Lecture 11: Randomized Algorithms I

Lecture Overview

- Randomized Algorithms: What, Why, and How
- Gambling in Las Vegas vs Monte Carlo (vs Atlantic City)
- Matrix Product Verification
- Quickselect & Quicksort
- A Quick Refresher on Probability: Tail Inequalities
  (Markov, Central Limit, Chernoff; also union bound)

We're going to look at new and interesting ways to solve and analyze old problems, and introduce a little randomness to help us do things faster. (All it takes is a little.. push.)

1 Introduction

What is a randomized algorithm?

- An algorithm that generates a random number $r \in \{1, ..., R\}$ and makes decisions based on r's value. (You can also imagine making decisions based on coin flips – this sometimes simplifies analysis – but programmers generally just use a pseudorandom number generator seeded with time, etc.)
- Given the same input on different executions, a randomized algorithm may
  - Run a different number of steps (say, a loop that runs a random number of times)
  - Produce a different output (say, it randomly adds 1 or 0 to the output)

Do we want to write algorithms whose behavior depends on the outcome of a coin flip, basically gabling with our runtime and/or correctness? Yes! Yuuuge speedups. (And sometimes, there's no deterministic way to solve something, but a simple randomized algorithm will give you an answer arbitrarily close to the actual solution.)
Randomized algorithms can be broadly classified into **Monte Carlo** and **Las Vegas**.

Monte Carlo	Las Vegas
runs fast	probably runs fast
probably correct output	correct output

In other words, a Las Vegas algorithm does not gamble with the correctness of the result; it gambles only with the resources used for the computation. A Monte Carlo algorithm can run in a specified amount of time/resource usage. This will use bounded resources, but may produce an incorrect answer or abort (typically with some small probability).

We'll need some new methods to deal with time complexity and correctness in these setups. For randomized algorithms, the notion of "worst-case" running time is more subtle than it is for deterministic algorithms. You'll see 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. (You can think of it as the adversary controlling the input, but not your random choices, because even you don't know what you're going to do.)

2 Matrix Product Verification

$$C = A \times B$$

(These are $n \times n$ matrices.) Simple algorithm: $O(n^3)$ multiplications.
Strassen: multiply two 2×2 matrices in 7 multiplications: $O(n^{\log_2 7}) = O(n^{2.81})$
Coppersmith-Winograd: $O(n^{2.376})$

Note that we care only about multiplications and not additions (so, not the total number of operations). The reason is that multiplications, in computers, take longer than additions. Extra note: This difference used to be much more dramatic, but thanks to pipelining, lots of low-level optimizations, and GPU usage, multiplies are actually pretty quick now. But they are still quite a bit slower than addition.

Specifications for Our Algorithm

Given $n \times n$ matrices A, B, C, the goal is to check if $A \times B = C$ or not.
Question. Can we do better than carrying out the full multiplication?

We will see an $O(n^2)$ algorithm that:

- if $A \times B = C$, then $Pr[\text{output=YES}] = 1$.
- if $A \times B \neq C$, then $Pr[\text{output=YES}] \leq \frac{1}{2}$.

We will assume entries in matrices $\in \{0, 1\}$ and also that the arithmetic is mod 2.

Frievald's Algorithm

Choose a random binary vector $r[1...n]$ such that $Pr[r_i = 1] = 1/2$ independently for $r = 1, ..., n$. The algorithm will output 'YES' if $A(Br) = Cr$ and 'NO' otherwise.

Observation

The algorithm will take $O(n^2)$ time, since there are 3 matrix multiplications Br, $A(Br)$ and Cr of a $n \times n$ matrix by a $n \times 1$ matrix.

Analysis of Correctness if $AB \neq C$

Claim. If $AB \neq C$, then $Pr[ABr \neq Cr] \geq 1/2$.

Let $D = AB - C$. Our hypothesis is thus that $D \neq 0$. Clearly, there exists r such that $Dr \neq 0$. Our goal is to show that there are many r such that $Dr \neq 0$. Specifically, $Pr[Dr \neq 0] \geq 1/2$ for randomly chosen r.

$D = AB - C \neq 0 \implies \exists\ i, j$ s.t. $d_{ij} \neq 0$. Fix vector v which is 0 in all coordinates except for $v_j = 1$. $(Dv)_i = d_{ij} \neq 0$ implying $Dv \neq 0$. Take any r that can be chosen by our algorithm. We are looking at the case where $Dr = 0$. Let

$$r' = r + v$$

Since v is 0 everywhere except v_j, r' is the same as r exept $r'_j = (r_j + v_j) \mod 2$. Thus, $Dr' = D(r + v) = 0 + Dv \neq 0$. We see that there is a 1 to 1 correspondence between r and r', as if $r' = r + V = r'' + V$ then $r = r''$.

Basically, for any "bad" value of r, there's always another "good" r. This implies that

$$\text{number of } r' \text{ for which } Dr' \neq 0 \geq \text{number of } r \text{ for which } Dr = 0$$

From this we conclude that $Pr[Dr \neq 0] \geq 1/2$.

3 Radomized selection: Quickselect

We did some pretty complicated stuff for median finding before. Here's a simple algorithm for finding an element with rank "sufficiently close" to $n/2$ in an unsorted array A:
Input: Array $A = A[1, \ldots, n]$ of n integers
Output: An element $x \in A$ s.t. $\frac{1}{10}n \leq rank(x) \leq \frac{9}{10}n$
Our algorithm: Pick an element from A uniformly at random.
This algorithm returns a correct answer with probability 0.8. (Monte Carlo!)

So let's try to do something a bit more serious, which takes an array A, a rank i, and returns the value at that rank.
RANDOMIZED-SELECT(A, i):

1. Pick an element $x \in A$, called the *pivot*, uniformly at random
2. Partition around x. Let $k = rank(x)$
   (values greater than x get moved to the right, and those less than x get moved to the left of x, leaving x in the index corresponding to its rank.)

3. If $i = k$, return x
   If $i < k$, recursively call RANDOMIZED-SELECT$(A[1, \ldots, k-1], i)$
   If $i > k$, recursively call RANDOMIZED-SELECT$(A[k+1, \ldots, i], i-k)$

Notice how bad random choices will simply mean longer running times. *E.g.* from the point of view of finding the median, a value which partitions A into very uneven portions will slow us down. You can see how, if we wanted to find the element with rank 1, and our random selection always found the largest element remaining, we'd end up with something like $\Theta(n^2)$ time. (Since we're dealing with random choices here, we'll need a little more math to say this rigorously for the expected worst case, but this is clearly a Las Vegas algorithm.)

Analysis

At first, we shall assume that our partitions are magically good, in the sense that our randomly chosen x has $\frac{1}{10}n \leq rank(x) \leq \frac{9}{10}n$. With this heroic assumption, we can say that the number of calls to RANDOMIZED-SELECT, $K \leq \log_{10/9} n$ (you can also think of this as the height of the recursion tree).

So, on the first recursion $r = 0$, we'll do n work. Then we'll do $\frac{9}{10}n$ work (assuming we're on the "bad" side of the partition). The next time $r = 2$, we'll do $(\frac{9}{10})^2 n$, etc. This happens upto #max-height-of-recursion-tree times. So, we could write:

$$T = \sum_{r=0}^{\log_{10/9} n} \Theta((\frac{9}{10})^r n)$$

I'll leave it to you to satisfy yourself that this converges, but let's fix our little assumption. Let's say that we can't actually get A down to $\frac{9}{10}^{ths}$ the size every single time. Let's say that it sometimes take 5 times, sometimes 10, etc. Let's call that random variable T_r. This is the number of recursive calls that occur between $|A| = (\frac{9}{10})^r n$ and $|A| = (\frac{9}{10})^{r+1} n$. Then we could say:

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

But how much is that in expectation? Recall that randomly picking an element gives us a 0.8 chance of getting a "good" partition. Then, $Pr[T_r > s] \leq 0.2^s$. Then, the expected value of $T_r \leq \sum_{s=0}^{\infty} s \cdot (\frac{1}{5})^s = \frac{5}{16}$. You don't need to do the calculation, just convince yourself that the series converges. If we had chosen our definition of "good" to be a 3/4 partition, then we'd have a 1/2 chance of getting a good element on a random pick, which would give us an expected value of 2 here.

Then, by linearity of expectation,

$$\mathbb{E}(T) = \mathbb{E}(\sum_{r=0}^{\log_{10/9} n} \Theta((\frac{9}{10})^r n) \cdot T_r) = \sum_{r=0}^{\log_{10/9} n} \Theta((\frac{9}{10})^r n) \cdot \mathbb{E}(T_r)$$
$$\leq \frac{5}{16} \sum_{r=0}^{\log_{10/9} n} \Theta((\frac{9}{10})^r n) = O(\sum_{r=0}^{\log_{10/9} n} \Theta((\frac{9}{10})^r n)) = O(n)$$

4 Quicksort

[Tony Hoare, 1959] This is a divide and conquer algorithm, but unlike most others, the bulk of the work is done in the divide step rather than combine. This is "in place" like insertion sort and unlike mergesort (which requires $O(n)$ auxiliary space – there are in-place variants of mergesort, but those are pretty tricky). In practice, efficient implementations of quicksort run 2-5 times faster than merge- or heap-sort.

We'll talk about some quicksort variants: Basic (good in average case), Median-based pivoting (uses median finding, disgustingly complicated), and Random (good for all inputs in expectation – Las Vegas algorithm).

QUICKSORT(A):

- Divide: Choose a "pivot" element $x \in A$ uniformly at random.
- Divide contd.: Partition A around x. (*i.e.*, we have the left sub-array L, consisting of all elements $< x$, and the right sub-array G, consisting of all elements $> x$)
- Conquer: recursively sort subarrays L and G
- Combine: trivial

Basic Quicksort

No randomization: Pivot around $x = A[1]$ or $A[n]$ (first or last element). To do this in place, see CLRS p. 171.

Analysis: If the input is sorted or reverse sorted, we are partitioning around the min or max element each time. This means one of L or G has $n-1$ elements, and the other 0. This gives:

$$\begin{aligned} T(n) &= T(0) + T(n-1) + \Theta(n) \\ &= \Theta(1) + T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

However, this algorithm does well on random inputs in practice – as you might imagine, the header in a random input will probably not be too bad a pivot. (Think about what we saw in quickselect.)

Pivot Selection Using Median Finding

We can guarantee balanced L and G using a rank/median selection algorithm that runs in $\Theta(n)$ time.

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) + \Theta(n)$$
$$T(n) = \Theta(n \log n)$$

This algorithm is slow in practice and loses to mergesort. Actually, it loses to pretty much everything given how much time it takes to debug.

Randomized Quicksort

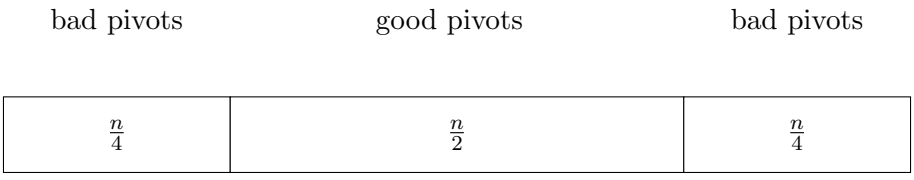
Going back to what we said in the beginning, x is chosen uniformly at random from array A (a random choice is made at each recursion). Expected time is $O(n \log n)$ for all input arrays A. See CLRS p.181-184 for the analysis of this algorithm; we will analyze a variant of this, and, if we have time, we'll talk about some probabilistic wizardry.

"Paranoid" Quicksort

Repeat
    choose pivot $x \in A$ u.a.r.
    perform Partition
Until resulting partition is such that $|L| \leq \frac{3}{4}|A|$ and $|G| \leq \frac{3}{4}|A|$
Recurse on L and G

"Paranoid" Quicksort Analysis

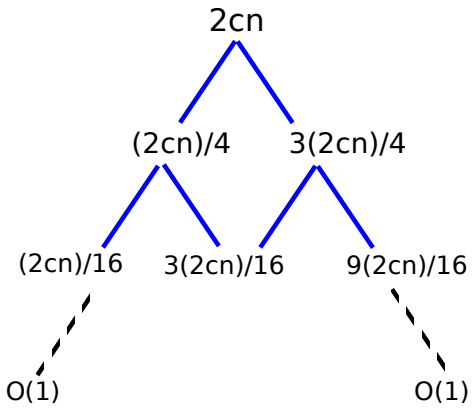
Just for fun, we're using a 3/4 partition rather than a 9/10 partition to define a "good pivot":



A pivot is good with probability $> 1/2$. Let $T(n)$ be an upper bound on the expected running time on any array of n size. T(n) comprises:

- time needed to sort left subarray
- time needed to sort right subarray
- the number of iterations to get a good call $c \cdot n$ (cost of partition)

Expectations



$$T(n) \leq max_{n/4 \leq i \leq 3n/4}(T(i) + T(n-i)) + E(\#\text{iterations}) \cdot cn$$

Now, since probability of good pivot $> \frac{1}{2}$,

$$E(\#iterations) \leq 2$$

$$T(n) \leq T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + 2cn$$

We see in the figure that the height of the tree can be at most $\log_{\frac{4}{3}}(2cn)$ no matter what branch we follow to the bottom. At each level, we do a total of $2cn$ work. Thus, expected runtime is $T(n) = \Theta(n \log n)$.

5 Some Useful Probability Theory

For all the formulae given below, you are only expected to be able to apply them at a superficial level. While we encourage you to delve more deeply into each of these (wonderful) inequalities and understand them properly, we only expect you to apply things like Markov and Chernoff in rudimentary ways.

The Markov bound, named after the Russian mathematician Andrey Markov, states that it is unlikely for a nonnegative random variable X to exceed its expected value by very much. (Of course, given Stigler's Law of Eponymy, it was first published by Markov's probability teacher, Pafnuty Chebyshev. Recursively, there's something we call Chebyshev's inequality which was invented by someone else.)

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

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

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 in the following is replaced by a sum. Since X is nonnegative, so is $\mathbb{E}[X]$. Thus we have

$$\mathbb{E}[X] = \int_0^\infty x f_X(x)dx \geq \int_{c\mathbb{E}[X]}^\infty x f_X(x)dx \geq c\mathbb{E}[X] \int_{c\mathbb{E}[X]}^\infty f_X(x)dx = c\mathbb{E}[X] \cdot Pr[X \geq c\mathbb{E}[X]]$$

Dividing through by $c\mathbb{E}[X]$, we get that $\frac{1}{c} \geq Pr[X \geq c \cdot \mathbb{E}[X]]$. □

Corollary 1. *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. (Recall that Monte Carlo may produce an incorrect answer or abort.) □

Theorem 2. *(Central Limit Theorem) We start with independent and identially distributed random variables $X_1, X_2, \ldots$, each with a finite mean μ and finite variance σ^2. Let $S_n = \frac{X_1+\cdots+X_n}{n}$ and $Y_n = \sqrt{n}(S_n - \mu)$. Then the variables $Y_1, Y_2, \ldots$ converge to a normal distribution:*

$$Y_n \to^d N(0, \sigma^2)$$

You don't need to study the CLT in detail, but you will be using Chernoff bounds (below) in class, psets, and quizzes.

Imagine you have a sample (a large number of observations, each observation being randomly generated in a way that does not depend on the values of the other observations) and that the arithmetic average of the observed values is computed. If we take multiple such samples, the central limit theorem says that the computed values of the average will be distributed according to the normal distribution or bell curve. E.g., flip a coin a large number of times, compute the number of heads and plot it. The probability of getting a given number of heads in a series of flips should follow a normal curve, with mean equal to half the total number of flips in each series.

While this heuristic reasoning does not rigorously prove any specific bound, there are a number of bounds that have been worked out for certain common distributions of X. One such bound is the Chernoff bound, one version of which is as follows:

Theorem 3. *(Chernoff bound). Let $Y = B(n, p)$ be a random variable representing the total number of heads in a series of n independent coin flips, where each flip has probability p of coming up heads. Then, for all $r > 0$, we have*

$$Pr[Y \geq \mathbb{E}[Y] + r] \leq e^{\frac{-2r^2}{n}}$$

The distribution above is called the binomial distribution with parameters (n, p). There are several different versions of the Chernoff bound – some with better bounds for specific hypotheses, some more general. Section C.5 of CLRS gives a classic proof of the Chernoff bound, using the Markov inequality.

Quicksort analysis using Chernoff bound

Why give you these complicated definitions? Because you can use them to prove all sorts of interesting things about randomized algorithms. Let's try and apply this new knowledge to quicksort. Hopefully, you've convinced yourself that quicksort runs in expected $\Theta(n \log n)$. Let's see what else we can prove!

Let $X_{i,k}$ be the indicator random variable for pivot quality (we're using a 9/10ths split for a good pivot). $X_{i,k} = 0$ if, a good pivot was chosen for a_i's subarray in the kth iteration / at recursive depth k (1 otherwise). $\mathbb{E}[X_{i,k}] = 0.2$. (We're assuming that you keep pivoting even after you reach size 1 with 0.8 probability of finding a good pivot – this is ridiculous, but reduces the complexity of our calculation, so we'll do it.) Note that Xs are independent for a given i. Given a recursion depth K, for any i,

$$\mathbb{E}[\sum_{k=1}^{K} X_{i,k}] = 0.2K$$

We can apply the Chernoff bound here, with $r = 0.1K$, to get:

$$Pr[\sum_{k=1}^{K} X_{i,k} > 0.2K + 0.1K] \leq e^{\frac{-2\cdot(0.1K)^2}{K}} = e^{0.02K}$$

Setting $K = 100 \ln n$ gives us $Pr[\sum_{k=1}^{K} X_{i,k} > 0.3K] \leq \frac{1}{n^2}$. If the value on the left is $\leq 0.3K$, then that means that at least $0.7K$ of the first K pivots were good, then the size of the working subarray containing a_i after K steps is at most

$$n \cdot (\frac{9}{10})^{0.7K} = n \cdot (\frac{9}{10})^{70 \ln n} = n \cdot n^{70 \ln(9/10)} \approx n^{-6.37} < 1$$

(*i.e.*, the size is at most 1; the reason for the fractional number is that we are ignoring issues of rounding). Thus, after $K = 100 \ln n$ iterations, the probability that the working subarray containing a_i has more than one element is at most $\frac{1}{n^2}$. So by the union bound, the probability that quicksort requires more than K iterations is at most $1/n$.

Recall: In general, the union bound says that the probability of a finite or countably infinite union of events E_i is at most the sum of their individual probabilities. The intuition is that equality is achieved when the E_i's are pairwise disjoint, and any overlap only diminishes the size of the union.

We've shown that that after time $cn \log n$, the probability that "the working subarray containing any given element a $a \in A$ consists of more than one element" is at most $\frac{1}{n^2}$. We can then use the unon bound to show that the probability that there exists any working subarray with more than one element is at most $\frac{1}{n}$. That is, we let E_i = the event that after time $cn \log n$, the wokring subarray containing a_i has more than one element. (We bounded this earlier.) Then,

$$Pr[\cup_{i=1}^{n} E_i] \leq \sum_{i=1}^{n} Pr[E_i] \leq n \cdot \frac{1}{n^2} = \frac{1}{n}$$