Lecture Notes on QuickSort Analysis[1]

1 The Problem

We are given an unsorted array A containing n numbers, and need to produce a sorted version (in increasing order) of these same numbers.

2 The Partition Subroutine

A key subroutine in QuickSort is to partition an array around a *pivot element*. This subroutine picks an element p of the array A (more on how to do this later), and rearranges the elements of A so that all array elements earlier than p in A are less than p and all array elements subsequent to p in A are greater than p. So if the initial array is $[3, 2, 5, 7, 6, 1, 8]$ and 3 is chosen as the pivot element, then the subroutine could output the rearranged array $[2, 1, 3, 5, 7, 8, 6]$. Note that we don't care about the relative order of elements that are on the same side of the pivot element.

Partitioning around a pivot is useful for two reasons: it reduces the problem size, and it can be implemented in linear time. First we show that, given the pivot element, this partitioning can be done in linear time. One easy way to accomplish this is to do a linear scan through A and copy its elements into a second array B from both ends. In more detail, initialize pointers j and k to 1 and n, respectively. For $i = 1$ up to n, copy the element $A[i]$ over to $B[j]$ (and advance j) if $A[i]$ is less than the pivot, and copy it to $B[k]$ (and decrement k) if it is greater than the pivot. (The pivot element is copied over to B last, in the final remaining slot.)

More interestingly and practically, there are various slick ways to partition around a pivot in place (i.e., without introducing the second array B), via repeated swaps. There is a video covering this in detail. It is such in-place versions that make partitioning such an attractive subroutine in practice.

3 The QuickSort Algorithm

Here is an underspecified description of the QuickSort algorithm.

QuickSort(A,n)

(0) If $n = 1$ return.

(1) p = ChoosePivot(A, n).

(2) Partition A around p.

(3) Recurse on the first part of A (the subarray of elements less than the pivot).

(4) Recurse on the second part of A (the subarray of elements greater than the pivot).

Correctness follows easily by induction on n (see the optional video). The running time of QuickSort depends on the quality of the pivot. In the worst case (e.g., if the minimum element is chosen each time) the running time is $\Theta(n^2)$. [Convince yourself of this.] In the best case (e.g., if we use a linear-time median algorithm to choose the pivot each time), the running time is $O(n \log n)$.

The goal of these notes is to analyze the average running time of the QuickSort algorithm when ChoosePivot chooses the pivot element *uniformly at random*, with each element equally like to be chosen as the pivot. Hopefully, a random pivot is "good enough" "often enough" so that the average running time will be much closer to the best case than the worst case (though this is far from obvious!).

4 Running Time Analysis of Randomized QuickSort

Fix an array A of length n. For notational purposes only, we use B to denote the sorted version of the array A. (Thus $B[i]$ denotes the ith smallest element in the array A.)[2] Let Ω denote the underlying sample space (all possible sequences of pivots that QuickSort might choose) and $\sigma \in \Omega$ a particular sequence of pivots. Note that if we fix both the input array A and the sequence σ of pivots, then QuickSort is simply a deterministic algorithm with some fixed running time (e.g., 103,477 operations). We use $RT(\sigma)$ to denote the running time of QuickSort on the input A with the pivot sequence σ. Note that RT is a random variable defined on the sample space Ω. Our goal is prove that the average running time of QuickSort is $O(n \log n)$, which we formalize as follows.

Theorem 1 *There is a constant $c > 0$ such that, for every input array A of length $n \geq 2$,*

$$\mathbf{E}[RT] \leq cn \log n.$$

Here $\mathbf{E}$ denotes the expectation of the random variable RT.

The key to analyzing QuickSort is to focus on the number of comparisons that it makes. Define the random variable X_{ij} to be equal to the number of times that QuickSort compares the elements $B[i]$ and $B[j]$. Note that X_{ij} is always either 0 or 1; $B[i]$ and $B[j]$ are only compared when one of them is the current pivot element, and the pivot is excluded from all future recursive calls. Let C denote the total number of comparisons made by QuickSort (a random variable). Note that

$$C = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}. \tag{1}$$

We begin with a lemma stating that the running time of QuickSort is dominated by the number of comparisons that it makes.

Lemma 2 *There is a constant $a > 0$ such that, for every input array A of length at least 2 and pivot sequence σ, $RT(\sigma) \leq a \cdot C(\sigma)$.*

Proof. (Sketch.) First, in every call to Partition, the pivot is compared once to every other element in the given array. Thus the number of comparisons in the call is linear in the array length, and the total number of operations in the call is at most a constant times this. Outside of the Partition call, QuickSort only performs a constant amount of work in each recursive call. Summing over all recursive calls yields the lemma. ■

Lemma 2 reduces the proof of Theorem 1 to showing the following inequality:

$$\mathbf{E}[C] \leq 2n \ln n. \tag{2}$$

To prove this inequality, we first apply linearity of expectations to (1) to obtain

$$\mathbf{E}[C] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbf{E}[X_{ij}] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1], \tag{3}$$

where the second equation follows from applying the definition of expectation to the 0-1 random variable X_{ij}. We now carefully analyze the event $\{X_{ij} = 1\}$—i.e., the event that the elements $B[i]$ and $B[j]$ are compared at some point in QuickSort's execution.

First think about the outermost call to QuickSort, and suppose that $B[k]$ is chosen as the pivot. There are four cases:

(1) $k = i$ or $k = j$: in this case, $B[i]$ and $B[j]$ are compared (and one of them is excluded from all future recursive calls).

[2] The video lectures use "z_i" instead of "$B[i]$".

(2) $k > i$ and $k < j$: in this crucial case, $B[i], B[j]$ are both compared to the pivot $B[k]$, but they are not compared to each other, and since they are placed in distinct recursive calls ($B[i]$ in the first and $B[j]$ in the second), *they will never be compared in the future.*

(3) $k > i, j$: in this case $B[i], B[j]$ are not compared to each other in this recursive call, but both are passed on to the first recursive call (so they might be compared in the future).

(4) $k < i, j$: similarly, in this case $B[i], B[j]$ are not compared to each other but are both passed on to the second recursive call.

More generally, the key point is this. Among the elements $B[i], B[i+1], \ldots, B[j-1], B[j]$, suppose $B[k]$ is the first to be chosen as a pivot in some recursive call. (Note that all of these elements will participate in the same sequence of recursive calls until one of them is chosen as a pivot.) Then $B[i]$ and $B[j]$ get compared *if and only if* $k = i$ or $k = j$. Since pivots are always chosen uniformly at random, each of the $j - i + 1$ elements $B[i], \ldots, B[j]$ is equally likely to be the first chosen as a pivot. Summarizing, we have

$$\Pr[X_{ij} = 1] = \frac{2}{j - i + 1}. \tag{4}$$

Combining (3) and (4) yields

$$\mathbf{E}[C] = 2\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{j-i+1}.$$

Recall that we are shooting for an $O(n \log n)$ upper bound. Since the above sum has $\Theta(n^2)$ summands, some of which are as large as $1/2$, we need to evaluate it carefully. Note that for each fixed i, the inner sum is

$$\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n-i+1} \leq \sum_{k=2}^{n}\frac{1}{k}.$$

We can upper bound the right-hand side by the area under the curve $f(x) = 1/x$ [draw a picture!]. In other words,

$$\sum_{k=2}^{n}\frac{1}{k} \leq \int_1^n \frac{dx}{x} = \ln x\Big|_1^n = \ln n.$$

Putting it all together, we have

$$\mathbf{E}[C] = 2\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{j-i+1} \leq 2n\sum_{k=2}^{n}\frac{1}{k} \leq 2n \ln n,$$

which completes the proof.