Notes on Expected Runtime of Randomized Quickselect and Randomized Quicksort

1 Introduction to Randomized Algorithms

A tale of two gamblers. Eva goes to the Monte Carlo Casino in Monaco and decides to play blackjack for 2 hours using her favorite card-counting strategy. Her goal is to finish with at least $1000 more than she started with. After the 2 hours are over, Eva has committed to leaving the casino and getting a drink with a friend (either using her winnings to celebrate, or at the treat of her friend who will helps her forget her losses). In this situation, the amount of money Eva spends is a random variable, but Eva is guaranteed to stop gambling within 2 hours. The algorithm Eva is employing is called a *Monte Carlo algorithm*. Such an algorithm is guaranteed to run within a certain time bound (in this case, 2 hours), but the outcome of whether or not the algorithm returns with a correct answer is a random variable (let us say Eva gets the wrong answer if she does not make her goal of winning $1000 over the 2 hours of play).

Neville goes to the Mirage casino in Las Vegas and also decides to play blackjack using his favorite card-counting strategy. Like Eva, his goal also is finish with at least $1000 more than he started with. However, unlike Eva, Neville is committed to keep playing for as long as it takes until he winds up $1000 richer.[1] The algorithm Neville is employing is called a *Las Vegas algorithm*. Such an algorithm is guaranteed to return a correct answer (Neville is guaranteed to win $1000), but the amount of time taken by the algorithm to do so is a random variable.

More formally. Las Vegas algorithms are randomized algorithms that are always guaranteed to return a correct answer (or notify when they fail to do so). Rather than gambling with the correctness of the algorithm, they gamble with the amount of resources they may use. We will look at a special case of a Las Vegas algorithm, Randomized Quicksort, that never fails to return a correct answer and that gambles with the amount of time it may take to return. On the other hand, Monte Carlo algorithms are algorithms which are allowed to fail with some (usually small) probability but always run within a known time bound. They gamble with correctness itself, but not with the resources they use.

[1] Of course, this is unrealistic, as Neville may need an infinite amount of money to ensure he doesn't dwindle all of his capital and have to leave the table prematurely. Neville will probably lose all his money!

2 Randomized Quickselect

In the previous lecture, we saw that Quickselect using a median of medians pivot has a worst-case runtime of $O(n)$. However, the median of medians procedure for selecting the pivot was quite complicated and itself involved additional recursion. We will now see that a much simpler algorithm, Randomized Quickselect, has an *expected runtime* of $O(n)$ over any possible input.

Recall that in Quickselect, each node of the recursion tree has three children, but whenever the algorithm has not yet found the correct answer (and so still has work to do), it can prune the middle child (corresponding to the pivot) and either the left or the right child. Thus, ignoring the pruned children, the recursion tree is really a *recursion path*. That is, the algorithm simply traverses a single path of a Quicksort recursion tree.

We therefore can view Quickselect as traversing a sequence of nodes of decreasing size. Let us first consider two extreme cases:

- In an unlucky round, our random choice of pivot is the minimum or maximum element; in this case, the node size decreases by only 1.
- In a lucky round, the pivot is the median, and the node size is roughly halved.

However, a median pivot is not really required to get a good reduction in node size, as we saw in the last lecture with β-approximate medians. If the pivot lies in the middle half of the elements when sorted by their values (see Figure 1), the node size will be reduced from n to at most $\frac{3}{4}n$. What is the probability that the pivot lies in the middle half? Since the pivot is drawn uniformly at random over all the values, this probability is $\frac{1}{2}$. Let's call any pivot in the middle half a *good pivot*.

Before formally upper bounding the expected runtime of Randomized Quickselect, let's first quickly sketch a proof. First, note that in each node of the recursion tree, the probability that the selected pivot is a good pivot is $\frac{1}{2}$. When a good pivot is selected, the next node's size will be at most 3/4 the current node's size. Therefore, it will be useful to upper bound the number of times we select a pivot until we get a good pivot. Let's call this number M. It follows that each M pivots (each M nodes in the recursion tree), we expect the node size to shrink to at most 3/4 of its current size. Since this is a geometric decrease, after roughly $\log n$ good pivots (which in expectation takes $M \cdot \log n$ nodes of the recursion tree), the node size is at most 1 and hence the algorithm will have finished.

A key quantity to bound in the sketch is M, the expected number of times we select a pivot until we get a good pivot. Equivalently, given a coin whose probability of Heads is[2] $1/2 = 1 - p$, we want to find the expected number of times we flip the coin until we receive the first Heads.

The next lemma bounds this quantity.

Lemma 1. *Suppose we have a coin with probability of Heads equal to $1 - p$ and probability of Tails equal to p. Let Z be the number of times we flip the coin until receiving the first Heads. Then $\mathsf{E}[Z] = \frac{1}{1-p}$.*

[2]The reason for using $1 - p$ for probability of Heads instead of p is that the math is easier when proving the next lemma.

Proof. First, by definition of expectation, we have

$$\mathsf{E}[Z] = \sum_{k=1}^{\infty} k \Pr(Z = k).$$

The probability that $Z = k$ is the same as the probability of the event that the first $k-1$ flips come out Tails and the k^{th} flip comes out Heads. From independence, we therefore have $\Pr(Z = k) = p^{k-1}(1-p)$.

Hence,

$$\begin{aligned}
\mathsf{E}[Z] &= \sum_{k=1}^{\infty} k p^{k-1}(1-p) \\
&= (1-p) \sum_{k=1}^{\infty} k p^{k-1} \\
&= (1-p) \sum_{k=1}^{\infty} \frac{d}{dp} p^k && \text{(“derivative trick”)} \\
&= (1-p) \frac{d}{dp} \sum_{k=1}^{\infty} p^k && \text{(linearity of the derivative)} \\
&= (1-p) \frac{d}{dp} \left(\frac{1}{1-p} - 1 \right) && \text{(geometric series)} \\
&= (1-p) \frac{d}{dp} \frac{1}{1-p} \\
&= (1-p) \frac{1}{(1-p)^2} \\
&= \frac{1}{1-p}.
\end{aligned}$$

□

Consider the special case of a good pivot, for which we have $1 - p = p = 1/2$. We have have just computed that the expected “waiting time” before getting a good pivot is at most $\frac{1}{1-1/2} = 2$ two draws of a pivot, which accords with our intuition that, on average, two flips of a fair coin should produce one outcome of Heads. Also, the above calculation is more general, as it is expressed in terms of p (consider what the expected waiting time would be if the probability of Tails (or a bad pivot) is instead $p = \frac{2}{3}$).

We now proceed with a formal analysis. We split the sequence of nodes into epochs, where each epoch consists of a contiguous sequence of nodes, and each node belongs to precisely one epoch. We decide which node belongs to which epoch as follows:

- Epoch zero consists of all nodes whose size is strictly greater than $\frac{3}{4}n$ and at most n.
- Epoch one consists of all nodes whose size is strictly greater than $\left(\frac{3}{4}\right)^2 n$ and at most $\frac{3}{4}n$.

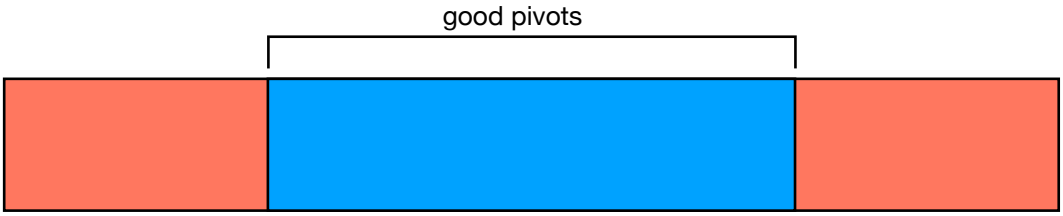

Figure 1: The sequence of elements in a given node. The worst-case choices of good pivots still lead to a reduction in node size by a factor 3/4. These worst-case choices are *(i)* the left-most element in the blue middle region, in which case the next node will contain nearly all of the blue middle and the red right region; *(ii)* the right-most element in the blue region, in which case the next node will contain nearly all of the blue middle and the red left region.

- More generally, for any node i belonging to epoch j, the size S_i of this node satisfies

$$\left(\frac{3}{4}\right)^{j+1} n < S_i \leq \left(\frac{3}{4}\right)^{j} n.$$

Let $\mathcal{E}_j$ be the indices of nodes belonging to epoch j. It is not hard to get a good upper bound on the total number of epochs. However, for our purposes, it will be fine to use the loose upper bound of n.

With the nodes split into epochs, let us now try to upper bound the runtime of Randomized Quickselect. We know that the amount of work the algorithm does in each node, ignoring a possible recursive call within that node, is linear in the number of elements. We can thus express the total runtime of the algorithm as

$$\sum_{i=1}^{n} c \cdot S_i,$$

for some constant $c \geq 1$, where we recall that S_i denotes the number of elements in node i.[3]

Since we have partitioned the nodes into epochs, we can rewrite this expression for the runtime as

$$c \sum_{j=0}^{n} \sum_{i \in \mathcal{E}_j} S_i.$$

Let $L_j = |\mathcal{E}_j|$ be the number of nodes in the j^{th} epoch (here, L stands for "length of the chain of nodes"). Note that L_j is a random variable because the number of pivots selected before moving to the next epoch depends on which random pivots are selected. Also, recall that for each node i in epoch j, the node size S_i is at most $\left(\frac{3}{4}\right)^j n$. Using these two facts, we see that the total amount of work done by Randomized Quickselect is at most

$$c \sum_{j=0}^{n} L_j \cdot \left(\frac{3}{4}\right)^{j} n.$$

[3]Although we sum all the way to n, it is likely that for some intermediate value of i and greater, all nodes are empty because the algorithm already has returned.

We will actually get an upper bound on the *expected* runtime of Randomized Quickselect, so we just take the expectation of the above random runtime, yielding

$$\mathsf{E}\left[c\sum_{j=0}^{n} L_j \cdot \left(\frac{3}{4}\right)^j n\right].$$

By linearity of expectation, this is equal to

$$cn\sum_{j=0}^{n}\left(\frac{3}{4}\right)^j \mathsf{E}\left[L_j\right]. \tag{1}$$

Let us upper bound the expected number of nodes in each epoch. First, from the definition of expectation, we have

$$\mathsf{E}[L_j] = \sum_{k=1}^{\infty} kP(L_j = k), \tag{2}$$

and so we just need to compute the probability (for a given j) that L_j is equal to k, for each k. We know that as soon as a random pivot is chosen to be in the middle half of the elements in the current node, the node size will be reduced by a factor of $\frac{3}{4}$ and the next epoch will start. Thus, the probability that L_j (for any epoch j) is equal to k is at most the probability of drawing $k-1$ bad pivots in a row (a probability $\left(\frac{1}{2}\right)^{k-1}$ event) followed by drawing a good pivot (a probability $\frac{1}{2}$ event). Therefore, the expected value of L_j is at most the expected number of coin flips (for a coin with probability of Heads 1/2) until we receive the first Heads. From Lemma 1, we therefore have that

$$\mathsf{E}[L_j] \leq 2.$$

Plugging in our upper bound $\mathsf{E}[L_j] \leq 2$ into (1), we obtain the following upper bound on the expected runtime of Randomized Quickselect:

$$2cn\sum_{j=0}^{n}\left(\frac{3}{4}\right)^j \leq 2cn\sum_{j=0}^{\infty}\left(\frac{3}{4}\right)^j = 2cn\cdot\frac{1}{1-\frac{3}{4}} = 8cn.$$

3 Randomized Quicksort

To analyze the expected runtime of Quicksort, let us quickly recall what the algorithm actually does. Each call to Quicksort involves a call to Partition followed by (in the worst case) two recursive calls to Quicksort. Thus, excluding the function calls themselves, all the work is done inside Partition. Every time Partition is called, an element is selected as the pivot; this element can never again be selected as a pivot in a later call to Partition. Therefore, there can be at most n calls to Partition[4], each of which has a cost of $O(1)$ if we only count the constant overhead for the call itself and lines of code that are executed only once (per call) within Partition. The cost accounted for thus far is $a \cdot n$ for some constant $a \geq 1$. In addition, each call to Partition will do some work inside the `for` loop (see the lectures slides from Lecture 3 for the code being discussed), and this is the only work we have not yet accounted for in the expected runtime.

The key observation is that for each iteration of the `for` loop in Partition, one element is compared to the current pivot and, in the remainder of the iteration, some constant amount of work is done. Thus, to upper bound how much time is spent cumulatively across *all* the `for` loops (i.e. over all calls to Partition), it is sufficient to simply count how many comparisons between an element and a pivot take place. Let us denote X as the total number of such comparisons performed.

That is, the runtime is bounded as

$$a \cdot n + cX,$$

for some constant $c \geq 1$.

Since we are analyzing Quicksort with a random pivot, the total number of comparisons X is a random variable. To get an expected runtime bound, we only need to bound

$$\mathsf{E}\left[a \cdot n + cX\right],$$

which, from linearity of expectation, is equal to

$$a \cdot n + c\,\mathsf{E}\left[X\right].$$

Now, how can we go about computing the expected value of the number of comparisons? We can use certain properties of the algorithm to make our task easier. First, let us denote the elements by $z_1, z_2, \ldots, z_n$ and assume that $z_1 \leq z_2 \leq \ldots \leq z_n$.[5] We will decompose the total number of comparisons X as a sum of all the individual comparisons:

$$X = \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} X_{jk},$$

where X_{jk} is the indicator random variable for the event $[z_j$ is compared to $z_k]$.

[4] In the worst-case. That is this bound is an absolutely true bound, not just one that holds in expecatation.

[5] Note that we are not assuming that the algorithm already has sorted data. We are just using this way of indexing into the data in our analysis.

Again applying linearity of expectation, we see that the expected runtime is upper bounded by

$$\begin{aligned} & a \cdot n + c \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \mathsf{E}\left[X_{jk}\right] \\ & = a \cdot n + c \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} P([z_j \text{ is compared to } z_k]). \end{aligned}$$

All we need to do is get a good upper bound on the probability that z_j and z_k are compared and we are done. It will be useful to define $Z_{j \to k}$ as the set $\{z_j, z_{j+1}, \ldots, z_k\}$. When can z_j and z_k be compared? Well, we need for $Z_{j \to k}$ to be contained in one node and either z_j or z_k must be selected as the pivot in this node. Let us formalize this node. Assume the nodes in the recursion tree are indexed from $1, 2, \ldots$. Let R_{jk} be the index of the first node containing $Z_{j \to k}$ such that some element of $Z_{j \to k}$ is selected as the pivot in node R_{jk}; if there is no such first node, then set $R_{jk} = 0$.

Note that if z_j is compared to z_k, then $R_{jk} > 0$ and it must be the case that either z_j or z_k is selected as the pivot in node R_{jk}; if some other element of $Z_{j \to k}$ is selected as the pivot, then Partition will place z_j and z_k into different subtrees and they can never be compared.

Therefore,

$$\begin{aligned} & \Pr\left(z_j \text{ is compared to } z_k\right) \\ & = \Pr\left((R_{jk} > 0) \textbf{ and } (z_j \text{ is pivot or } z_k \text{ is pivot in node } R_{jk})\right) \\ & = \Pr(R_{jk} > 0) \cdot \Pr\left(z_j \text{ is pivot or } z_k \text{ is pivot in node } R_{jk} \mid R_{jk} > 0\right) \\ & \leq \Pr\left(z_j \text{ is pivot or } z_k \text{ is pivot in node } R_{jk} \mid R_{jk} > 0\right) \end{aligned}$$

It remains to bound the above probability. Since we condition on the event $R_{jk} > 0$, we may assume that an element of $Z_{j \to k}$ is selected as the pivot in node R_{jk}. Since each element in node R_{jk} is equally likely to be selected as the pivot, it follows that the above probability is equal to

$$\frac{2}{|Z_{j \to k}|} = \frac{2}{k - j + 1}.$$

Table 1: Elements in node R_{jk}. One of z_j through z_k will be selected as the pivot.

$\ldots$	z_j	z_{j+1}	z_{j+2}	$\ldots$	z_{k-2}	z_{k-1}	z_k	$\ldots$

The expected number of comparisons is therefore at most

$$\begin{aligned}\sum_{j=1}^{n-1}\sum_{k=j+1}^{n}\frac{2}{k-j+1} &= \sum_{j=1}^{n-1}\sum_{k=2}^{n-j+1}\frac{2}{k}\\ &\leq (n-1)\sum_{k=2}^{n}\frac{2}{k}\\ &\leq 2(n-1)\int_{1}^{n}\frac{1}{x}dx\\ &= 2(n-1)\log n,\end{aligned}$$

where the last inequality visually can be proved by expressing the summation on the second line as the sum of areas of rectangles and the integral on the third line as the area under the curve $f(x) = \frac{1}{x}$ between the limits of integration.

Therefore, the expected runtime of Randomized QuickSort is at most

$$a \cdot n + 2c(n-1)\log n = O(n \log n).$$