Glossary

a.s. = almost surely = with probability 1

$[\![$some event$]\!]$ = the indicator function for occurrence of the event

$\mathbb{N}_+ =$ the positive integers

$[n] =$ the set $\{1, \ldots, n\}$, provided $n \in \mathbb{N}_+$.

Chapter 1

Term B Appetizers

1.1 Lecture 1 (2/Oct): Sampling factored numbers

We're going to start off with an example that is self-contained yet connects nicely to some important problems.

The context is this. As you are probably aware, we have the following contrast.

1. The "primality testing" problem:

   Given an n-bit number m, is m prime?

   is computable in polynomial time. Already in the 1970's people put this problem in ZPP.[1] Then, in 2004 it was even found that primality testing is in P [2].

2. The "factoring" problem:

   Given an n-bit number m, give its prime factorization

   has no known sub-exponential-time algorithms (where by exponential time we mean[2] $\bigcup_{c>0} 2^{n^c}$), despite enormous efforts in this regard, driven largely by cryptographic applications [37]. The best known runtime (and even this is only a heuristic estimate, not a proven bound[3]) is around $2^{n^{1/3}}$. (We are, of course, discussing complexity for classical computers. Quantum computers will be able to factor in poly-time if they are built [6, 40].)

[1] A reminder from Lecture **??**:

(a) For a language L to be in RP means that there is a poly-time randomized algorithm which is always correct on $x \notin L$ and usually correct on $x \in L$. In other words, it has only (and not often) "false negatives." So when it outputs Yes, it is always correct (i.e., its transcript, which means its code plus the random bits it chose on this run, proves membership in L); but when it outputs No it can occasionally be wrong.

(b) For L to be in co-RP means that $L^c \in$ RP.

(c) ZPP $=$ RP $\cap$ co-RP. (This is sometimes called a "Las Vegas" type algorithm.)

(d) For L to be in BPP means that there is a poly-time randomized algorithm which on all x is usually correct. (This is sometimes called a "Monte Carlo" type algorithm.)

(e) For L to be in P simply means that there is a deterministic poly-time algorithm which is correct on all x.

[2] Sometimes people reserve this term for the more restrictive $\bigcup_c 2^{cn}$.

[3] Because the runtime estimate relies on a conjecture that numbers generated by the algorithm are about as likely to be smooth as a random number of the same size would be. A number is b-smooth if all its prime factors are bounded by b.

If you want to think about what complexity class factoring belongs to, you have to deal with the fact that it isn't a decision problem (every n has a factorization) but still, it is "essentially" in NP; what it really is, is a TFNP search problem. TFNP is the set of binary functions $f(x,y)$ such that (a) $|y| = |x|^c$ for some constant c, (b) $f \in$P, (c) $\forall x \exists y : f(x,y) = 1$. The corresponding search problem is, given x, to find a y s.t. $f(x,y) = 1$. Obviously this will be in P if P=NP, but the converse is not known to hold, i.e., it could be that TFNP search problems are all in P, yet P$\neq$NP.

Now, in view of this contrast, the following problem is interesting:

Given an n-bit number m, sample a uniform $1 \leq r \leq m$ along with its factorization.

The naïve approach is to pick r at random and then factor it, but in view of the contrast given above, that won't work. We can make procedure calls to a primality testing algorithm, so we can discover when r is not prime, but then we don't know how to go ahead and factor it.

By the way, you might object that you'll just focus your sampling on numbers that have only small factors, and that will be "good enough". Specifically, define:

Definition 1. *A number is b-smooth if all its prime factors are bounded by b.*

If r is $(\log r)$-smooth then the Sieve of Eratosthenes is good enough to factor it in polylog r time. If most numbers were $(\log r)$-smooth, then you might have suggested settling for sampling just from such r's as a reasonable substitute for the original goal.

However, there are two reasons this objection doesn't hold up. The first is merely that it isn't what we asked for: we want to exactly sample from the distribution, not from a distribution that is close to it. The second, more substantial response is that even if you say you're willing to settle for the lesser goal, most numbers simply aren't smooth enough. About 30% of numbers r have a prime factor larger than $\sqrt{r}$, i.e., are not even $\sqrt{r}$-smooth (let alone polylogarithmically smooth). So, about 30% of numbers r have a prime factor whose number of digits is at least half of the number of digits of r itself. Perhaps even more compellingly, over 99% of numbers have a prime factor with at least one quarter as many digits. [4].

So, we must do something clever. The first to solve this was E. Bach [5]. We'll show an elegant (albeit slightly slower) method of A. Kalai [20]. (Notation: as usual, for positive integer a, write $[a] = \{1, \ldots, a\}$.)

Algorithm:

(0) Case $m = 1$ is trivial. Otherwise set $s_0 = m$.

(1) For $i = 1$, etc., iteratively pick $s_i \in_U [s_{i-1}]$ until $s_L > 1, s_{L+1} = 1$.

Test each $s_1, \ldots, s_L$ for primality. Compute $\varepsilon_i = [\![s_i \text{ is prime }]\!]$. [5]

(2) Let $r = \prod_{i=1}^{L} s_i^{\varepsilon_i}$.

(3) If $r \leq m$, then with probability r/m output r. Else restart at line (1).

Clearly the strategy here is to start with the factors of r, rather than starting with r and looking for its factors. The question is, why does this procedure give us a uniform distribution on r, and why

[4] Let $\Psi(m, B) :=$ fraction of numbers $r \leq m$ that are B-smooth. The numbers above come from the following two theorems (the second subuming the first) (see [12],[35],[32], and [46] for plots): (a) For fixed $1 \leq u \leq 2$ and $m \to \infty$, we have that $\Psi(m, m^{1/u}) \sim 1 - \log u$. (b) For $u \geq 2$ this function still has a limit but it's a little more complicated. Theorem (Dickman): For all $u \geq 1$, $\Psi(m, m^{1/u}) \sim \rho(u)$. The notation $\sim$ allows for a multiplicative factor $1 + o(1)$. Here $\rho(u)$ is defined as the solution of the delay differential equation $u\rho'(u) = -\rho(u-1)$ on $u \geq 1$, with boundary condition $\rho(u) = 1$ for $u \in [0,1]$. For $1 \leq u \leq 2$, $\rho(u) = 1 - \log u$; and it is known that $\rho(u)$ is positive for all $u \geq 0$, and tends to 0 roughly as u^{-u}.

[5] Although a randomized algorithm would be sufficient for our purposes, there is in fact a deterministic polytime (i.e., polylog m time) algorithm for this [2].

does it terminate in expected polynomial time? (Ok that's two questions.) Obviously for the latter point we just need to ensure that the expected number of calls to a primality-testing algorithm is polynomial.

Let $q(r)$ be the probability we generate candidate r in line (2). In view of the r/m "rejection sampling" in line (3), in order to show a uniform distribution on the output, we need to show that for $r \le m$, $q(r)$ is proportional to $1/r$.

For $1 < k \le m$ write $H(k) = |\{j : s_j = k\}|$. Let $1 < p_1 < \ldots < p_v \le m$ be all the primes $\le m$. Define Merten's function, $M(m) = \prod_{i=1}^{v}(1 - \frac{1}{p_i})$.

Lemma 2. *Let $\alpha_1, \ldots, \alpha_v$ be any nonnegative integers. Then* $\Pr(H(p_1) = \alpha_1 \wedge \ldots \wedge H(p_v) = \alpha_v) = \prod_{i=1}^{v}(1 - \frac{1}{p_i})p_i^{-\alpha_i} = M(m)\prod_{i=1}^{v} p_i^{-\alpha_i}$.

Proof. Here is an equivalent description of the sampling process. At each $1 < k \le m$, pick an infinite sequence of independent Bernoulli variables $X_0^k, X_1^k, \ldots$ with $\Pr(X_i^k = 1) = 1/k$. Let $h(k) \ge 0$ be the length of the initial run of 1's at k, namely, $h(k) = \min\{i : X_i^k = 0\}$. If k is largest with $h(k) > 0$ then set $s_1, \ldots, s_{h(k)} = k$. If $k' < k$ is the next largest then set $s_{h(k)+1}, \ldots, s_{h(k)+h(k')} = k'$, and so forth. Observe that conditional on $s_{i-1} = a$, $\Pr(s_i = b) = \frac{1}{a}\prod_{\ell=b+1}^{a}(1 - \frac{1}{\ell}) = \frac{1}{a}$, which is why this is the same process as described in the algorithm. This process gives a product distribution over geometric random variables, that is, if we set $r_k = 1 + \max\{i : s_i \ge k\} - \min\{i : s_i \le k\}$ then $\Pr(\bigwedge_k H(k) = r_k) = \prod_k (1 - \frac{1}{k})(\frac{1}{k})^{r_i}$. "Marginalizing over" the terms with k composite (that is, summing over all the values they might take, which doesn't affect the expressions since this is a product distribution) yields the lemma. □

Corollary 3. *Let r be an m-smooth number, $r = \prod_1^v p_i^{\alpha_i}$. Then $q(r) = M(m)\prod_{i=1}^{v} p_i^{-\alpha_i} = \frac{1}{r}M(m)$.*

This already verifies correctness of the process, since $q(r)$ is proportional to $1/r$.

Now we deal with the runtime. The first thing to do is to demystify $M(m)$: by Merten's "third theorem" [30, 45] this limits to $\frac{e^{-\gamma}}{\log m}$, for the Euler–Mascheroni constant $\gamma \cong 0.577$.

The number of rounds of the algorithm (i.e., the number of "restarts" plus 1) is a geometric rv; the probability that any particular round is the last is

$$\sum_{1 \le r \le m} q(r)\frac{r}{m} = \sum_{1 \le r \le m} \frac{1}{r}M(m)\frac{r}{m} = M(m). \tag{1.1}$$

This is great news because it tells us that we have inverse-linear (in our complexity parameter) probability of success in a round. All that's really left is to bound the number of primality tests(which is our main computational cost) per round.

Let $Y_i(k)$ be the number of times we test whether k is prime, in round i of the algorithm, and let $Y_i = \sum_k Y_i(k)$. There is a fine point here about whether we re-test the same number k if we encounter it repeatedly. It doesn't matter for the asymptotic runtime. So for simplicity, let us suppose that we do not re-test within the same round, but do re-test across rounds. (So $Y_i(k) \in \{0, 1\}$ but $Y(k)$ can be larger than 1.)

Now $\Pr(Y_i(k) = 1) = \frac{1}{k}$ so $E(Y_i) = \mathcal{H}(m)$ where $\mathcal{H}$ is the harmonic sum, $\mathcal{H}(m) \cong \gamma + \log m$.

Combining this information with the expected number of rounds, which from (1.1) is a geometric rv with parameter $M(m)$ and therefore has expectation $\frac{1}{M(m)}$, we have that the total expected number of primality tests is $O(\log^2 m)$.

Chapter 2

Randomized and Distributional Complexity; Linear Programming

2.1 Game tree evaluation: deterministic algorithms

A *game tree* is a tree—in general it can be infinite, but here we assume it finite—in which levels of the tree alternately belong to the "0 player" and the "1 player". The leaves of the tree are labeled with either 0's or 1's. The game starts with a token at the root, and the players take turns, each playing when the token is at a level they own, moving the token down to one of its current children. The goal of each player is to reach a leaf with its own label.

E.g., chess, where the players are called White and Black. White chooses a first move among a finite number of possibilities. White's move gives black a finite number of possible next moves, and so forth.

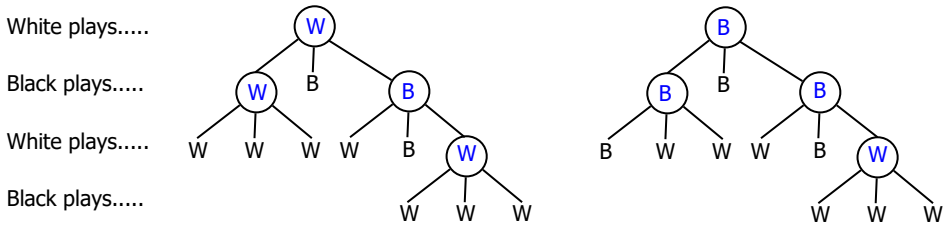

Figure 2.1: Two game trees with same geometry. Labels indicate subtree winners.

The tree is finite, so the game ends when we hit a leaf, which is marked by the name of the winner. (Whether the game of chess induces a finite game tree depends on some fine points of the rules that have little bearing on normal game play.) See Fig. 2.1

Returning to general game trees, note a game always has a winner: this we observe by induction on the depth of the tree (maximum depth of any leaf). Label a vertex with the identity of the player who wins the game if we root the tree at that vertex. (This labeling is separate from who owns the vertex.) Depth 0 is trivial because a leaf is automatically correctly labeled regardless of who owns it. Depth $t \geq 1$: if the owner is b, label the vertex b if any of the children has label b, otherwise label the vertex $1 - b$.

Let N be the number of leaves. We are interested in the worst-case complexity of evaluating the tree. We assume we know the structure of the tree but need to make queries to find out the labels of leaves.

Clearly we can evaluate the tree deterministically with N queries; this is also best possible, for any tree. We prove this by induction on the depth of the tree, by showing that the adversary can reply to queries in such a way that so long as we have not queried all leaves, he retains control over the labeling of the root. For depth 0, i.e., $N = 1$, the root label is precisely the adversary's label.

Before carrying out the formal induction it's worth seeing how it works just for depth 1: if the root is owned by player b, the adversary keeps replying with "$1 - b$" to every query, thereby leaving the value of the root in question, until the final leaf is queried.

The same idea works higher up. For root depth $t \geq 1$, suppose the root v is owned by player b and has children $v_1, \ldots, v_c$. Inside each of the subtrees rooted at v_i ($i = 1, \ldots, c$), the adversary uses his recursive strategy up until the very last leaf in the subtree is queried. That is, until that point, his responses don't depend on what queries and answers have happened outside the subtree, and they simply ensure that $\ell(v_i)$ is undetermined. Finally when the last leaf under v_i is queried, if that is not the last leaf of v to be queried, the leaf label is chosen to set $\ell(v_i) = 1 - b$.

In this way, $\ell(v)$ is not determined until the last leaf under v is queried. At that point, if the last leaf is in say the v_i subtree, we can (by induction) use the choice at the leaf to set $\ell(v_i)$ at will. The setting at the other children of v to $1 - b$ ensures that $\ell(v) = \ell(v_i)$.

We have shown:

Theorem 4. *The deterministic query complexity of game tree evaluation equals the number of leaves of the tree.*

This problem has an equivalent interpretation in terms of *evaluating a boolean formula.* Interpret a "0-owned" internal vertex of tree T as an AND gate, and a "1-owned" internal vertex as an OR gate; call this formula C_T. See Fig. 2.2.

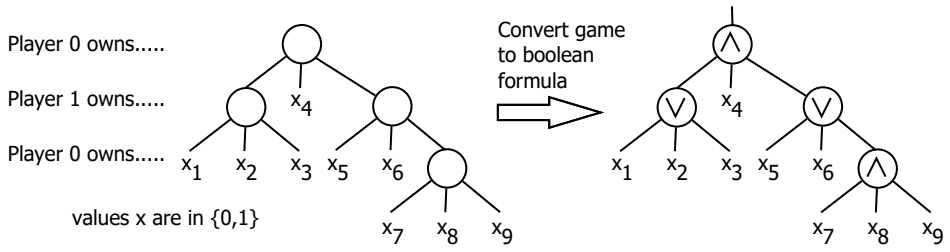

Figure 2.2: Correspondence between Alternating Move Games and Boolean Formulas

This is an exact correspondence because optimal play at any node is (a) if its a winning node (the node owner equals the node label), choose a subtree with the same label; (b) if its a losing node (the node owner differs from the node label) then it doesn't matter what you do. So we have:

Lemma 5. *The winner of game T is the value of the formula C_T.*