Analysis of QuickSort Sorting Algorithm

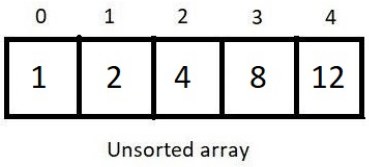
In the last video, we learned to use the quick sort algorithm. We saw how we used the methods of divide and conquer and partitioning to achieve sorting. Later, in the video, we even saw the program to implement all these methods to sort an array using the quick sort algorithm. Today, we'll see the analysis of the quicksort algorithm on all criteria we defined earlier.

1. **Time Complexity:**

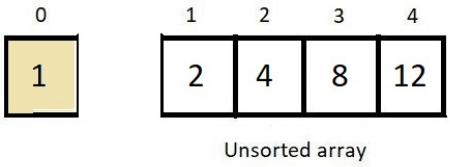
Let's start with the runtime complexity of the algorithm:

Worst Case:

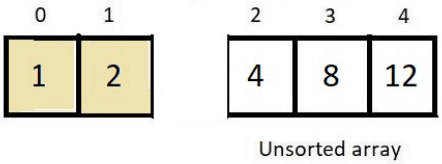
The worst-case in a quicksort algorithm happens when our array is already sorted. I'll take a sorted array of length 5 to demonstrate how it reaches the worst case. Take the one below as an example.



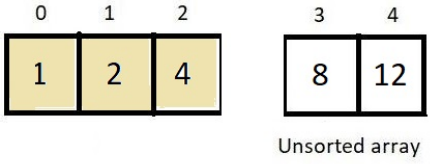
In the first step, you choose 1 as the pivot and apply a partition on the whole array. Since 1 is already at its correct position, we apply quicksort on the rest of the subarrays.



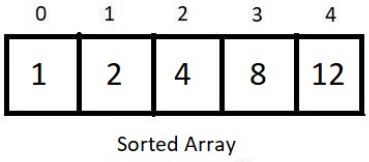
Next, the pivot is element 2, and when applied partition, we found that there is no element less than 2 in the subarray; hence, the pivot remains there itself. We further apply quicksort on the only subarray that is to the right.



Now, the pivot is 4, and since that is already at its correct position, applying partition did make no change. We move ahead.



Now, the pivot is 8, and even that is at its correct position; hence things remain unchanged, and there is just one subarray with a single element left. But since any array with a single element is always sorted, we mark the element 12 sorted as well. And hence our final sorted array becomes,



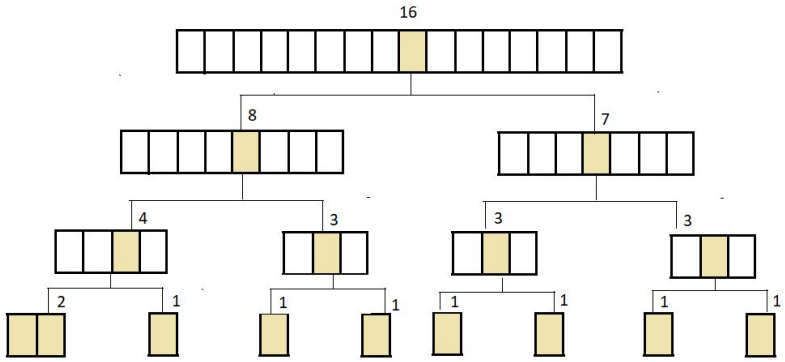
So, if you calculate carefully, for an array of size 5, we had to partition the subarray 4 times. That is, for an array of size n, there would be (n-1) partitions. Now, during each partition, long story short, we made our two index variables, *i and j* run from either direction towards each other until they actually become equal or cross each other. And we do some swapping in between as well. These operations count to some linear function of n, contributing O(n) to the runtime complexity.

And since there are a total of (n-1) partitions, our total runtime complexity becomes n(n-1) which is simply **$O(n^2)$**. This is our worst-case complexity.

Best Case:

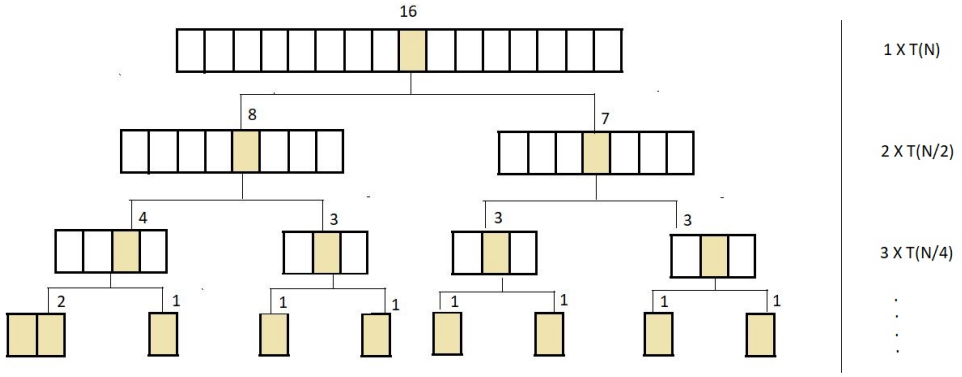
The condition when our algorithm performs in its best possible time complexity is when our array gets divided into two almost equal subarrays at each partition. Below mentioned tree defines the state of best-case when we apply quicksort on an

array of 16 elements, of which each newly made subarray is almost half of its parent array.



No. partitions were different at each level of the tree. If we count starting from the top, the top-level had one partition on an array of length (n=16), the second level had 2 partitions on arrays of length n/2, then the third level had 4 partitions on arrays of length n/4... and so on.

For the above array of length 16, the calculation goes like the one below.



Here, T(x) is the time taken during the partition of the array with x elements. And as we know, the partition takes a linear function time, and we can assume T(x) to be equal to x; hence the total time complexity becomes,

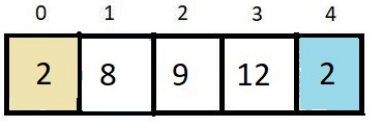
Total time = 1(n) + 2(n/2) +4(n/4) + ...+ until the height of the tree(h) Total time = n*h

Now, can you decide what h is? H is the height of the tree, and the height of the tree, if you remember, is $\log_2(n)$, where n is the size of the given array. In the above example, h = 4, since $\log_2(16)$ equals 4. Hence the time complexity of the algorithm in its best case is **O(nlogn)**.

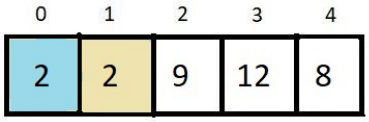
Note: The average time complexity remains O(nlogn). Calculations have been avoided here due to their complexity.

2. **Stability:**

The QuickSort algorithm is not stable. It does swaps of all kinds and hence loses its stability. An example is illustrated below.



When we apply the partition on the above array with the first element as the pivot, our array becomes



And the two 2s get their order reversed. Hence quick sort is not stable.

3. Quicksort algorithm is an in-place algorithm. It doesn't consume any extra space in the memory. It does all kinds of operations in the same array itself.
4. There is no hard and fast rule to choose only the first element as the pivot; rather, you can have any random element as its pivot using the rand() function and that you wouldn't believe actually reduces the algorithm's complexity.

So, that was all we had to discuss regarding the QuickSort algorithm. Seeing each criterion and how our algorithm performs on it gave us a good idea of how good or bad our algorithm is. If it is still taking you a deal to understand things, give it some time and practice more. You will naturally become accustomed to it. You might even want to skip it for now. And it is nowhere a wrong decision. Be consistent, and that would do all.

I appreciate your support throughout. I hope you enjoyed the tutorial. If you genuinely appreciate my work, please let your friends know about this course too. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial, where we'll discuss another sorting algorithm called the **Merge Sort Algorithm.** Till then, keep coding.