Recap: Lectures 1 & 2: Randomized Algorithms

Note: the purpose of these notes is to help recap what we did in the lecture. They are not designed to be exhaustive lecture notes.

Randomized Algorithms: what are they?

Standard algorithms for problems (e.g., merge sort, graph search / shortest paths, basic dynamic programming) are all *deterministic*. This means that given an input, the entire execution of the algorithm is pre-determined (or fixed). Randomized algorithms differ in this basic sense: they use *random bits* in the execution of the algorithm to make decisions. Our goal here is to see what extra power this simple change gives.

Example 1. Given an array of integers $a_1, a_2, \ldots, a_n$, and the promise that at least half of the array elements are 0, find one index j such that $a_j = 0$.

We called this problem, *finding hay in a haystack*.

To solve this deterministically, one has to go over the entire array, so it takes $O(n)$ time. In fact, any deterministic algorithm must take roughly n time, because it has to examine the elements in a pre-determined order (possibly depending on the previously seen elements), so an adversary can create an input on which the algorithm takes ~ (n/2) time steps.

However, a randomized algorithm for this problem is easy:

1. Pick an index j uniformly at random from $[n]$ (shorthand for $\{1, 2, \ldots, n\}$)
2. If $a_j = 0$ return j, else go back to step (1).

How long does this procedure take? In principle, it could run forever, because the algorithm could always find an index j with $a_j \neq 0$. However, it is easy to see that the probability that it runs for t steps without returning a j is $\leq (1/2)^t = 2^{-t}$.

In this case, the *running time* of the algorithm depends on the random choices made by the algorithm. So it is **random variable**. If we call this X, we can actually compute $\mathbb{E}[X]$, the expected value.

It is a simple calculation (if you do not remember it, look up Geometric Random Variables) to see that $\mathbb{E}[X] = 1/p$, where p is the exact fraction of 0s in the array. We are given that $p \geq 2$, so $\mathbb{E}[X] \leq 2$. In other words, the *expected* running time of the algorithm is ≤ 2. The actual running time can be greater, but the earlier observation says that, for example, the probability of the running time being > 50 is $< 2^{-50}$, which is extremely small.

This example, of course, seems (and is) trivial. But surprisingly, many randomized algorithms have this flavor!

Example 2. (Checking matrix multiplication) Given three matrices A, B, C (all $n \times n$), check if $C = AB$.

The obvious way to do it is to first compute the product AB and then check if it equals C. This can take time as large as n^3 if one uses the naive algorithm for matrix multiplication, or perhaps $n^{2.35}$ if one uses the super-state-of-the-art algorithm. However, it is still much worse than n^2. Can *checking* be done faster? (After all, no one is asking us to compute AB.)

- Main idea: hit LHS and RHS with a random vector x (assume a binary vector)
- Observe that Cx and ABx can be computed in $O(n^2)$ time
- Prove that equality $Cx = ABx$ holds with probability $\leq 1/2$. Proof uses the fact that given two unequal *vectors* a, b, the probability that $\langle a, x \rangle = \langle b, x \rangle$ is at most $1/2$. (Special case of the Schwartz-Zippel Lemma.)
- Repeat this process 20 (or whatever you wish) times. If none of the checks fail, declare that $C = AB$, else return false.

In the two examples, we see two slightly different phenomena: in the first one, the algorithm always returns the right answer, but the running time is probabilistic. In the second, the algorithm can return an incorrect answer (it can falsely conclude that $C = AB$), but the running time is fixed.

- Amplification of success probability

Wednesday (1/9):

We discussed two main examples: QuickSort, and Estimation via random sampling.

- Classic quicksort algorithm
- The running time is a random variable
  - Unlucky choice of pivot can result in lopsided subproblems --> $O(n^2)$ time
  - Lucky choice results in two subproblems of size $n/2$. This results in a recurrence $T(n) = 2T(n/2) + O(n)$, which solves to $n \log n$
- What is the typical running time?
- We can try to compute the expectation of the running time in the standard way ($\sum_{t \geq 0} \Pr[$ run-time = t]). But this is infeasible because estimating the probability is very messy. Instead, we can write down a recurrence for the expected running time.
- Using this idea, one can show that the expected running time is $2n \log n$.
- The next question is, is this good enough? Is the running time "with high probability" close to $2n \log n$, or can it be n^2 say 50% of the time?
- Markov's inequality
- It's the most basic (and also fundamental) example of a *concentration* inequality (people also call this the "concentration of measure phenomenon")

- Markov's inequality is good, but (as is) it is quite weak. For example, if we want to have a limit of say $40n \log n$ on the running time, Markov's inequality bounds the failure probability by $2/40 = 1/20$. However, if we do a more hacky thing: we stop the QuickSort after $4n \log n$ steps (by Markov's inequality, this has at least a (1/2) probability of success), and then repeat ten times, the failure probability drops to $(1/2)^{10}$ which is about 1 in a thousand!

Estimation by sampling

TODO