Randomized Algorithms

CS6170

Jul - Nov 2021, Dept. of CSE, IIT Madras

Contents

Preface

These are my personal notes for the Randomized Algorithms course for the July - November semester, 2021. The material that is covered here is not new, and there are a number of references available for these topics. As is most often the case, I prefer the presentation of a particular topic better in one book than in another. Therefore, even though the material covered is basic, I have not restricted myself to one textbook. These notes collate the many different expositions and tries to make a consistent presentation.

Caveat Lector! These notes are meant for me to organize my thought before the lectures. It is likely that there are errors in the notes. Read the notes at your own risk!

If you feel that something in the notes do not make sense, then try to look it up on one of the textbooks. Let me know if there are any egregious errors.

1 Introduction

Randomization is ubiquitous in computer science. In many cases, we obtain faster, simpler and more elegant algorithms that better the "polynomial-time" algorithm known for the same problem. An example is the problem of primality testing. Suppose you are given an integer n as input and you want to check if n is prime or not. The trivial algorithm would be to check whether some number between 1 and $\sqrt{n}$ divides n or not. This is highly inefficient since for a number with 100 digits could be as large as 2^{100} and the naive algorithm has a running time for 2^{50}. In a breakthrough result in 2002, AKS showed that there is a polynomial time algorithm for testing primality, and currently the best upper bound known is $O(\log^7 n)$. While this running time does not look as prohibitive as the naive algorithm, it is, nonetheless, not very practical. As it turns out, there is a very simple algorithm to test primality, albeit one that can make an error occassionally. We will make the statement "make an error occassionally" precise in a moment. But, let us first look at the algorithm known as the Miller-Rabin test.

Algorithm 1: Miller-Rabin Primality Test

Input: Integer n

Let $n = 2^r s + 1$, where s is odd
Choose a uniformly at random from $\{2, \ldots, n-1\}$
if $a^s \neq 1 \pmod m$ **then** return `composite`;
for $0 \leq i \leq s-1$ **do**
  if $a^{2^i s} = -1 \pmod m$ **then** return `prime`;
end
return `composite`

By repeated squaring and computation of remainder modulo n, it is easy to see that the running time of the algorithm is $O(\log^2 n)$. As you can see, the algorithm is easy to implement in terms of the calculations that has to be done, and its running time is also good asymptotically. It can shown that if n is a prime, then irrespective of the choice of a then algorithm will always return "prime". When analyzing the algorithm (which is non-trivial and requires some basic algebra), we can show that for at least 3/4th fraction of numbers between 2 and $n-1$, the algorithm will return "composite". In other words, if the number is prime, then the probability that the algorithm errs is zero, whereas if it is composite, then the probability that it errs is at most $1/4$. It is possible to bring down with error probability to something as tiny as $1/2^40$ without changing the asymptotic complexity of the algorithm (we will see how to do this a little later). Note that while the algorithm is simple to explain,

its proof of correctness will not be easy, in most cases.

We will see various settings where randomized algorithms give simple, elegant and fast algorithms where fast deterministic algorithms are not known. Nonetheless, most theoretical computer scientists believe that randomness does not inherently add more computational power. In other words, they believe that if a computational problem has a fast randomized algorithm, then it also has a fast deterministic algorithm! Then why study randomized algorithms at all? Firstly, we are currently far away from that goal of converting every fast randomized algorithm to a fast deterministic algorithm. Secondly, fast here means polynomial-time. So, you could have a randomized primality test that runs in $O(\log^2 n)$ time and the best deterministic primality test may still require $O(\log^7 n)$. That is a considerable gap in many practical situations. Finally, there are scenarios where randomization is unavoidable. One such example is counting motifs in large graphs that are too big to run classical algorithms on. Any algorithm that can approximate these counts necessarily has to be randomized.

In the rest of this lecture, we will see a few more examples while refreshing some basic concepts in discrete probability.

1.1 Polynomial Identity Testing

Suppose that you are given a degree-d polynomial $p(x) = \sum_{i=0}^{d} c_i x^i$ in the explicit way. Your friend claims that the factorization of $p(x)$ is given by $q(x) = \prod_{i=1}^{d}(x - a_i)$ by using a program for polynomial factorization that she has written. How do you check if your friend is indeed telling the truth? The most straightforward way would be for you to expand the factorization $q(x)$ and check that it indeed gives $p(x)$. But, this is tedious since you might end up with far more terms than just the d terms that you will then have to cancel and reduce. With the power of randomness there is something very simple that you can do!

Let us assume for the moment that the numbers involved are all rationals. Consider the polynomial $p(x) - q(x)$. Notice that this polynomial is identically zero precisely when $q(x)$ is the factorization of $p(x)$. Furthermore, $p(x) - q(x)$ is a degree-d polynomials. Now, you know that every polynomial of degree d has at most d roots (this is known as the fundamental theorem of algebra). This gives you an idea for a solution! You choose a number a uniformly at random from the set $\{1, 2, \ldots, 100d\}$, and check if $p(a) - q(a) = 0$. Notice that if $q(x)$ is indeed the factorization of $p(x)$, then for every a, $p(a) - q(a) = 0$ and we will answer correctly. Moreover, if $q(x)$ is not the correct factorization of $p(x)$, then $p(x) - q(x)$ is a non-zero polynomial of degree d. Hence, it has at most d distinct roots, and the probability that a is one such root is at most $1/100$.

A generalization of this problem is the famous *Polynomial Identity Testing* problem (PIT for short). Here you have access to a multivariate polynomial $p(x_1, x_2, \ldots, x_n)$ of degree d. Observe that the number of monomials of this polynomial can be as large as $\binom{n+d-1}{n}$, and the polynomial is expressed succinctly as a product of a small number of polynomials. You want to check if this polynomial is identically zero. Since the explicit description of the polynomial can be exponentially larger than the given representation, it is inefficient to write out the full

polynomial and check whether it is indeed zero. Once again randomness comes to our rescue! We will state a lemma that generalized the fundamental theorem of arithmetic.

LEMMA 1.1 (DeMillo-Lipton-Schwartz-Zippel). *Let $p(x_1, x_2, \ldots, x_n)$ be a non-zero degree d polynomial over rationals. Let S be a subset of rational numbers. Then,*

$$\Pr_{a_1,a_2,\ldots a_n \in_r S}[p(a_1, a_2, \ldots, a_n) = 0] \leq \frac{d}{|S|}$$

Here by $a_1, a_2, \ldots, a_n \in_r S$ we mean that each a_i is picked from S with replacement. We will use this notation through the lecture notes. Observe that when $n = 1$, then the lemma follows from the fundamental theorem of algebra. The lemma can be proved by an induction on n (the number of variables). We will see this shortly.

1.2 Sample spaces, events, probability

Let us recall the notions of sample spaces, events and probability distributions. A randomized algorithm is a random process, and analyzing the algorithm will involve understand the sample space of the random process and the distribution over this sample space. Formally, a *sample space* Ω is the set of possible outcomes of a random experiment. An *event* E is a subset of Ω. A *probability space* is a 3-tuple $(\Omega, \mathcal{F}, \Pr)$ where $\mathcal{F}$ is the set of subsets of Ω and $\Pr : \mathcal{F} \to \mathbb{R}$ is a function that satisfy the following conditions:

1. For every event E, $0 \leq \Pr[E] \leq 1$.
2. $\Pr[\Omega] = 1$
3. For any countable sequence $E_1, E_2, \ldots$ of pairwise disjoint events, $\Pr[\cup E_i] = \sum \Pr[E_i]$.

The following statement, which follows from the inclusion-exclusion principle is very useful, and we will refer to it frequently.

FACT 1.2 (Union bound). *Let $E_1, E_2, \ldots, E_n$ be n events. Then*

$$\Pr\left[\bigcup_{i=1}^{n} E_i\right] \leq \sum_{i=1}^{n} \Pr[E_i]$$

The *conditional probability* of an even E conditioned on another event F is denoted as $\Pr[E|F]$ and is defined as

$$\Pr[E|F] = \frac{\Pr[E \cap F]}{\Pr[F]}.$$

Notice that the conditional probability is well-defined only when $\Pr[F] \neq 0$. Intuitively, the sample space of interest is the set F, and hence we normalize it with $\Pr[F]$ so that the conditional probability remains a valid probability function.

We say that two events E and F are independent if $\Pr[E|F] = \Pr[E]$ and $\Pr[F|E] = \Pr[F]$. In other words, conditioned on the even E(or F) occurring, the probability of the occurrence of F(or E) does not change. An equivalent way to say this is that $\Pr[E \cap F] = \Pr[E]\Pr[F]$. Notice that $\Pr[E \cap F] = \Pr[E]\Pr[F|E] = \Pr[F]\Pr[E|F]$.

Another useful, and fairly straightforward, property described below is known as the law of total probability.

THEOREM 1.3 (Law of Total Probability). *Let $E_1, E_2, \ldots, E_n$ be mutually disjoints events in the sample space Ω such that $\bigcup_{i=1}^{n} E_i = \Omega$. Let F be any event in the sample space Ω. Then,*

$$\Pr[B] = \sum_{i=1}^{n} \Pr[B \cap E_i]$$

Let us now look at the proof of Lemma 1.1. We will explicitly show the underlying sample space and the events of interest to illustrate the definitions and concepts described above.

Proof of Lemma 1.1. The proof is via an induction on n (the number of variables). The base case is when $n = 1$. Now the sample space of the experiment is the set S. Since we are sampling uniformly at random from the set S, the probability function Pr assigns the value $1/|S|$ to every element in the sample space. Now the event that we are interested in is the subset $Z \subseteq S$ such that for every $z \in Z$, $p(z) = 0$. By the fundamental theorem of algebra, we know that $|Z| \leq d$. Therefore,

$$\Pr_{a_1 \in_r S}[p(a_1) = 0] = \Pr[Z] = \frac{|Z|}{|S|} \leq \frac{d}{|S|}.$$

Let us prove the inductive step. The sample space Ω for this experiment is the set of n-tuples over S with Pr being the function that assigns the same value to every element in the sample space. We can write the polynomial $p(x_1, x_2, \ldots, x_n)$ as

$$p(x_1, x_2, \ldots, x_n) = \sum_{i=0}^{d} x^i p_i(x_2, x_3, \ldots, x_n).$$

If p is not identically zero, then there exists an i such that $p_i(x_2, x_3, \ldots, x_n)$ is not identically zero. Consider the largest such i for which p_i is not identically zero. Let $Z_i \subseteq \Omega$ be tuples such that p_i evaluates to zero on these points. By the inductive hypothesis, $\Pr[Z_i] \leq (d - i)/|S|$. Let $Z \subseteq \Omega$ denote the tuples on which p evaluates to zero. By the law of total probability, we can write

$$\begin{aligned}
\Pr_{a_1, a_2, \ldots a_n \in_r S}[p(a_1, a_2, \ldots, a_n) = 0] = \Pr[Z] &= \Pr[Z \cap Z_i] + \Pr[Z \cap \bar{Z}_i] \\
&= \Pr[Z_i]\Pr[Z|Z_i] + \Pr[\bar{Z}_i]\Pr[Z|\bar{Z}_i] \\
&\leq \Pr[Z_i] + \Pr[Z|\bar{Z}_i]
\end{aligned}$$

If the event Z_i does not occur, then we have a polynomial $p(x_1, a_2, \ldots, a_n)$ of degree at most i and hence $\Pr[Z|\bar{Z}_i] \leq i/|S|$. Therefore, we have

$$\Pr_{a_1,a_2,\ldots a_n \in_r S}[p(a_1, a_2, \ldots, a_n) = 0] = \Pr[Z] \leq \frac{d}{|S|}.$$

□

We will see one more example problem of randomization that gives non-trivial savings in the running time while understanding the basic concepts of discrete probability better.

1.3 Verifying matrix multiplication

Suppose that you are provided with three $n \times n$ matrices A, B and C with the claim that $AB = C$. You want to verify whether C is indeed the product of the matrices A and B. The simplest way to do this is to actually perform the matrix multiplication and check AB and C entry-wise. The naive matrix multiplications algorithm takes time $O(n^3)$. Currently, the fastest matrix multiplication algorithm has a running time of $O(n^{2.37})$, though it is conjectured that there is an $O(n^{2+\epsilon})$-time algorithm for matrix multiplication for every $\epsilon > 0$. We will see an $O(n^2)$-time randomized algorithm to verify matrix multiplication over the field $\mathbb{F}_2$ (i.e. we do addition and multiplication modulo 2).

The algorithm is quite simple to state: Choose a random vector $r \in \{0,1\}^n$ and check if $ABr = Cr$. Since r is an $n \times 1$-matrix, the multiplication Br and Cr takes $O(n^2)$ time, and the result of Br is another $n \times 1$-matrix. Thus, the verification can be done in $O(n^2)$-time. It remains to show that the procedure does not err too much. Notice that if $AB = C$, then this procedure does not err at all. Hence all that remains is to show that if $AB \neq C$, then the probability that this process gives a wrong answer is small. The following lemma is sufficient, since if $AB \neq C$, then $D = AB - C$ is a non-zero matrix.

Lemma 1.4. *Let D be a non-zero $n \times n$-matrix. Then, we have*

$$\Pr_{r \in_r \{0,1\}^n}[Dr = 0] \leq \frac{1}{2}$$

Proof. Let $r = (r_1, r_2, \ldots, r_n)$. If r is sampled uniformly at random from $\{0,1\}^n$, then it is equivalent to saying that each r_i is set to $0/1$ with probability $1/2$. Since D is non-zero, assume wlog that $D_{1,1} \neq 0$. Consider the $(1,1)$-th entry of Dr given by $\sum_{k=1}^n D_{1,k} r_k$. We will argue that $\Pr[\sum_{k=1}^n D_{1,k} r_k \neq 0] = 1/2$.

The sample space we are working with is the set $\{0,1\}^n$ with each element in the set being assigned the same probability. The event we are interested in the set of points such that $\sum_{k=1}^n D_{1,k} r_k \neq 0$. First notice that if $\sum_{k=1}^n D_{1,k} r_k \neq 0$, then $r_1 = -\sum_{k=2}^n D_{1,k} r_k / D_{1,1}$. Therefore,

we can use the law of total probability to write our event of interest as

$$\begin{aligned}
\Pr[\sum_{k=1}^{n} D_{1,k} r_k \neq 0] &= \sum_{b_2,b_3,\ldots,b_n \in_r \{0,1\}} \Pr\left[r_1 = \frac{-\sum_{k=2}^{n} D_{1,k} r_k}{D_{1,1}} \cap (r_2, r_3, \ldots, r_n) = (b_1, b_2, \ldots, b_n)\right] \\
&= \sum_{b_2,b_3,\ldots,b_n \in_r \{0,1\}} \Pr\left[(r_2, r_3, \ldots, r_n) = (b_1, b_2, \ldots, b_n)\right] \Pr\left[r_1 = \frac{-\sum_{k=2}^{n} D_{1,k} b_k}{D_{1,1}}\right] \\
&= \frac{1}{2}
\end{aligned}$$

□

An intuitive way to understand the proof is as follows. Suppose that you don't choose all the r_is together; rather you choose them one at a time. Say that we choose $r_2, r_3, \ldots, r_n$ one at a time. Once all of these are fixed, then the probability that you will choose the value for r_1 which will make the sum non-zero is $1/2$. This intuition is sometimes referred to as the *principle of deferred decisions*, and the way to formalize it is through the conditional probability calculations as illustrated above.

Another bothersome point for you might be the guarantee of the algorithm. The analysis shows that the algorithm correctly answers with probability $\geq 1/2$, which does not seem much. Notice that this algorithm has *one-sided error*. Therefore, to reduce the error probability of the algorithm, we need to just repeat it many times with vectors $\mathbf{r}_1, \mathbf{r}_2, \ldots \mathbf{r}_k$ chosen at random and with replacement and check whether $AB\mathbf{r}_i \neq C\mathbf{r}_i$ for even one of them. Let E_i be the event that $AB\mathbf{r}_i = C\mathbf{r}_i$. We are interested in the event $\bigcup_{i=1}^{k} \overline{E}_i$. First we compute

$$\begin{aligned}
\Pr\left[\bigcap_{i=1}^{k} E_i\right] &= \prod_{i=1}^{k} \Pr[E_i], \text{ since the events } E_i \text{ are independent} \\
&\leq \frac{1}{2^k}.
\end{aligned}$$

Therefore, we have

$$\begin{aligned}
\Pr\left[\bigcup_{i=1}^{k} \overline{E}_i\right] &= 1 - \Pr\left[\bigcap_{i=1}^{k} E_i\right] \\
&\geq 1 - \frac{1}{2^k}.
\end{aligned}$$

Observe that if we had sampled $\mathbf{r}_i$ without replacement, then the probability of error can only reduce further since if in the first iteration you do not find a vector $\mathbf{r}$ such that $AB\mathbf{r} \neq C\mathbf{r}$, then in the next iteration you have reduced the space from which you are sampling. But, the number of vectors $\mathbf{r}$ such that $AB\mathbf{r} \neq C\mathbf{r}$ has not reduced. While analyzing such experiments where you are sampling elements one after the other without replacement, the analysis is slightly more complicated due to the conditional probability calculations that you have to do. Moreover, it is easier to implement sampling with replacement than without replacement. In most cases, the bounds that we obtain are useful for the problem at hand.

We will not explicitly write down the sample space and event in every problem that we study if it is clear from the context. We will end this lecture with an important graph problem that arises as a primitive in many scenarios.

1.4 Minimum cut

A *cut-set* in a graph $G(V, E)$ is a collection of edges $E' \subseteq E$ such that the graph gets disconnected. A minimum cut is a cut-set with the minimum cardinality among all cut sets. It is possible that there are more than one minimum cuts in the graph. The simple algorithm that we are going to describe was first given by David Karger in 1993.

The idea of the algorithm is to choose an edge uniformly at random, and contract the end points to obtain a multigraph. In each iteration, the number of vertices reduce by one, and hence after $n - 2$ iterations, there are two vertices remaining. The algorithm outputs the number of edges between these two vertices as the minimum cut of the graph. You can see that the final cut that is obtained is also a cut in the original graph, but it is possible that it is not a minimum cut. We will bound the probability that this procedure does not give a minimum cut.

Algorithm 2: KARGER'S MIN-CUT

Input: Graph $G(V, E)$ with n vertices

1 **repeat**
2 Choose e uniformly at random from G
3 $G \leftarrow G \setminus e$
4 **until** $|V| = 2$;
5 Return the number of edges in G

We have the following lemma about Karger's algorithm.

LEMMA 1.5. *Algorithm 2 returns the minimum cut with probability* $\frac{2}{n(n-1)}$.

Proof. We will argue with respect to a fixed cut C of size k, and compute the probability that this cut remains after the execution of Algorithm 2. A fixed cut C remains after the execution of the algorithm if during every random choice, no cut-edge was chosen.

If the minimum cut size is at most k, every vertex must have degree at least k. Thus there are at least $nk/2$ edges in the graph. Let E_i be the event that the edge contracted in the i^{th} iteration of the algorithm is not an edge in C. Let $F_i = \cap_{j=1}^{i} E_i$ be the event that no edge from C was contracted in the first i iterations. We are interested in computing $\Pr[F_{n-2}]$.

Firstly, $\Pr[E_1] = \Pr[F_1] \geq 1 - \frac{2k}{nk} = 1 - \frac{2}{n}$. If a cut edge was not chosen in the first iteration, we have an $(n-1)$-vertex graph with cut size equal to k. In particular, if a cut edge was not chosen in the first $i-1$ iterations, then we are left with an $(n-i+1)$-vertex graph with cut

size equal to k. Thus, we can say that $\Pr[E_i|F_{i-1}] \geq 1 - \frac{2k}{k(n-i+1)} = 1 - \frac{2}{n-i+1}$. Finally, we have

$$\begin{aligned}
\Pr[F_{n-2}] &= \Pr[E_{n-2} \cap F_{n-3}] \\
&= \Pr[E_{n-2}|F_{n-3}] \Pr[F_{n-3}] \\
&= \left(\prod_{i=1}^{n-3} \Pr[E_{i+1}|F_i]\right) \Pr[F_1] \\
&\geq \left(1 - \frac{2}{n}\right) \prod_{i=1}^{n-3} \left(1 - \frac{2}{n-i}\right) = \prod_{i=1}^{n-2} \left(1 - \frac{2}{n-i+1}\right) = \frac{2}{n(n-1)}.
\end{aligned}$$

□

Once again we note that the success probability of the algorithm is small, but the algorithm has one-sided error. So, we do the standard trick of repeating the algorithm independently k times and taking the minimum value among the results. Analyzing this like in the previous section, the probability that the minimum cut is not output in any of the k iterations is at most

$$\left(1 - \frac{2}{n(n-1)}\right)^k \leq e^{-\frac{2k}{n(n-1)}},$$

where we use the inequality that $1 - x \leq e^{-x}$ (remember this! It is a standard inequality used in a lot of calculations). Thus if we were to repeat this algorithm for $k = n(n-1)\log n$ times, the error probability is at most $1/n^2$.