Lecture 8

Randomized Algorithms I

Supplemental reading in CLRS: Chapter 5; Section 9.2

Should we be allowed to write an algorithm whose behavior depends on the outcome of a coin flip? It turns out that allowing random choices can yield a tremendous improvement in algorithm performance. For example,

- One can find a minimum spanning tree of a graph $G = (V, E, w)$ in linear time $\Theta(V + E)$ using a randomized algorithm.
- Given two polynomials p, q of degree $n-1$ and a polynomial r of degree $n-2$, one can check whether $r = pq$ in linear time $\Theta(n)$ using a randomized algorithm.

No known deterministic algorithms can match these running times.

Randomized algorithms are generally useful when there are many possible choices, "most" of which are good. Surprisingly, even when most choices are good, it is not necessarily easy to find a good choice deterministically.

8.1 Randomized Median Finding

Let's now see how randomization can improve our median-finding algorithm from Lecture 1. Recall that the main challenge in devising the deterministic median-finding algorithm was this:

Problem 8.1. Given an unsorted array $A = A[1, \ldots, n]$ of n numbers, find an element whose rank is sufficiently close to $n/2$.

While the deterministic solution of Blum, Floyd, Pratt, Rivest and Tarjan is ingenious, a simple randomized algorithm will typically outperform it in practice.

Input: An array $A = A[1, \ldots, n]$ of n numbers

Output: An element $x \in A$ such that $\frac{1}{10}n \leq \text{rank}(x) \leq \frac{9}{10}n$.

Algorithm: FIND-APPROXIMATE-MIDDLE(A)
Pick an element from A uniformly at random.

Note that the above algorithm is *not* correct. (Why not?) However, it does return a correct answer with probability $\frac{8}{10}$.

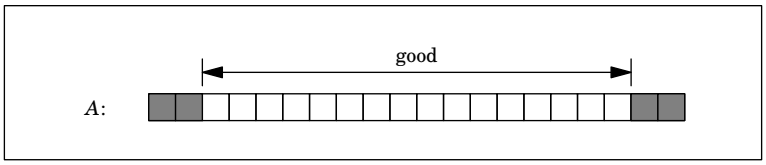

Figure 8.1. Wouldn't it be nice if we had some way of choosing an element that probably lies in the middle?

8.1.1 Randomized Median Finding

Let's put the elegant and simple algorithm FIND-APPROXIMATE-MIDDLE to use in a randomized version of the median-finding algorithm from Lecture 1:

Algorithm: RANDOMIZED-SELECT(A, i)

1. Pick an element $x \in A$ uniformly at random.
2. Partition around x. Let $k = \text{rank}(x)$.
3. - If $i = k$, then return x.
   - If $i < k$, then recursively call RANDOMIZED-SELECT$(A[1, \ldots, k-1],\ i)$.
   - If $i > k$, then recursively call RANDOMIZED-SELECT$(A[k+1, \ldots, i],\ i-k)$.

Note two things about this algorithm:

- Unlike FIND-APPROXIMATE-MIDDLE, RANDOMIZED-SELECT makes several random choices: one in each recursive call.
- Unlike FIND-APPROXIMATE-MIDDLE, RANDOMIZED-SELECT *is* correct. The effect of bad random choices is to prolong the running time, not to generate an incorrect answer. For example, if we wanted to find the middle element ($i = n/2$), and if our random element x happened to always be the smallest element of A, then RANDOMIZED-SELECT would take $\Theta(n) \cdot T_{\text{partition}} = \Theta(n^2)$ time. (To prove this rigorously, a more detailed calculation is needed.)

The latter point brings to light the fact that, for randomized algorithms, the notion of "worst-case" running time is more subtle than it is for deterministic algorithms. We cannot expect randomized algorithms to work well for every possible input *and* every possible sequence of random choices (in that case, why even use randomization?). Instead, we do one of two things:

- Construct algorithms that always run quickly, and return the correct answer with high probability. These are called **Monte Carlo algorithms**. With some small probability, they may return an incorrect answer or give up on computing the answer.
- Construct algorithms that always return the correct answer, and have low *expected* running time. These are called **Las Vegas algorithms**. Note that the expected running time is an average over all possible sequences of random choices, but *not* over all possible inputs. An algorithm that runs in expected $\Theta(n^2)$ time on difficult inputs and expected $\Theta(n)$ time on easy inputs has worst-case expected running time $\Theta(n^2)$, even if most inputs are easy.

FIND-APPROXIMATE-MIDDLE is a Monte Carlo algorithm, and RANDOMIZED-SELECT is a Las Vegas algorithm.

8.1.2 Running Time

To begin our analysis of RANDOMIZED-SELECT, let's first suppose all the random choices happen to be good. For example, suppose that every recursive call of RANDOMIZED-SELECT(A, i) returns an element whose rank is between $\frac{1}{10}|A|$ and $\frac{9}{10}|A|$. Then, the argument to each recursive call of RANDOMIZED-SELECT is at most nine-tenths the size of the argument to the previous call. Hence, if there are a total of K recursive calls to RANDOMIZED-SELECT, and if n is the size of the argument to the original call, then

$$\left(\tfrac{9}{10}\right)^K n \geq 1 \quad \Longrightarrow \quad K \leq \log_{10/9} n.$$

(Why?) Of course, in general we won't always get this lucky; things might go a little worse. However, it is quite unlikely that things will go *much* worse. For example, there is a 96% chance that (at least) one of our first 15 choices will be bad—quite high. But there is only a 13% chance that three of our first 15 choices will be bad, and only a 0.06% chance that nine of our first 15 choices will be bad.

Now let's begin the comprehensive analysis. Suppose the size of the original input is n. Then, let T_r be the number of recursive calls that occur after the size of A has dropped below $\left(\frac{9}{10}\right)^r n$, but before it has dropped below $\left(\frac{9}{10}\right)^{r+1} n$. For example, if it takes five recursive calls to shrink A down to size $\frac{9}{10}n$ and it takes eight recursive calls to shrink A down to size $\left(\frac{9}{10}\right)^2 n$, then $T_0 = 5$ and $T_1 = 3$. Of course, each T_i is a random variable, so we can't say anything for sure about what values it will take. However, we do know that the total running time of RANDOMIZED-SELECT$(A = A[1, \ldots, n], i)$ is

$$T = \sum_{r=0}^{\log_{10/9} n} \Theta\left(\left(\tfrac{9}{10}\right)^r n\right) \cdot T_r.$$

Again, T is a random variable, just as each T_r is. Moreover, each T_r has low expected value: Since each recursive call to RANDOMIZED-SELECT has a 4/5 chance of reducing the size of A by a factor of $\frac{9}{10}$, it follows that (for any r)

$$\Pr\left[T_r > s\right] \leq \left(\tfrac{1}{5}\right)^s.$$

Thus, the expected value of T_r satisfies

$$\mathbb{E}[T_r] \leq \sum_{s=0}^{\infty} s\left(\tfrac{1}{5}\right)^s = \frac{5}{16}.$$

(The actual value 5/16 is not important; just make sure you are able to see that the series converges quickly.) Thus, by the linearity of expectation,

$$\begin{aligned}
\mathbb{E}[T] &= \mathbb{E}\left[\sum_{r=0}^{\log_{10/9} n} \Theta\left(\left(\tfrac{9}{10}\right)^r n\right) \cdot T_r\right] \\
&= \sum_{r=0}^{\log_{10/9} n} \Theta\left(\left(\tfrac{9}{10}\right)^r n\right) \cdot \mathbb{E}[T_r] \\
&\leq \frac{5}{16} \sum_{r=0}^{\log_{10/9} n} O\left(\left(\tfrac{9}{10}\right)^r n\right) \\
&= O\left(n \sum_{r=0}^{\log_{10/9} n} \left(\tfrac{9}{10}\right)^r\right)
\end{aligned}$$

$$\leq O\left(n \sum_{r=0}^{\infty} \left(\tfrac{9}{10}\right)^r\right)$$

$$= O(n),$$

since the geometric sum $\sum_{r=0}^{\infty} \left(\frac{9}{10}\right)^r$ converges (to 10).

8.2 Another Example: Verifying Polynomial Multiplication

Suppose we are given two polynomials p, q of degree $n-1$ and a polynomial r of degree $2n-2$, and we wish to check whether $r = pq$. We could of course compute pq in $\Theta(n \lg n)$ time using the fast Fourier transform, but for some purposes the following Monte Carlo algorithm would be more practical:

Algorithm: VERIFY-MULTIPLICATION(p, q, r)

1. Choose x uniformly at random from $\{0, 1, \ldots, 100n-1\}$.
2. Return whether $p(x) \cdot q(x) = r(x)$.

This algorithm is not correct. It will never report no if the answer is yes, but it might report yes when the answer is no. This would happen if x were a root of the polynomial $pq - r$, which has degree at most $2n-2$. Since $pq - r$ can only have at most $2n-2$ roots, it follows that

$$\Pr[\text{wrong answer}] \leq \frac{2n-2}{100n} < 0.02.$$

Often, the performance improvement offered by this randomized algorithm (which runs in $\Theta(n)$ time) is worth the small risk of (one-sided) error.

8.3 The Markov Bound

The Markov bound states that it is unlikely for a nonnegative random variable X to exceed its expected value by very much.

Theorem 8.2 (Markov bound)**.** *Let X be a nonnegative random variable with positive expected value.*[1] *Then, for any constant $c > 0$, we have*

$$\Pr\left[X \geq c \cdot \mathbb{E}[X]\right] \leq \frac{1}{c}.$$

Proof. We will assume X is a continuous random variable with probability density function f_X; the discrete case is the same except that the integral is replaced by a sum. Since X is nonnegative, so is $\mathbb{E}[X]$. Thus we have

$$\mathbb{E}[X] = \int_0^{\infty} x f_X(x)\, dx$$

$$\geq \int_{c\mathbb{E}[X]}^{\infty} x f_X(x)\, dx$$

[1] i.e., $\Pr[X > 0] > 0$.

$$\geq c\,\mathbb{E}[X] \int_{c\,\mathbb{E}[X]}^{\infty} f_X(x)\,dx$$

$$= c\,\mathbb{E}[X] \cdot \Pr\Big[X \geq c\,\mathbb{E}[X]\Big].$$

Dividing through by $c\,\mathbb{E}[X]$, we obtain

$$\frac{1}{c} \geq \Pr\Big[X \geq c\,\mathbb{E}[X]\Big]. \qquad \square$$

Corollary 8.3. *Given any constant $c > 0$ and any Las Vegas algorithm with expected running time T, we can create a Monte Carlo algorithm which always runs in time cT and has probability of error at most $1/c$.*

Proof. Run the Las Vegas algorithm for at most cT steps. By the Markov bound, the probability of not reaching termination is at most $1/c$. If termination is not reached, give up. $\square$

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012