CS124 **Lecture 13** **Spring 2015**

13.1 Randomized Algorithms

A *randomized* algorithm is an algorithm that, during its execution, is allowed to toss fair coins and make decisions based on the outcomes of the tosses. In this lecture we looked at two types of randomized algorithms:

- **Las Vegas:** A Las Vegas algorithm is always correct, but its running time is a random variable. Often we try to bound its expected running time, or to show that its running time is is small with high probability.
- **Monte Carlo:** Such an algorithm has a fixed running time, but its correctness is random. The goals are thus two-fold: (1) obtain a small running time, but also (2) upper bound the probability that the algorithm outputs an incorrect answer. Typically we have some target failure probability P and would like the running time to be small both in terms of the problem input size as well as $1/P$.

Note that, just as in previous algorithms analyzed in this class, we would like randomized algorithms with *worst-case* guarantees (i.e. guarantees that apply to every input). Random behavior, such as outputting incorrect answers or having randomly long running times, occurs due to the internal coin tosses used in the algorithm's decision-making, and *not* due to randomness in the input to the algorithm. For all problems discussed in this lecture, *the algorithm's input is worst-case and not random.*

13.2 Short probability review

It is assumed that you are familiar with basic probability in this course, but we include a very basic refresher here.

Definition 13.1 *Let $S \subset \mathbf{R}$ be a discrete set of numbers. Then a* random variable *X supported on S can be described by $|S|$ numbers $\{\mathbb{P}(X = s)\}_{s \in S}$. These numbers satisfy $0 \le \mathbb{P}(X = s) \le 1$ for each $s \in S$, and furthermore*

$$\sum_{s \in S} \mathbb{P}(X = s) = 1.$$

We verbalize $\mathbb{P}(X = s)$ as "the probability X equals s".

Definition 13.2 *Given a random variable X supported on S, its* expectation $\mathbb{E}X$ *is defined as*

$$\mathbb{E}X = \sum_{s \in S} s \cdot \mathbb{P}(X = s).$$

For example, we may have a fair six-sided die, and X may be the random variable denoting the outcome of a die roll. Then $\mathbb{P}(X = i)$ is $1/6$ for $i = 1, 2, \ldots, 6$, and

$$\mathbb{E}X = \sum_{i=1}^{6} i \cdot \mathbb{P}(X = i) = \sum_{i=1}^{6} i \cdot \frac{1}{6} = 3.5$$

Lemma 13.3 (Linearity of expectation) *For any $a, b \in \mathbf{R}$ and any random variables X, Y each supported on S,*

$$\mathbb{E}(aX + bY) = a \cdot \mathbb{E}X + b \cdot \mathbb{E}Y.$$

Proof:

$$\begin{aligned}
\mathbb{E}(aX + bY) &= \sum_{x \in S} \sum_{y \in S} (ax + by) \cdot \mathbb{P}(X = x, Y = y) \\
&= \left[a \sum_{x \in S} x \sum_{y \in S} \mathbb{P}(X = x, Y = y) \right] + \left[b \sum_{y \in S} y \sum_{x \in S} \mathbb{P}(X = x, Y = y) \right] \\
&= \left[a \sum_{x \in S} x \cdot \mathbb{P}(X = x) \right] + \left[b \sum_{y \in S} y \cdot \mathbb{P}(Y = y) \right] \\
&= a \cdot \mathbb{E}X + b \cdot \mathbb{E}Y.
\end{aligned}$$

■

Lemma 13.4 (Markov's inequality) *If X is a nonnegative random variable, then for any $\lambda > 0$*

$$\mathbb{P}(X > \lambda \cdot \mathbb{E}X) < \frac{1}{\lambda}.$$

Proof: Let S be the support of X, where every entry in S is nonnegative. Then

$$\begin{aligned}
\mathbb{E}X &= \sum_{x \in S} x \cdot \mathbb{P}(X = x) \\
&\geq \sum_{\substack{x \in S \\ x > \lambda \cdot \mathbb{E}X}} x \cdot \mathbb{P}(X = x) \\
&> \sum_{\substack{x \in S \\ x > \lambda \cdot \mathbb{E}X}} \lambda \cdot (\mathbb{E}X) \cdot \mathbb{P}(X = x) \\
&= \lambda \cdot (\mathbb{E}X) \cdot \mathbb{P}(X > \lambda \cdot \mathbb{E}X),
\end{aligned}$$

from which the lemma holds by dividing through. ■

We leave the proof of the following lemma as an exercise.

Lemma 13.5 *If X is supported on the natural numbers $0,1,2\dots$, then*

$$\mathbb{E}X = \sum_{k=0}^{\infty} \mathbb{P}(X > k)$$

Corollary 13.6 *Suppose we repeatedly flip a coin which comes up heads with probability p and tails with probability $1-p$. Let X be the number of times we flip the coin before obtaining heads for the first time. Then $\mathbb{E}X = 1/p$.*

Proof: By Lemma 13.5

$$\begin{aligned}\mathbb{E}X &= \sum_{k=0}^{\infty} \mathbb{P}(X > k)\\ &= \sum_{k=0}^{\infty} (1-p)^k\\ &= \frac{1}{1-(1-p)},\end{aligned}$$

which is $1/p$ as desired. ■

13.3 Freivalds' algorithm

Suppose that we have two $n \times n$ matrices A, B that we would like to multiply. The straightforward algorithm would take $\Theta(n^3)$ time. As we saw with Strassen's algorithm, it is possible to achieve $o(n^3)$ time, but even the best known algorithms to date do not get down to $O(n^2)$ time. Suppose that n is large enough that we are comfortable spending $O(n^2)$ time worth of computation, but not much more than that. Meanwhile, some cloud service company with many more powerful machines than we have at our disposal is willing to accept A, B from us, run some algorithm to compute $C = A \times B$ on their machine, then return C back to us. We are unsure of exactly what algorithm that cloud service is running (maybe their code is buggy? or malicious?), so we would like to verify that the C they gave us actually does equal $A \times B$.

Thus in this problem, we ask ourselves: given $n \times n$ matrices A, B, C, is there an easier way to verify that $C = A \times B$ than actually computing $A \times B$? Note that simply sampling entries of C to check that $C_{i,j} = (A \times B)_{i,j}$ for a few different (i, j) pairs won't work: it may be that C equals $A \times B$ except for one entry, and we'll miss that unless we basically check every entry.

The idea is as follows: rather than multiply $A \times B$, we pick some $x \in \mathbf{R}^n$ and check that $Cx = ABx$. By associativity we can compute ABx as $A(Bx)$, i.e. a sequence of two matrix-vector multiplications, and thus this check can be performed in time $O(n^2)$. Unfortunately, in general it does not work though. If indeed $C = A \times B$, then we will have $Cx = ABx$. But this is not an "if and only if"; it can happen that $Cx = ABx$ even though $C \neq AB$. For example, for $D = C - AB$, consider

$$D = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}, \; x = \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix}$$

so that $Dx = 0$, but clearly $D \neq 0$.

The idea behind Freivalds' algorithm is to choose x *randomly*. That is, we will choose $x \in \{0,1\}^n$ to be a random binary vector. In fact we will do this k times, picking $x^1, x^2, \ldots, x^k$ independently and uniformly at random from $\{0,1\}^n$. If $Cx^i = ABx^i$ for all $i = 1,2,\ldots,k$ then we output "$C = AB$". If there exists at least one i such that $Cx^i \neq ABx^i$, then we output "$C \neq AB$". Clearly if indeed C does equal AB, then we will not err. However if C does not equal AB, then it is possible for us to be fooled k times in a row and output the wrong answer. Thus, Freivalds' algorithm is a Monte Carlo algorithm (the running time is deterministic: it is $\Theta(kn^2)$.

Lemma 13.7 *Suppose $C \neq AB$. If $x \in \{0,1\}^n$ is a binary vector chosen uniformly at random, then $\mathbb{P}(Cx \neq ABx) \geq$ 1/2.*

Proof: Let $D = C - AB$ so we want to show $\mathbb{P}_x(Dx \neq 0) \geq 1/2$. Since D is not the zero matrix, in particular there is some $1 \leq j \leq n$ such that D has a non-zero entry in column j. For any vector x, we define x' as being the vector obtained from x by flipping the jth bit (thus either $x' = x + e_j$ or $x' = x - e_j$, where e_j is the jth standard basis vector with a 1 in the jth coordinate and zeroes elsewhere). Now, note that $(x')' = x$. Thus if we define a graph G whose vertices correspond to $\{0,1\}^n$ and draw an edge between x and x' for every x, then G is what's called a *perfect matching*. That is to say, every vertex has an edge to exactly one other vertex: its "match partner".

Now we claim that if x, x' are match partners in G then it cannot be the case that both $Dx = 0$ and $Dx' = 0$ occur. That is to say, at least one of $Dx \neq 0$ or $Dx' \neq 0$ must hold. Note that this would imply the lemma, since this would mean that at least half of all vectors z, $Dz \neq 0$. So, why is it that $Dx = Dx' = 0$ is impossible? This is because if $Dx = 0$, then $Dx' = D(x \pm e_j) = Dx \pm De_j = \pm De_j$. But De_j is just the jth column of D, which we said was not zero. ■

Corollary 13.8 *For any $0 < P < 1$, Freivalds' algorithm can be used to obtain an algorithm for the matrix multiplication verification problem with running time $\Theta(n^2 \log(1/P))$ such that (1) if $C = AB$, then we will say so, and (2) if $C \neq AB$, then the probability that the algorithm errs is at most P.*

Proof: The runtime of Freivalds' algorithm is $\Theta(kn^2)$. The probability of failure if $C \neq AB$ is the probability that each of $x^1, \ldots, x^k$ satisfies $Dx^i = 0$. Each of these happens with probability at most $1/2$, and thus the probability that this happens k times in a row is at most $1/2^k$, which is at most P if we choose $k = \lceil \log_2(1/P) \rceil$. ■

13.4 QuickSort

The goal here is to sort n elements in an array $A[1 \ldots n]$. We will assume that the $A[i]$ are distinct. We know how to do this in $\Theta(n \log n)$ time using MergeSort, but here we will give another approach, QuickSort will be a Las Vegas algorithm with $\Theta(n \log n)$ expected running time.

The idea behind QuickSort is simple: when given an array A, we pick some pivot element $A[p]$. We then compare $A[p]$ with $A[i]$ for every other i and partition the indices into two sets: $S_L = \{i : A[i] < A[p]\}$, and $S_r = \{i : A[i] > A[p]\}$. We move those elements with indices in S_L to the left part of the array, followed by $A[p]$, followed by the elements with indices in S_R to the right part of the array. Then we recursively sort left and right subarrays.

How do we pick the pivot element? In the worst case the running time of QuickSort could be terrible, if the pivot happens to be the absolute smallest element for example. In this case, we would have to recursively sort an array of size $n-1$, yielding the running time recurrence $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$. The ideal case is that the pivot is the median element: smaller than half the elements in A, and also bigger than the other half. Then we would obtain $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$. The idea behind QuickSort is to pick the pivot *randomly*. The intuition is that, at least on average, the pivot element should split our recursions into two subarrays that are each of size roughly $n/2$.

How do we formalize this? For $1 \leq i < j \leq n$, let $X_{i,j}$ be an indicator random variable which is 1 if the ith smallest and jth smallest elements of A (from now on we just refer to these items i, j) are *ever* compared over the course of QuickSort's execution, and let $X_{i,j} = 0$ otherwise. Note that QuickSort's running time is proportional to the total number of comparisons it makes, so the running time of QuickSort is proportional to the random variable $\sum_{1 \leq i < j \leq n} X_{i,j}$.

Theorem 13.9 *The expected running time of QuickSort on an n-element array is* $\Theta(n \log n)$.

Proof: As stated above, the running time of QuickSort is within a constant factor of $\sum_{1 \leq i < j \leq n} X_{i,j}$. Then by linearity of expectation

$$\mathbb{E} \sum_{1 \leq i < j \leq n} X_{i,j} = \sum_{1 \leq i < j \leq n} \mathbb{E} X_{i,j} = \sum_{1 \leq i < j \leq n} \mathbb{P}(X_{i,j} = 1)$$

What is the probability that $X_{i,j} = 1$, i.e. that i, j are at some point compared during the execution of QuickSort? Starting at the root of the recursion tree, i, j (and all items in the interval $[i, j]$) are in the same subarray to be sorted. Working down the recursion tree, this continues to be true until finally the pivot which is chosen is some element in $[i, j]$. Now, if at that point the pivot is either i or j, then the two are compared. Otherwise, the two are never compared (they break into different recursive subtrees). Since the pivot is chosen uniformly at random, the probability that the pivot is either i or j is $2/(j-i+1)$. Thus, letting k denote $j-i$,

$$\begin{aligned}\sum_{1\le i<j\le n} \mathbb{P}(X_{i,j}=1) &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{2}{j-i+1}\\ &= 2\sum_{i=1}^{n}\sum_{k=1}^{n-i} \frac{1}{k+1}\\ &\le 2\sum_{i=1}^{n}\int_{1}^{n-i+1} \frac{1}{k}\\ &= 2\sum_{i=1}^{n} \ln(n-i+1)\\ &\le 2\sum_{i=1}^{n} \ln n\\ &= 2n\ln n\end{aligned}$$

The bound of a sum by an integral above follows by remembering that the integral of a function is the area under the curve when plotting that function, whereas the summation is the sum of areas of certain unit-width rectangles. Exercise: convince yourself with a picture (the curve drawn will always lie above the upper edges of the rectangles, and thus contain more area below them!). ■

13.5 QuickSelect

In the *selection* problem we are given an array $A[1\ldots n]$ and an integer $1 \le k \le n$ and would like to output the kth smallest value in the array A. Recall that in an earlier lecture we gave a $\Theta(n)$ time deterministic algorithm for this problem. The algorithm was recursive: given an array A it picked a pivot element x, partitioned the elements of A into two sets (those less than and those greater than x), then recursively called itself on the array containing the kth smallest element. The pivot was chosen carefully to ensure that the next recursive call was on a subarray of size at least a constant factor smaller than the original input, which guaranteed a linear overall running time.

In QuickSelect, the algorithm is the same except for the choice of pivot. Rather than the complicated recursive "median of medians" used in the deterministic algorithm, we simply pick the pivot to be a random element in A. Let us now argue that this algorithm runs in expected $O(n)$ time. As before, let us refer to the the ith item in the sorted order amongst elements in A as “the ith item”. We will be calling QuickSelect recursively on various subintervals of items. Initially we call QuickSelect on all items, which is the subinterval of items $S_0 = \{1,\ldots,n\}$. Let S_j be the set of items we recursively call our function on at recursive level j. Call a recursive level j “good” if $|S_{j+1}| \le (3/4)|S_j|$. Define $n_0 = n$, and for $i > 0$ let n_i denote $|S_{j_i+1}|$ where j_i is the index of the ith good level when considering levels in increasingly deeper levels of recursion (define $j_0 = 0$, the root level of recursion). Then $n_i \le (3/4)n_{i-1}$, which implies $n_i \le (3/4)^i n$ by induction on i. Now define the random variable $X_i = j_i - j_{i-1} + 1$, i.e. the number of recursive calls we go through after the $(i-1)$st good level before reaching the ith good level. Note there are at most $d = \lceil \log_{4/3} n \rceil$ good levels (since beyond that the set sizes are at most 1) , so the running time of QuickSelect is proportional to

$$|S_0| + |S_1| + \ldots \le \sum_{i=0}^{d} X_i n_i \le n \cdot \sum_{i=0}^{d} \left(\frac{3}{4}\right)^i X_i$$

Thus by linearity of expectation, the expected running time is at most

$$\mathbb{E}\left(n \cdot \sum_{i=0}^{d} \left(\frac{3}{4}\right)^i X_i\right) = n \cdot \sum_{i=0}^{d} \left(\frac{3}{4}\right)^i \mathbb{E}X_i$$

Note that a level is guaranteed to be good as long as the pivot is not in the bottom or top quartile, and thus any level is good with probability at least $1/2$. Thus by Corollary 13.6, $\mathbb{E}X_i \le 2$. Thus the above is at most

$$2n \cdot \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i = 8n.$$