length i out of the pulled elements.

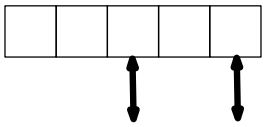
Add C to π (think of π as being represented in terms of its cycles).

If $i = n$, then output π and send the dart-thrower to bed.

Set $n := n - i$.

Intuitively, the dart-thrower is defining the lengths of cycles in the permutation π, and we are filling in these cycles with an ordering on $[n]$.

Example. Let $n = 5$, and let L be $[3, 5, 2, 4, 1]$. Suppose the dart arrangement looks like this:



Then the cycles are of length 3 and 2, and $\pi = (3\ 5\ 2)(4\ 1)$.

Notice that the *first line* of the new brain-addled method is the sane method. Therefore we will use quite a few more random bits in the new method than the sane method, and take more time. We are only doing more work by specifying these cycles upfront. What is interesting is that the brain addled method *also generates permutations uniformly at random.*

Theorem 3.1 *Fix a permutation π^* on $[n]$. The brain-addled method above generates π^* with probability $\frac{1}{n!}$.*

The above theorem can be very useful in inferring properties of a random permutation. Here are a few.

- What's the probability that a random permutation is a cycle? This is just the probability that the dart-thrower throws only one dart in total, and it lands on the nth cell: $1/n$.
- What's the expected number of cycles in a random permutation? The answer is $\ln n + O(1)$, and can be gotten by analyzing the drunken dart-thrower. Let $D(n)$ be the expected number of darts that a dart is thrown, before a dart is in the nth cell.

  We claim that $D(n) = D(n-1) + \frac{1}{n}$. This can be proved by induction on n, but here is a shorter argument. Consider the indicator variable X_i that is 1 or 0 depending on whether or not cell i gets a dart. Clearly $D(n) = E[\sum_i X_i]$. Break the expectation $D(n)$ into two cases: (1) $E[X_1]$, and (2) $E[X_2 + \cdots + X_n]$. The first case is equal to $1/n$, since we choose the 1st cell with $1/n$ probability, and if it is not chosen first, then the 1st cell is never chosen. The second case amounts to just $D(n-1)$. By linearity of expectation, $D(n) = D(n-1) + \frac{1}{n}$.

  Note $D(1) = 1$. Unfolding the recursion, $D(n) = \sum_{i=1}^{n} 1/i \leq \ln n + \gamma$, where $\gamma \approx 0.577$ is Euler's constant.
- Let n be even. What's the probability that a random permutation consists of exactly two $n/2$ cycles? (As we shall see, this is related to the problem of a random number being the product of two large primes.) The probability is $1/n \cdot 2/n = 2/n^2$.

Rather than proving the entire theorem, we just give the proof for $n = 3$.

First, consider the identity map on [3]. This is generated by the above method if and only if the first dart hits cell 1 and the second dart hits cell 2, because this is the case iff the permutation is a collection of 1-cycles. But a dart is thrown in cell 1 with probability 1/3, and then in cell 2 with probability 1/2, so the probability of the identity is $1/6 = 1/3!$, as we would expect.

Let's now think of the permutations which are cycles. There are two distinct cycles, (1 2 3) and (1 3 2). A cycle is obtained with probability 1/3, by hitting cell 3 first. Given this, a random arrangement will result in (1 2 3) or (1 3 2), with probability 1/2 each. Hence both of the cycles have probability 1/6 of being generated.

Finally, we consider the permutations which are a 1-cycle and a 2-cycle. There are three of these: (1)(2 3), (2)(1 3), (3)(1 2). These are generated either when the first dart hits cell 1 and the second dart hits cell 3, or when the first dart hits cell 2. The first situation happens with probability 1/6, and the second happens with probability 1/3. Fixing one of the two situations, each of the three permutations is generated with 1/3 probability. (The probability amounts to which of the three elements is sent to the 1-cycle.) Therefore, each of the above permutations is generated with probability $1/6 \cdot 1/3 + 1/3 \cdot 1/3 = 1/18 + 1/9 = 3/18 = 1/6$.

4 Generating Random Integers With Their Factorizations

Having built up some intuition about one-dimensional dartboards, it is time to switch over to the problem of generating random factored integers. Let's start with a naive way of doing it, and see what goes wrong.

4.0.3 The Wrong (But Promising) Way

Effectively, our high-level idea is strongly related to the random permutation algorithm: to generate a number N, we first pick the lengths of the prime factors in N, then we pick the factors themselves.

Let's return to our drunken friend and his 1-D dartboard. Think of darts being thrown as before: chuck a dart at the board, remove all cells to the left of that dart, then chuck another dart, repeating until the nth cell gets a dart. Imagine that the n cells represent the bits of some n-bit number N we are randomly choosing. Then, the number of empty cells between two consecutively thrown darts will roughly indicate the number of bits in each prime factor of N.

More precisely, for two darts d_1 and d_2 thrown consecutively, the number of bits in the corresponding factor of N will be $1 + |j - i|$, where i and j are the positions of d_1 and d_2, respectively. So for example, two darts adjacent to each other means that there is a 2-bit factor, *i.e.* either 2 or 3.

In this manner, from a particular dartboard configuration we obtain a sequence of numbers $1 + |i_1 - i_2|$, $1 + |i_2 - i_3|$, *etc.*, where i_k is the cell in which the kth dart was thrown. These numbers are the number of bits in each factor of N. We then pick a random prime of $1 + |i_1 - i_2|$ bits, a random prime of $1 + |i_2 - i_3|$ bits, and so on. Multiplying all of these primes together, we output some number N.

What's wrong with the above procedure? There are several problems; here are two major ones.

1. If the procedure is correct, then the number of n-bit integers divisible by 3 is roughly the same as the number of n-bit integers divisible by 2. Of course this is false; the first quantity is roughly 1/3 of the numbers and the second quantity is roughly 1/2. The probability of a prime being chosen should be biased towards smaller primes.

2. If the procedure is correct, then the probability of generating a prime is $1/n$. (Recall $1/n$ is the probability of the dart-thrower hitting the nth cell on the first throw.) Hence there are $2^n/n$ primes on n-bits.

   That contradicts the Prime Number Theorem. To see this, let $N = 2^n$. Then the above estimate is that there are $N/\log_2 N$ primes, whereas the Prime Number Theorem says that the estimate is more like $N/\ln N$.

4.0.4 Generating n-Yit Factored Integers

The first issue we highlighted above does not seem too hard to handle; we just have to adjust the probabilities of choosing each prime. The second issue appears to be a more substantial obstacle. In order to deal with the number-of-primes issue, let's change our problem to explicitly account for the Prime Number Theorem. Let's forget about trying to generate a random factored integer from 1 to 2^n, and instead look at generating in the interval $[1, e^n]$. That is, we work over base-e, instead of base-two.

We say that N is an *n-yit number* if N is an integer in the range $[1, e^n]$. As before, we use the dartboard to generate a sequence of numbers $n_1 = 1 + |i_1 - i_2|$, $n_2 = 1 + |i_2 - i_3|$, *etc.* However, instead of choosing primes as we did earlier, we'll choose distinct *prime powers* $p_1^{e_1}$, $p_2^{e_2}$, *etc.*, where $p_k^{e_k}$ is an n_k-yit number for all k. That is, $n_1 = \ln(p^{e_1})$, $n_2 = \ln(p^{e_2})$, *etc.*

Here's a little more detail of how the procedure goes. We first choose a random prime power $p_1^{e_1}$ such that $e_1 \ln p_1 = n_1$.[1] Next, we choose a random prime power $p_2^{e_2}$ such that $p_1 \neq p_2$ and $e_2 \ln p_2 = n_2$. Finally, choosing the kth prime power $p_k^{e_k}$ is done where $p_k \neq p_j$ for all $j < k$, and $e_k \ln p_k = n_k$.

We will not go into the details here of how the generating procedure can be done efficiently. (Note in the previous lecture, we saw how to choose a prime at random, using a primality test.) As mentioned earlier, we mainly wanted to provide a strong intuition for how Eric Bach arrived at his generation method.

5 Some Applications/Verification of the Procedure

Lemma 5.1 *The number of prime powers up to N is asymptotic to the number of primes up to N.*

Proof. Note that the number of prime powers up to N is at most the number of primes up to N, plus the number of prime-squares up to N, plus the number of prime-cubes, up to the number of primes raised to the $(\log_2 N)$th power. This is bounded from above by

$$\pi(N) + \sqrt{N} + \sqrt[3]{N} + \cdots + \sqrt[\log_2 N]{N} \sim \frac{N}{\ln N}.$$

□

Lemma 5.2 (Prime Number Theorem) *The number of n-yit primes is $\sim e^n/n$.*

[1] N.B. We will disregard floors and ceilings for the sake of presentation. Our probability estimates are asymptotic estimates, anyway.

5.1 Partial Verification

Now we prove some simple results that give a partial verification of the above procedure. For example, each prime is generated (asymptotically) uniformly at random.

Corollary 5.1 *Fix a prime $p \in [1, e^n]$. The probability that the procedure generates p is $\sim 1/e^n$.*

Proof. The probability that the procedure outputs p is the probability that the first dart goes into the nth cell, times the probability that p is chosen out of all the possible prime powers, is

$$\Pr[p] \sim \frac{1}{n \cdot e^n/n} = 1/e^n,$$

as desired. □

What is the probability that a 1-yit number is 2? By the above, the probability is roughly $1/e$. We can build upon that fact with another corollary, showing that slightly more than the primes are also uniformly generated:

Theorem 5.1 *Let $N = 2p$ where p is prime and $N \in [1, e^n]$. The probability that the procedure generates N is $\sim 1/e^n$.*

Proof. In order to obtain N, we need to have that either: (1) the first dart falls in the $(n-1)$th cell, or (2) the first dart falls in the 1st cell, and the second dart falls in the last cell. These are the only two ways that the product is split into an $(n-1)$-yit number and a 1-yit number. The probability that p is chosen from a random $(n-1)$-yit prime is $\sim (n-1)/e^{n-1}$. The probability that 2 is chosen is $\sim 1/e$.

Therefore, in case (1) of the above, N is generated with $\frac{1}{n} \cdot \frac{n-1}{e^{n-1}} \cdot 1/e$ probability. In case (2), N is generated with $\frac{1}{n(n-1)} \cdot 1/e \cdot \frac{n-1}{e^{n-1}}$ probability. Putting it all together, the probability that N is generated is

$$\sim \frac{1}{n} \cdot \frac{n-1}{e^{n-1}} \cdot 1/e + \frac{1}{n(n-1)} \cdot 1/e \cdot \frac{n-1}{e^{n-1}} = 1/e^n \cdot \left(\frac{n-1}{n} + \frac{1}{n} \right) = 1/e^n.$$

□

Finally, we move to a more general case: the probability of generating a number with k primes, each with exactly n/k-yits each.

Theorem 5.2 *Let $N = p_1 \cdots p_k$ where $p_1, \ldots, p_k$ are prime, $p_i \in [e^{n/k-1}, e^{n/k}]$ for $i = 1, \ldots, k$, and $N \in [1, e^n]$. The probability that the procedure generates N is $\sim 1/e^n$.*

Proof. In order to have this kind of prime distribution we need that the dartboard puts a dart at the $(i \cdot n/k)$th cell, for $i = 1, \ldots, k$. Fix primes $p_1, \ldots, p_k$. Suppose p_1 is chosen first, p_2 is chosen second, *etc.* To account for all the other possible orderings of these primes, we add a $k!$ factor. The probability is then

$$\sim k! \cdot \left(\frac{1}{n} \Pr[p_1] \cdot \frac{1}{n - n/k} \Pr[p_2] \cdot \ \cdots \ \cdot \frac{1}{n/k} \Pr[p_k] \right),$$

where $\Pr[p_i]$ is the probability p_i is chosen as an n/k-yit prime. But we know that $\Pr[p_i] \sim \frac{n/k}{e^{n/k}}$ for all primes. Simplifying, we get

$$\begin{aligned} k! \cdot \left(\frac{\Pr[p_1]}{n}\right)^k \cdot \frac{1}{1(1-1/k)(1-2/k)\cdots(1/k)} &= k! \cdot \left(\frac{\Pr[p_1]}{n}\right)^k \cdot \frac{k^k}{k!} \\ &= \left(\frac{k\Pr[p_1]}{n}\right)^k \sim \left(\frac{k \cdot n/k}{n \cdot e^{n/k}}\right)^k = 1/e^n. \end{aligned}$$

□

Similar results hold for prime powers, due to the asymptotic equivalence of the number of primes and the number of prime powers. This leads to the general result that every number is generated uniformly.

5.2 Two Applications

An interesting aspect of Eric Bach's method, besides its usefulness in cryptography, is all of the nice corollaries that one can obtain from it. For example, what is the probability that a random n-yit number has an $\lfloor n/2 \rfloor$-yit prime factor? The probability is $1/2 - o(1)$, since there's a $1/2$ chance that the first dart thrown will be among cells $\{\lceil n/2 \rceil, \ldots, n\}$, and by Lemma 5.1, there is a $1 - o(1)$ chance that a random $n/2$-yit prime power is prime. Thus a number with a large prime factor is generated with roughly $1/2$ probability. Here's another good example.

Corollary 5.2 *The expected number of distinct prime powers in the factorization of a random n-yit number is* $\ln n + O(1)$.

That is, the expected number of prime factors in a number chosen in $[1, N]$ is $\ln \ln N + O(1)$.

Proof. Recall the expected number of darts thrown in a board of n cells: it is $\ln n + O(1)$. This corresponds to the expected number of distinct prime powers, by construction. □

6 Supplementary Note

There exists an easier-to-analyze procedure for generating random factors, however there do not seem to be as many nice implications from its analysis. Google for Adam Kalai's paper " Generating Random Factored Numbers, Easily." It's a very short read (2 pages) and it gives another perspective on the problem. Kalai uses the 1-D dartboard method as well, but in a different way.