Lecture Notes, CSE 232, Fall 2014 Semester

Dr. Brett Olsen

Week 12 - Random Algorithms

So normally when we talk about algorithms, we can think about them as a deterministic black box: we put input of some kind into the box, maybe a graph, or a list of integers, or the coordinates of points on a plane, and we get back an answer based on that input: a traversal tree of the graph, or the sorted order of the integers, or the convex hull of the points. For algorithms like this, with a given input we're guaranteed that 1) the output of the algorithm will always be the same (and correct, if the algorithm is correct) and 2) the time taken to run the algorithm, the number of computational steps, will always be the same. Of course, the output and run time will change with different inputs — ask an algorithm to sort a list of 10 elements and it'll run faster than when asked to sort a list of 10,000,000 — but if given the same input, we expect the same answer and the same runtime.

What we're going to talk about today are a separate class of algorithms, called *randomized* algorithms, that take, in addition to the input data, non-deterministic, random numbers, and in which one or both of those guarantees are lost. So if we lose guaranteed output (and sometimes even guaranteed correctness!) or guaranteed run time, and we need to supply random numbers as well, why would we want to use randomized algorithms at all?

Well, there are two upsides: 1) Often randomized algorithms are conceptually simpler, and thus simpler to implement, than deterministic algorithms 2) In exchange for losing guarantees about worst-case performance, well-designed randomized algorithms can give much improved expected performance.

Generating Random Numbers Before I go into more detail about algorithms, let's talk a bit about the extra input into randomized algorithms — random numbers. When I talk about random numbers today, I mean uniform deviates - numbers that lie within a specified range, with any one number in the range just as likely as any other and there are no correlations between successive random numbers. Where can we get random numbers from?

The first option is to gather random numbers from inherently unpredictable physical sources. Some examples of this strategy include coin flipping, dice rolling, radioactive decay, microwave background radiation, etc. This provides (if the physical source is well chosen) truly random numbers, but has the downside that generation of random numbers is relatively costly and slow — you can't provoke radioactive decay on demand, you just have to wait, and rolling dice takes time and effort as well.

Because of this, nearly always when we use random numbers in computer algorithms, what we're really using are *pseudo-random* numbers — generated not by a physical process, but rather by a deterministic "random number generator". I won't go into detail about these other than to give a couple of warnings:

1) You should *never*, literally *never* try to design your own random number generator or modify another implementation. E.g., don't try to do clever things to make a generator "more random" like swap bits around, or combine multiple calls, etc.

2) You should think long and hard — very hard — before trying to *implement* your own random number generator from a known (and tested!) design.

3) You should almost always use the built-in random generator provided by your language, and when you do, every time you need a new random number, you need to make a new call to the generator. One number, one call.

The reason for these warnings is that random number generators are very, very easy to get very, very wrong. Errors in your random number generator can lead to serious problems in your algorithms! A couple of potential pitfalls: random number generators have *periods* — they repeat after some number of calls, and poor implementation can lead to a short period. Gathering 10,000,000 “random” numbers from a generator with a period of 32,000 is obviously a problem. Poor choice of seeds — the initial start to a random number generator — can lead to non-random behavior. A common, but particularly bad, option is to seed with some property of the computer. Usually these seeds are not nearly diverse enough, leading to the same problem.

Case Study in Bad RNG (Taken from http://www.cigital.com/papers/download/developer_gambling.php)
Back in the late 90s, when online poker started getting popular, there was a particular poker site that was trying to build its userbase. To convince users that its shuffles were fair, it publicly published its shuffling algorithm:

- start with an ordered deck of cards
- initialize a RNG with a seed taken from the system clock (a 32-bit integer)
- for each position in the deck, swap the card in that position with a random card from the deck

There are a couple of problems with this: first, the algorithm doesn't produce all shuffled decks with equal probability. The correct algorithm for that is to swap each card randomly with a card at a position greater than or equal to its original position, rather than with any card in the deck. (Take some time and prove that this is the case!) More troubling, though, was the random number generator. The period on the RNG was on the order of 2^{32}, or a billion. But the number of total shuffled decks is much larger, 52!, meaning that many decks will simply never occur as no seed will result in that particular shuffle. This isn't great, but a billion possible shuffled decks is still more than we probably need. But the implementation used seeded the RNG with a system call — the number of milliseconds since midnight on the server clock. There are only a limited number of milliseconds in a day, so the total number of seeds was reduced to only 86 million. But wait — this is a server clock. If we know where the server is physically located, then we can make a pretty good guess about what the server clock time is, and so we can reduce the number of seeds down to a few hundred thousand (if we can get the server clock time to within a few minutes). *This is small enough to do a complete search.* So once we're dealt our hand, we can brute force all possible RNG seeds, pick those that would give us the hand that we see, and then we know the complete shuffle of the deck, including all of our opponents' hands.

The lesson is: think very carefully about the limitations of the random number generator you're using.

Las Vegas vs. Monte Carlo Randomized algorithms are usually classified based on which of those guarantees they choose to give up. Monte Carlo algorithms give up guaranteed correctness: they have a fixed running time, but might get the wrong answer. They usually work by randomly generating a fixed number of possible answers and then report the best one. Las Vegas algorithms give up guaranteed running time: they randomly generate possible answers until they get the correct answer. Note that if you have a way to test for an answer's correctness, converting a Monte Carlo algorithm to a Las Vegas one is trivial — just keep generating answers until one tests as correct.

Let's talk about some times you might think about using randomness and go over some example algorithms.

Lots of Good-Enough Answers A very common situation is a problem where we're asked not for the best answer, but rather just for an answer that satisfies the constraints. Take, for example, a problem where you're given a list of 1,000,000 positive integers less than 1,000 and asked to find a pair of identical integers. There are 10^{12} possible pairs of integers, so we can't possibly test them all. But how common are pairs of identical integers? Obviously it will depend on what the numbers in the list actually are. We might have a list full of identical numbers, say, where they're all 10. In that case, *every* possible pair will be an identical pair. On the other hand, we might have a list with every number equally represented, where there are 1,000 copies of each number. In that case, each random pair will have a 1 in 1000 chance of being identical, which is the worst case.

Avoiding Bad Input Another common use for randomization is in avoiding worst-case input. Let's look at quicksort. This sort works by choosing a *pivot* from the array and adjusting the array so that all elements less than the pivot are at lower indices than the pivot, and all those greater than the pivot are at higher indices. We then recursively sort on the two separate upper and lower sections. On average, this runs in $O(nlgn)$ time, but in the worst case, it can run in the much slower $O(n^2)$ time. If, for example, we choose the pivot as the first element in the array, then when given an already-sorted array to sort, each pivot won't actually divide the problem into two smaller pieces, giving us very poor performance. For many problems, worst-case inputs are not particularly likely. For programming challenges, though, you're almost guaranteed to see a worst-case input. So how can we avoid this problem? Well, rather than simply choosing the pivot as the first element, randomly pick the pivot from all the elements in the array. It's still possible to get a poor performance, but there's no specific input that could be provided to guarantee bad performance. Any time you have an algorithm that performs well in general but poorly on specific kinds of input, think about how randomization might help you avoid the problem.

Simplifying an Implementation Let's look at the minimum-cut problem in a graph. Now, we didn't manage to get to this problem in the graph theory lecture, so let me describe the problem. Let's take some graph G with E edges and V vertices. A *cut* of the graph is a division of the vertices of the graph into two subsets, and the size of the cut is the number of edges between the distinct subsets. So when we ask for a minimum cut, we want to find a division of the graph so that there's a minimum number of edges between the two divisions. There exist deterministic algorithms for this problem, but they're complicated enough that I won't cover them today. Instead let's think about randomized algorithms.

One way we might think of is to generate random subsets of vertices and measure the size of the cut. Suppose that there's exactly one minimum cut of size k. There are $2^{V-1} - 1$ total possible cuts, and only one of them is correct, so we would have to generate a very large number of random subsets in order to get a good chance of finding the minimum — large enough that it would probably be faster to just loop through all the subsets (or, of course, use one of the faster deterministic algorithms). But there's another way we can generate cuts using a method called graph contraction.

In this method, rather than generating an initial random cut, we repeatedly choose random edges and *contract* them, replacing the nodes connected by that edge with a supernode, so that all edges to the two nodes go instead to the supernode, removing any self-loops, but keeping multiedges. We repeat this procedure until we have only two supernodes remaining. Each of these represents a subset of the vertices, and the number of edges between them is the size of the cut they represent. How likely is it that we'll be able to find the minimum cut?

Well, we have k edges of the minimum cut. In order to construct this cut using this contraction algorithm, none of the edges chosen to contract must be one of the k edges present in the minimum cut. The first edge chosen to contract will avoid the minimum cut with probability $1 - k/E$. The minimum degree of the graph must be at least k, so there must be at least $Vk/2$ edges, giving the probability of avoiding the minimum cut on the first contraction as at least $1 - 2/V$. After contraction, the same solution recurs on a graph with one fewer vertex, so the probability of avoiding the minimum cut after fully contracting the graph is:

$p \geq \prod_{i=0}^{V-3}(1 - \frac{2}{V-i}$,

which simplifies down to $\binom{V}{2}$. So with a polynomial, rather than an exponential, number of random choices, we can get a very high probability of the correct answer!

Lab Section This week's homework problem is Google Code Jam Round 1B 2012 - Equal Sums.