Lecture 13 — Probability and Randomized Algorithms

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

Lectured by Kanat Tangwongsan — February 28, 2012

The main theme of this lecture is *randomized algorithms*. These are algorithms that make use of randomness in their computation. We will begin this lecture with a simple question:

> **Question:** How many comparisons do we need to find the second largest number in a sequence of n distinct numbers?

Without the help of randomization, there is a naïve algorithm that requires about $2n-3$ comparisons and there is a divide-and-conquer solution that needs about $3n/2$ comparisons. With the aid of randomization, there is a simple randomized algorithm that uses only $n-1+2\log n$ comparisons on average, under some notion of averaging. In probability speak, this is $n-1+2\log n$ comparisons in expectation. We'll develop the machinery needed to design and analyze this algorithm.

By the end of this lecture, you will be well-equipped to analyze the randomized quick select algorithm that finds k smallest elements in a sequence. This is what we covered recently in recitation. We will show that if we pick the pivot element randomly at every step, the algorithm has expected $O(n)$ work and $O(\log^2 n)$ span.

1 Discrete Probability: Let's Flip Some Coins

You probably have heard about probability a lot already. We will spend some time in this lecture reviewing basic probability concepts. To begin, let's suppose we have a *fair* coin, meaning that it is equally likely to land on either side. If we flip this coin, what is the chance that it is going to turn up heads? The answer, as we know already, is

$$\frac{\text{\# of outcomes that meet the condition}}{\text{\# of total outcomes}} = \frac{1}{2}$$

because in this case, these outcomes are equally likely to happen.

In a more formal setting, we have a *probability space*, which is made up of the following 3 components: First, we have a set of possible outcomes, known as a *sample space* Ω. Second, we have a family of sets $\mathcal{F}$ of allowable events, where each set in $\mathcal{F}$ is a subset of the sample space Ω. Third, we have a probability function $\mathbf{Pr} : \mathcal{F} \to [0,1]$ satisfying

1. $\mathbf{Pr}[\Omega] = 1$; and
2. for any finite (or countably infinite) sequence of disjoint events $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$,

$$\mathbf{Pr}\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right] = \sum_{i=1}^{n} \mathbf{Pr}\left[\mathcal{E}_i\right].$$

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

In the previous example, the sample space is $\{H, T\}$, denoting heads and tails respectively. The family of sets of allowable events $\mathcal{F}$ is simply $\{\{H\}, \{T\}\}$ because that coin has to come up either heads or tails, but not both at the same time.

Often, each outcome in the sample space is equally likely to happen. In this case, the probability of an event $\mathcal{E} \in \mathcal{F}$ can be found by counting the number of outcomes in the event and dividing by the total number of possible outcomes; that is,

$$\mathbf{Pr}\left[\mathcal{E}\right] = \frac{|\mathcal{E}|}{|\Omega|}.$$

Independence. Two events A and B are independent if and only if $\mathbf{Pr}\left[A \cap B\right] = \mathbf{Pr}\left[A\right] \cdot \mathbf{Pr}\left[B\right]$. When we have multiple events, we say that $A_1, \ldots, A_k$ are *mutually independent* if and only if for any non-empty subset $I \subseteq \{1, \ldots, k\}$,

$$\mathbf{Pr}\left[\bigcap_{i \in I} A_i\right] = \prod_{i \in I} \mathbf{Pr}\left[A_i\right].$$

1.1 Random Variables and Expectation

A *random variable* is a function $f : \Omega \to \mathbb{R}$. We typically denote random variables by capital letters $X, Y, \ldots$. That is, a random variable assigns a numerical value to an outcome $\omega \in \Omega$. For example, we could have a random variable X which will be 1 if the coin turns up heads and 0 otherwise. This particular type of random variables is called indicator random variables: for an event $\mathcal{E}$, the *indicator random variable* $\mathbb{I}\{\mathcal{E}\}$ takes on the value 1 if $\mathcal{E}$ occurs and 0 otherwise.

For a (discrete) random variable X and a number $a \in \mathbb{R}$, the event "$X = a$" is the set $\{\omega \in \Omega : X(\omega) = a\}$. Therefore,

$$\mathbf{Pr}\left[X = a\right] = \mathbf{Pr}\left[\{\omega \in \Omega : X(\omega) = a\}\right]$$

The *expectation* (or expected value) of a random variable is simply the weighted average of the value of the function over all outcomes. The weight is the probability of each outcome, giving

$$\mathbf{E}\left[X\right] = \sum_{\omega \in \Omega} X(\omega)\,\mathbf{Pr}\left[\omega\right] = \sum_{k} k \cdot \mathbf{Pr}\left[X = k\right]$$

in the discrete case. Applying this definition, we have $\mathbf{E}\left[\mathbb{I}\{\mathcal{E}\}\right] = \mathbf{Pr}\left[\mathcal{E}\right]$.

Example 1.1. *The expectation of the variable X representing the value of a fair die is*

$$\mathbf{E}\left[X\right] = \sum_{i=1}^{6} i\,\mathbf{Pr}\left[X = i\right] = \sum_{i=1}^{6} i \times \tfrac{1}{6} = \frac{7}{2}.$$

1.2 Linearity of Expectations

One of the most important theorem in probability is *linearity of expectations*. It says that given two random variables X and Y, $\mathbf{E}\left[X\right] + \mathbf{E}\left[Y\right] = \mathbf{E}\left[X + Y\right]$. It is really powerful because the theorem does not require the events to be independent.

1.3 Examples

Example 1.2. *Suppose we toss n coins, where each coin has a probability p of coming up heads. What is the expected value of the random variable X denoting the total number of heads?*

Solution I: We'll apply the definition of expectation directly. This will rely on our strength to compute the probability and the simplify the sum:

$$\begin{aligned}
\mathbf{E}[X] &= \sum_{k=0}^{n} k \cdot \mathbf{Pr}[X = k] \\
&= \sum_{k=1}^{n} k \cdot p^k (1-p)^{n-k} \binom{n}{k} \\
&= \sum_{k=1}^{n} k \cdot \frac{n}{k} \binom{n-1}{k-1} p^k (1-p)^{n-k} && \text{[because } \binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} \text{]} \\
&= n \sum_{k=1}^{n} \binom{n-1}{k-1} p^k (1-p)^{n-k} \\
&= n \sum_{j=0}^{n-1} \binom{n-1}{j} p^{j+1} (1-p)^{n-(j+1)} && \text{[because } k = j+1 \text{]} \\
&= np \sum_{j=0}^{n-1} \binom{n-1}{j} p^{j} (1-p)^{(n-1)-j} \\
&= np(p + (1-p))^n && \text{[Binomial Theorem]} \\
&= np
\end{aligned}$$

That was pretty tedious :(

Solution II: We'll use linearity of expectations. Let $X_i = \mathbb{I}\{i\text{-th coin turns up heads}\}$. That is, it is 1 if the i-th coin turns up heads and 0 otherwise. Clearly, $X = \sum_{i=1}^{n} X_i$. So then, by linearity of expectations,

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbf{E}[X_i].$$

What is the probability that the i-th coin comes up heads? This is exactly p, so $\mathbf{E}[X] = 0 \cdot (1-p) + 1 \cdot p = p$, which means

$$\mathbf{E}[X] = \sum_{i=1}^{n} \mathbf{E}[X_i] = \sum_{i=1}^{n} p = np.$$

Example 1.3. *A coin has a probability p of coming up heads. What is the expected value of Y representing the number of flips until we see a head? (The flip that comes up heads counts too.)*

Solution I: We'll directly apply the definition of expectation:

$$\begin{aligned}
\mathbf{E}[Y] &= \sum_{k\geq 1} k(1-p)^{k-1}p \\
&= p\sum_{k=0}^{\infty}(k+1)(1-p)^k \\
&= p\cdot\frac{1}{p^2} && \text{[by Wolfram Alpha, though you should be able to do it.]} \\
&= 1/p
\end{aligned}$$

Solution II: Alternatively, we'll write a recurrence for it. As it turns out, we know that with probability p, we'll get a head and we'll be done—and with probability $1-p$, we'll get a tail and we'll go back to square one:

$$\mathbf{E}[Y] = p\cdot 1 + (1-p)\Big(1+\mathbf{E}[Y]\Big) = 1 + (1-p)\mathbf{E}[Y] \implies \mathbf{E}[Y] = 1/p.$$

by solving for $\mathbf{E}[Y]$ in the above equation.

2 Finding The Top Two

The top-two problem is to find the two largest elements from a sequence of n (unique) numbers. For inspiration, we'll go back and look at the naïve algorithm for the problem:

```
Input: S
if S_1 > S_2 then
   max = S_1; max2 = S_2;
else
   max = S_2; max2 = S_1;
end

For i = 3 to n:
  if (S_i > max2) then
    if (S_i > max) then
       max2 = max; max = S_i
    else
       max2 = S_i;
    end
  end
```

In the following analysis, we will be meticulous about constants. The naïve algorithm requires about $2(n-2)+1 = 2n-3$ comparisons; we only compare the first two elements once. On the surface, this may seem like the best one can do. Surprisingly, there is a divide-and-conquer algorithm that uses only about $3n/2$ comparisons (exercise to the reader). More surprisingly still is the fact that it can be done in $n + O(\log n)$ comparisons. But how?

Puzzle: How would you solve this problem using only $n + O(\log n)$ comparisons?

A closer look at the analysis above reveals that we were pessimistic about the number of comparisons; not all element will get pass the "if" statement in Line 8; therefore, only some of the elements will need the comparison in Line 9. But we didn't know how many of them, so we analyzed it in the worst possible scenario.

Let's try to understand what's happening better by looking at the worst-case input. Can you come up with an instance that yields the worst-case behavior? It is not difficult to convince yourself that there is a sequence of length n that causes this algorithm to make $2n - 3$ comparisons. In fact, the instance is simple: an increasing sequence of length n, e.g., $\langle 1, 2, 3, \ldots, n\rangle$. As we go from left to right, we find a new maximum every time we counter a new element—this new element gets compared in both Lines 8 and 9.

But perhaps it's unlikely to get such a nice—but undesirable—structure if we consider the elements in random order. With only 1 in $n!$ chance, this sequence will be fully sorted. You can work out the probability that the random order will result in a sequence that looks "approximately" sorted, and it would not be too high. Our hopes are high that *we can save a lot of comparisons in Line 9 by considering elements in random order.*

The algorithm we'll analyze is the following. On input a sequence S of n elements:

1. Let $T = \texttt{permute}(S, \pi)$, where π is a random permutation (i.e., we choose one of the $n!$ permutations).
2. Run the naïve algorithm on T.

Remarks: We don't need to explicitly construct T. We'll simply pick a random element which hasn't been considered and consider that element next until we are done looking at the whole sequence. For the analysis, it is convenient to describe the process in terms of T.

In reality, the performance difference between the $2n - 3$ algorithm and the $n - 1 + 2\log n$ algorithm is unlikely to be significant—unless the comparison function is super expensive. For most cases, the $2n - 3$ algorithm might in fact be faster to due better cache locality.

The point of this example is to demonstrate the power of randomness in achieving something that otherwise seems impossible—more importantly, the analysis hints at why on a typical "real-world" instance, the $2n - 3$ algorithm does much better than what we analyzed in the worst case (real-world instances are usually not adversarial).

2.1 Analysis

Let X_i be an indicator variable denoting whether Line 9 gets executed for this particular value of i (i.e., Did T_i get compared in Line 9?) That is, $X_i = 1$ if T_i is compared in Line 9 and 0 otherwise.

Therefore, the total number of comparisons is

$$Y = \underbrace{1}_{\text{Line 2}} + \underbrace{n-2}_{\text{Line 8}} + \underbrace{\sum_{i=2}^{n} X_i}_{\text{Line 9}};$$

This expression in true regardless of the random choice we're making. We're interested in computing the expected value of Y where the random choice is made when we choose a permutation. By linearity of expectation, we have

$$\mathbf{E}[Y] = 1 + (n-2) + \sum_{i=2}^{n} \mathbf{E}[X_i].$$

Our tasks therefore boils down to computing $\mathbf{E}[X_i]$ for $i = 2, \ldots, n$. To compute this expectation, we ask ourselves: *What is the probability that $T_i >$ `max2`?* A moment's thought shows that the condition $T_i >$ `max2` holds exactly when T_i is either the largest element or the largest element in $\{T_1, \ldots, T_i\}$. So ultimately we're asking: what is the probability that T_i is the largest or the second largest element in randomly-permuted sequence of length i?

To compute this probability, we note that each element in the sequence is equally likely to be anywhere in the permuted sequence (we chose a random permutation. In particular, if we look at the k-th largest element, it has $1/i$ chance of being at T_i. (You should also try to work it out using a counting argument.) Therefore, the probability that T_i is the largest or the second largest element in $\{T_1, \ldots, T_i\}$ is $\frac{1}{i} + \frac{1}{i} = \frac{2}{i}$, so

$$\mathbf{E}[X_i] = 1 \cdot \tfrac{2}{i} = 2/i.$$

Plugging this into the expression for $\mathbf{E}[Y]$, we get

$$\begin{aligned}
\mathbf{E}[Y] &= 1 + (n-2) + \sum_{i=2}^{n} \mathbf{E}[X_i] \\
&= 1 + (n-2) + \sum_{i=2}^{n} \frac{2}{i} \\
&= 1 + (n-2) + 2\left(\tfrac{1}{2} + \tfrac{1}{3} + \ldots \tfrac{1}{n}\right) \\
&= n - 3 + 2\left(1 + \tfrac{1}{2} + \tfrac{1}{3} + \ldots \tfrac{1}{n}\right) \\
&= n - 3 + 2H_n,
\end{aligned}$$

where H_n is the n-th Harmonic number. But we know that $H_n \leq 1 + \log_2 n$, so we get $\mathbf{E}[Y] \leq n - 1 + 2\log_2 n$. We could also use the following sledgehammer:

As an aside, the Harmonic sum has the following nice property:

$$H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n} = \ln n + \gamma + \varepsilon_n,$$

where γ is the Euler-Mascheroni constant, which is approximately $0.57721\cdots$, and $\varepsilon_n \sim \frac{1}{2n}$, which tends to 0 as n approaches ∞. This shows that the summation and integral of $1/i$ are almost identical (up to constants and a low-order term).

3 Finding The Bottom k

Consider the following problem:

> **Input:** S — a sequence of n numbers (not necessarily sorted)
>
> **Output:** a sequence of the smallest k numbers (could be in any order).
>
> *Requirement:* $O(n)$ expected work and $O(\log^2 n)$ span.

Note that the linear-work requirement rules out the possibility of sorting the sequence. Here's where the power of randomization gives you a simple algorithm. For simplicity, we'll assume the elements are unique.

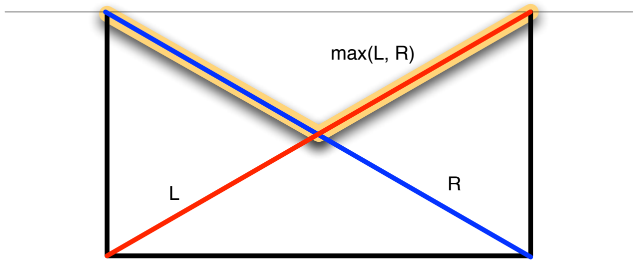
```
SmallestK (k, S) =
  1. If |S| <= k, return S.
  2. Pick p in S uniformly at random.
  3. Let L = { x in S: x <= p } and R = {x in S : x > p}
  4. If |L| >= k then return SmallestK(k, L)
     Else return append(L, SmallestK(k - |L|, R).
```

We'll try to analyze the work and span of this algorithm. Let $X_n = \max\{|L|, |R|\}$, which is the size of the larger side. Notice that X_n is an upper bound on the size of the side the algorithm actually recurses into. Now since Step 3 is simply two `filter` calls, we have the following recurrences:

$$\begin{aligned} W(n) &= W(X_n) + O(n) \\ S(n) &= S(X_n) + O(\log n) \end{aligned}$$

Let's first look at the work recurrence. Specifically, we are interested in $\mathbf{E}[W(n)]$. First, let's try to get a sense of what happens in expectation.

How big is $\mathbf{E}[X_n]$*?* To understand this, let's take a look at a pictorial representation:



The probability that we land on a point on the curve is $1/n$, so

$$\mathbf{E}[X_n] = \sum_{i=1}^{n-1} \max\{i, n-i\} \cdot \tfrac{1}{n} \le \sum_{j=n/2}^{n-1} \frac{2}{n} \cdot j \le \frac{3n}{4}$$

(Recall that $\sum_{i=a}^{b} i = \frac{1}{2}(a+b)(b-a+1)$.)

Aside: This is a counterexample showing that $\mathbf{E}[\max\{X,Y\}] \neq \max\{\mathbf{E}[X], \mathbf{E}[Y]\}$.

This computation tells us that in expectation, X_n is a constant fraction smaller than n, so we should have a nice geometrically decreasing sum, which works out to $O(n)$. Let's make this idea concrete: Suppose we want each recursive call to work with a constant fraction fewer elements than before, say at most $\frac{3}{4}n$.

What's the probability that $\mathbf{Pr}\left[X_n \le \frac{3}{4}n\right]$*?* Since $|R| = n - |L|$, $X_n \le \frac{3}{4}n$ if and only if $n/4 < |L| \le 3n/4$. There are $3n/4 - n/4$ values of p that satisfy this condition. As we pick p uniformly at random, this probability is

$$\frac{3n/4 - n/4}{n} = \frac{n/2}{n} = \frac{1}{2}.$$

Notice that given an input sequence of size n, how the algorithm performs in the future is irrespective of what it did in the past. Its cost from that point on only depends on the random choice it makes after that. So, we'll let $\overline{W}(n) = \mathbf{E}[W(n)]$ denote the expected work performed on input of size n.

Now by the definition of expectation, we have

$$\begin{aligned}
\overline{W}(n) &\le \sum_i \mathbf{Pr}[X_n = i] \cdot \overline{W}(i) + c \cdot n \\
&\le \mathbf{Pr}\left[X_n \le \tfrac{3n}{4}\right] \overline{W}(3n/4) + \mathbf{Pr}\left[X_n > \tfrac{3n}{4}\right] \overline{W}(n) + c \cdot n \\
&= \tfrac{1}{2}\overline{W}(3n/4) + \tfrac{1}{2}\overline{W}(n) + c \cdot n \\
&\implies (1 - \tfrac{1}{2})\overline{W}(n) = \tfrac{1}{2}\overline{W}(3n/4) + c\cdot && \text{[collecting similar terms]} \\
&\implies \overline{W}(n) \le \overline{W}(3n/4) + 2c \cdot n. && \text{[multiply by 2]}
\end{aligned}$$

In this derivation, we made use of the fact that with probability $1/2$, the instance size shrinks to at most $3n/4$—and with probability $1/2$, the instance size is still larger than $3n/4$, so we pessimistically upper bound it with n. Note that the real size might be smaller, but we err on the safe side since we don't have a handle on that.

Finally, the recurrence $\overline{W}(n) \le \overline{W}(3n/4) + 2cn$ is root dominated and therefore solves to $O(n)$.

3.1 Span

Let's now turn to the span analysis. We'll apply the same strategy as what we did for the work recurrence. We have already established the span recurrence:

$$S(n) = S(X_n) + O(\log n)$$

where X_n is the size of the larger side, which is an upper bound on the size of the side the algorithm actually recurses into. Let $\overline{S}(n)$ denote $\mathbf{E}[S(n)]$. As we observed before, how the algorithm performs in the future is irrespective of what it did in the past. Its cost from that point on only depends on the random choice it makes after that. So then, by the definition of expectation, we have

$$\begin{aligned}
\overline{S}(n) &=\le \sum_i \mathbf{Pr}\left[X_n = i\right] \cdot \overline{W}(i) + c\log n \\
&\le \mathbf{Pr}\left[X_n \le \tfrac{3n}{4}\right] \overline{S}(3n/4) + \mathbf{Pr}\left[X_n > \tfrac{3n}{4}\right] \overline{S}(n) + c \cdot \log n \\
&\le \tfrac{1}{2}\overline{S}(3n/4) + \tfrac{1}{2}\overline{S}(n) + c \cdot \log n \\
&\implies (1 - \tfrac{1}{2})\overline{S}(n) \le \tfrac{1}{2}\overline{S}(3n/4) + c\log n \\
&\implies \overline{S}(n) \le \overline{S}(3n/4) + 2c\log n,
\end{aligned}$$

which we know is balanced and solves to $O(\log^2 n)$.

4 Teaser — How to Analyze the Work and Span of QuickSort?

The following implementation of quick sort should familiar. The algorithm is randomized because the pivot selection in Line 4 picks a random element of S.

```
 1  fun qsort($S, k$) =
 2    if $|S| \le 1$ then $S$
 3    else let
 4          val $p$ = pick a random element of S
 5          val $S_1 = \langle s \in S \mid s < p \rangle$
 6          val $S_2 = \langle s \in S \mid s = p \rangle$
 7          val $S_3 = \langle s \in S \mid s > p \rangle$
 8       in
 9          qsort($S_1$) @ $S_2$ @ qsort($S_3$)
10       end
```

We will show that this `qsort` algorithm has *expected* $O(n\log n)$ work and $O(\log^2 n)$ span.