QuickSort & Probabilistic Analysis

Suhas Arehalli

COMP221 - Spring 2024

1 QuickSort

Much like MergeSort, QuickSort is a divide-and-conquer sorting algorithm. To reiterate, a divide-and-conquer algorithm follows the following template:

1. **Divide** the problem into multiple smaller (closer to trivial) sub-problems,
2. **Conquer** each sub-problem, typically through recursion or with a trivial base-case answer,
3. *Recombine* solutions to the subproblem into a solution for the full problem.

Unlike MergeSort, QuickSort will spend most of it's effort dividing the problem into subproblems rather than recombining. We'll see that clever division will actually make recombination effectively free!

1.1 Partitions

Let's first consider what a good division would look like. The goal is to minimize the work we'll later have to do in recombination, and ideally, we would just be able to concatenate the two sorted subarrays together. When can we do this? When all the elements of one subarray are grader than all of the elements in the other! this leads us to the notion of a *partition*:

Def 1 (Partition)**.**
An $A[p]$ *partitions* $A[1 \dots n]$ if $\forall e \in A[1 \dots p-1], e \leq p$ and $\forall e \in A[p+1 \dots n], p \leq e$.

More colloquially, this means that all elements to the left of index p are less than $A[p]$ and all elements to the right of p are greater than $A[p]$

The key observation is that if we have a p s.t. $A[p]$ partitions A, we can simply sort $A[1 \dots p-1]$ and $A[p+1 \dots n]$ and sandwich $A[p]$ in the middle — the easiest possible recombination! Better yet, if we can sort the two subarrays *in-place* (i.e., within the same initial subarray), we have to do no additional work at all! This idea gives us QuickSort, as in Alg. 1.

Its helpful to observe that QuickSort is not really a single sorting algorithm, but a sorting strategy that depends on a particular implementation of the Partition method. This is not too different from our other sort's helper functions — we can think of HeapSort and TreeSort as algorithms in the SelectionSort or InsertionSort families — but this is particularly helpful when navigating QuickSort because we typically don't change the name based on the different partition algorithms. For our purposes, we'll be studying the simplest to analyze, the *Lomuto* partioning scheme, outlined in 2.

Algorithm 1 An in-place QuickSort implementation that uses a recursive helper function

```
function QuickSort(Array A)
    QuickSort(A, 1, N)
end function
function QuickSort(Array A, Integer low, Integer high)
    if low ≥ high then
        return
    end if
    p ←Partition(A, low, high)
    QuickSort(A, low, p − 1)
    QuickSort(A, p + 1, high)
end function
```

Algorithm 2 The Lomuto partitioning scheme

```
function Partition(Array A[1 . . . n], Integer low, Integer high)
    p ← high
    split ← low
    for i ← low . . . high − 1 do
        if A[i] ≤ A[p] then
            Swap((A[i], A[split]))
            split ← split + 1
        end if
    end for
    Swap((A[p], A[split]))
    return split
end function
```

As always, its helpful to build an intuition for how this partition algorithm works: We attempt to partition the subarray $A[low \dots high]$, and choose $A[high]$ to be the value to partition. We then sweep through $a[low \dots high - 1]$ keeping all elements less than $A[high]$ to the left of index *split* and all of the large elements between *split* and i using Swap. Then we put $A[high]$ into it's proper position in the middle. This scheme is visualized in Fig. **??**

1.2 Proof of Correctness

First, we need to construct the right definition of correctness for Partition!

Problem Statement (Partition)**.**
Input: Array $A[1 \dots n]$, indices $1 \leq low, high \leq n$
Output: p such that $A[p]$ *partitions* $A[low \dots high]$.

First, let's assume Partition from above is correct and prove QuickSort correct first!

Statement 1.
After QuickSort(A, low, $high$), $A[low \dots high]$ is sorted.

Proof. First let n be the size of the subarray we're sorting, and observe that $n = high - low + 1$. Then proceed by induction over n.

Base Case: $n = 0, 1$. Arrays of length 0 and 1 are, by definition, sorted. The note that if $n = 0$ or 1, we have that $low \geq high$. In this case, we immediately return, as expected.

Inductive Step: We will use strong induction, so we assume by our inductive hypothesis that $\forall 1 \leq n \leq k$, QuickSort will be correct. We now need to show that for a subarray of size $k + 1$, QuickSort will sort the subarray.

First, we call Partition on the full array, which, because we're assuming it's correct, will return p such that $A[p]$ partitions $A[low \dots high]$. This gives us that $\forall e \in A[low \dots p - 1]$, $e \leq A[p]$ and $\forall e \in A[p + 1 \dots high]$, $A[p] \leq e$.

We then make two recursive calls to QuickSort. Observe that neither of these recursive calls has index p within the bounds of the subarray it's given as an argument. This means that the length of the subarray it recurses on is, at most, of size k, which means that by our inductive hypothesis, the recursive calls must be correct! This means that $A[low \dots p - 1]$ and $A[p + 1 \dots high]$ are sorted after the call, and thus

$$A[low] \leq \cdots \leq A[p - 1] \leq A[p] \leq A[p + 1] \leq \cdots \leq A[high]$$

The left chain of inequalities comes from the fact that $A[low \dots p - 1]$ is sorted, the last chain of inequalities comes from the fact that $A[p + 1 \dots high]$ is sorted. Then observe that $A[p - 1]$ was, before the sort of the left half of the array, an element in $A[1 \dots p - 1]$(the left hand size of the partition), and thus $A[p - 1] \leq A[p]$ from the fact that partition is correct. A parallel argument gives us the last remaining inequality: Since $A[p + 1]$ was, before the recursive sort, an element in $A[p+1 \dots N]$ (the right hand side of our partition), we have by the correctness of our partitioning that $A[p] \leq A[p+1]$. This completes the above inequality, and then observing that this is the same as saying $A[low \dots high]$ completes the proof. □

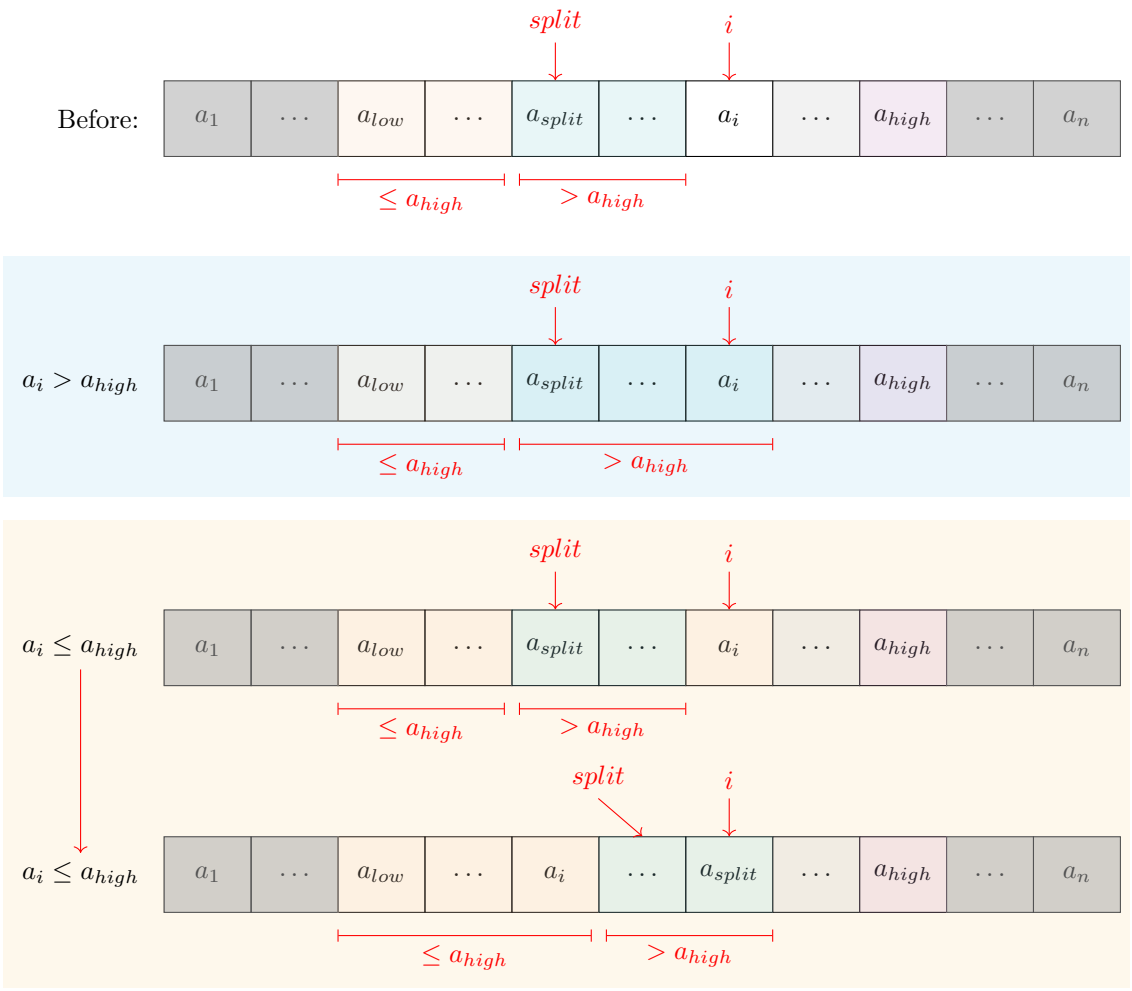

Figure 1: A sketch of the Lomuto Partition algorithm. We begin an iteration with the *Before* configuration: We do not care about the shaded sections outside of $A[low \dots high]$, and have chosen a_{high} in purple to be our pivot. With $A[low \dots i-1]$, we have elements $\leq a_{high}$ before index *split* and those $> a_{high}$ at index *split* or further right. Our job is to properly sort a_i at index i into this arrangement. **Case 1:** If $A_i > a_{high}$, we are in the solution shaded in blue: nothing needs to change! **Case 2:** If $a_i \leq a_{high}$, we need to move a_i to the left of index *split*. We do this by swapping a_split and a_i and incrementing *split*, as shown in the orange shaded region.

Now let's prove PARTITION correct! Since the algorithm is iterative, we must first construct a loop invariant. I recommend this as an exercise (or would if it wasn't in the reading...) — try and make a loop invariant based on the description of the algorithm's motivation above and prove it correct.

Statement 2 (PARTITION LI)**.**
After the kth iteration of the for-loop, $\forall e \in A[low \dots split - 1]$, $e \leq A[p]$ and $\forall e \in A[split \dots low + k - 1]$, $e > A[p]$

Proof. By induction over the number of iterations of the for-loop k.

Base case: Before the loop ($k = 0$), split is initialized to low, so $A[low \dots split - 1]$ and $A[split \dots low - 1]$ are both empty, and the loop invariant is trivially true: There are no elements to compare!

Inductive step Assume that after the kth iteration, we have that $\forall e \in A[low \dots split - 1]$, $e \leq A[p]$ and $\forall e \in A[split \dots low + k - 1]$, $e > A[p]$. We must show that after the $k + 1$st iteration (where $i = low + k$), $\forall e \in A[low \dots split - 1]$, $e \leq A[p]$ and $\forall e \in A[split \dots low + k]$, $e > A[p]$

Let's break this into two cases, based on whether we enter the if statement! **Case 1:** If we don't enter the if-statement, we know $A[i] = A[low+k] > A[p]$ and do nothing. Because nothing changed, by our IH we know that $\forall e \in A[low \dots split - 1]$, $e \leq A[p]$ and $\forall e \in A[split \dots low + k - 1]$, $e > A[p]$. The only thing missing from our desired statement is to show that $A[low+k] > A[p]$, which we get from the conditional.

Case 2: If we do enter the if-statement, we know that $A[low + k] \leq A[p]$. We then swap $A[k + 1]$ and $A[split]$. This means that after the swap, $A[split] \leq A[p]$. Since before the swap, $A[split] \in A[split \dots high]$, by our inductive hypothesis we know that after the swap $A[low + k] > A[p]$.

We then increment $split$. For clarity, let $split_{post}$ be this new value of $split$ and $split_{pre}$ the previous value of $split$. $A[low \dots split_{pre} - 1]$ is untouched in the loop, so for $e \in A[low \dots split_{pre} - 1]$, $e \leq A[p]$. similarly, $A[split_{pre} + 1 \dots k]$ is untouched, so for $e \in A[split_{pre} + 1 \dots k]$, $e > A[p]$.

Converting these to $split_{post}$ terms, we have that $\forall e \in A[low \dots split_{post} - 2]$, $e \leq A[p]$ and $\forall e \in A[split_{post} \dots k]$, $e > A[p]$. If we compare this to what we need to show, all that's left is that $A[split_{post} - 1] \leq A[p]$ and that $A[k+1] > A[p]$. Of course, those are the exact two things we concluded after the swap. □

Then, to close things out,

Statement 3.
PARTITION is correct.

Proof. Using the loop invariant, we can conclude that after the final, $high - low$ iteration, we have that $\forall e \in A[low \dots split - 1]$, $e \leq A[p]$ and $\forall e \in A[split \dots low + (high - low) - 1] = A[split \dots high - 1]$, $e > A[p]$. We then swap $A[p]$ and $A[split]$. Then observe that p

is defined just for clarity: By convention, $p = high$! So this is just swapping $A[high]$ and $A[split]$. $A[split] > A[p]$ before the swap, so after $A[high] > A[split]$. Because of the swap, now all relations w.r.t. $A[p]$ are now w.r.t. $A[split]$! This means that $\forall e \in A[low \dots split - 1]$, $e \leq A[split]$ (none of the elements in in this subarray changed!) and $\forall e \in A[split + 1 \dots high]$, $e > A[split]$ (here split became the *pivot* element and $A[high]$ became what used to be in $A[split]$). Now we just note that this is the definition of $A[split]$ partitioning $A[low \dots high]$ and we return *split*, as desired. □

1.3 Runtime Analysis

1.3.1 Worst-Case

The worst-case time complexity is unfortunately fairly easy to see, even without fancy mathematical tools.

First, convince yourself that PARTITION is $\Theta(n)$

Suppose we're really unlucky with our pivots, and we always pick the largest element in the subarray. After each partitioning, we will split a subarray of size k into two subproblems of sizes... $k - 1$ and 0. It will take $\Theta(n)$ recursive calls to reduce a problem of size n to our base cases of size 0 or 1, and thus our algorithm will be $\Theta(n^2)$. Not good! Worse than MergeSort and HeapSort, and just as bad as Insertion-, Selection-, and BubbleSort!

1.3.2 Average-Case

This is a rare case where we have a substantially better average-case analysis, but this means that we'll need to pull out some discrete probability knowledge. Hopefully it's light enough that you can informally follow this argument if you didn't see probability in Discrete. This argument is borrowed from Skiena's analysis in *The Algorithm Design Handbook*, the old textbook for the course.

Lets define a *good-enough* pivot as a pivot in the middle 50% of values. Formally, lets define the *rank* of an element in an Array as it's index in a sorted version of that array. Observe that this is also the index the element is placed at if it's chosen as a pivot! A good enough pivot p, as defined by Skiena, has rank r such that $\frac{N}{4} \leq r \leq \frac{3N}{4}$.

In an average-case analysis, we need to start talking about probabilities. What is an average case for sorting? Let's assume all orderings of the N elements in our array have equal probability (oof — Erickson said this was unrealistic!). This means that there is a 50% chance that we get a good-enough pivot for every call to PARTITION.

What happens when we get a good-enough pivot? At worst, we divide a problem of size k into a problem of roughly size $\frac{3k}{4}$ and one of roughly size $\frac{k}{4}$. If we always follow the larger of the two subproblems down the recursion chain, we can find the deepest recursion chain. How many times do we recurse? Well if we multiply the size by roughly $\frac{3}{4}$ after each recursive call, we can solve for the max recursive depth $\lceil d \rceil$ by finding d that satisfies

$$(\frac{3}{4})^d n = 1$$

$$d = \log_{4/3} n$$

But wait, this is only true if we get a good-enough pivot at every call to PARTITION! This only happens 50% of the time! It's a coin flip!

Well now to get to the average part of average-case analysis. What is the *expected* (i.e., average) number of flips it takes to get d good-enough pivots?

Formally, we construct a random variables X_i such that $P(X_i = 1) = 0.5$ and $P(X_i = 0) = 0.5$, and define our count of good-enough pivots on a recursion chain k deep $D = \sum_{i=1}^{k} X_i$, and then ask what k gives us $E[D] = d$. Solving this out, we get

$$\begin{aligned}
E[D] &= d \\
E[\sum_{i=1}^{k} X_i] &= \log_{4/3} n \\
\sum_{i=1}^{k} E[X_i] &= \log_{4/3} n \\
kE[X_i] &= \log_{4/3} n \\
k(0.5(1) + 0.5(0)) &= \log_{4/3} n \\
0.5k &= \log_{4/3} n \\
k &= 2\log_{4/3} n
\end{aligned}$$

Of course, we could also just reason that if we have a fair coin, we should expect to half of our flips to come up heads (a consequence of what probability folks call *linearity of expectation*), and thus after $2d$ recursions we should expect d good enough pivots.

Now we just observe that we do $\Theta(n)$ work at each recursive depth, and thus doing $\Theta(n)$ work $2\log_{4/3} n$ times is $O(n \log n)$!

Note that here we assume that if we *don't* get a good-enough pivot, we do *no work at all*! This is a very conservative estimate that gives us an upper bound! What we've actually shown is that the average case of QUICKSORT is $O(n \log n)$. Of course, we've also made it a conservative estimate when we said our good-enough pivot splits the array 75/25 instead of 50/50! This entire argument is only about upper bounds, but that's fine because we care about Big-O!

We *could* do a more sophisticated lower bound for the average case, but luckily, the Ω comes for free! It turns out, we can prove all comparison-based sorts (like all of them we've looked at so far!) are $\Omega(n \log n)$. Stay tuned on this... it'll come up later!

1.3.3 Average-Case *Inputs*

- Let's dig a little deeper into what we mean when we say *average-case analysis*.
- We adopt *-case analysis because for an input of size n, there are often multiple different inputs (i.e., for sorting, there are many different arrays of size n!). The best/worst-case analysis tells us for each n, we write the time complexity growth function for the best/worst *inputs* to our function (of size n).
- It's important to keep in mind that *-case analysis is about the kinds of inputs of size n, distinct from our conversations about big-O, Ω, Θ. One way to make this clear is to note

that if we're doing this formally, picking a case to analyze lets us write growth functions, while choosing a big-O, Ω, Θ notation gives us a nicer way to write a growth function that we already have by letting us talk about that growth function in terms of asymptotic upper and lower bounds that are neater to write down (i.e., nice functions!)

- Average-case analysis is a little bit more complex that best/worst-case, because we consider the *average input*.
  - In practice, this analysis is probabilistic. We assume that we know a probability distribution over possible inputs and we define our growth function as the *expected* number of time steps as a function of n.
  - Since this course requires no background in probability (and it's tricky enough as it is!), we won't be diving deep into this kind of analysis, but know it's there!
- Note that when we talk about the average case analysis above, we are talking about the average *input* under some probability distribution. The execution of the algorithm isn't itself probabilistic (yet...), and so we can still have inputs that are particularly bad for QuickSort (i.e., inputs that are always in the worst-case).
  - For example, consider giving a sorted array as input to the QuickSort we showed above. What always happens with partition?
  - This is often inconvenient for someone just trying to pick the right sorting algorithm to use. Why should I have to think hard about the kinds of arrays I'll be feeding into my sorting algorithm? As Erickson notes, the arrays we sort aren't always uniformly distributed across every possible random ordering as we assume above!
- One clever trick to solve this is to fully commit to the randomization.
  - Before we run QuickSort, spend $O(n)$ time to shuffle your input array.
  - Better yet, note that partition's success depends only on its *rank* within the subarray. Simply randomly select p rather than choosing the rightmost element!
  - Now the probability of our pivots rank being any value 1 to N is now actually equal no matter what the input is!
    - We've moved from having average guarantees across all inputs to having an probabilistic guarantee for *every* input.
    - i.e., no matter what the input order is, if we're not unlucky, we should expect $\Theta(n \log n)$ asymptotic time complexity.