Overview of Quicksort

Like merge sort, quicksort uses divide-and-conquer, and so it's a recursive algorithm. The way that quicksort uses divide-and-conquer is a little different from how merge sort does. In merge sort, the divide step does hardly anything, and all the real work happens in the combine step. Quicksort is the opposite: all the real work happens in the divide step. In fact, the combine step in quicksort does absolutely nothing.

Quicksort has a couple of other differences from merge sort. Quicksort works in place. And its worst-case running time is as bad as selection sort's and insertion sort's: $\boldsymbol{\Theta(n^2)}$. But its average-case running time is as good as merge sort's: $\boldsymbol{\Theta(nlgn)}$. So why think about quicksort when merge sort is at least as good? That's because the constant factor hidden in the **big-Θ** notation for quicksort is quite good. In practice, quicksort outperforms merge sort, and it significantly outperforms selection sort and insertion sort.

Here is how quicksort uses divide-and-conquer. As with merge sort, think of sorting a subarray **array[p..r]**, where initially the subarray is **array[0..n-1]**.

1. Divide by choosing any element in the subarray **array[p..r]**. Call this element the ***pivot***. Rearrange the elements in **array[p..r]** so that all other elements in **array[p..r]** that are less than or equal to the pivot are to its left and all elements in **array[p..r]** are to the pivot's right. We call this procedure partitioning. At this point, it doesn't matter what order the elements to the left of the pivot are in relative to each other, and the same holds for the elements to the right of the pivot. We just care that each element is somewhere on the correct side of the pivot.
   As a matter of practice, we'll always choose the rightmost element in the subarray, **array[r]**, as the pivot. So, for example, if the subarray consists of [9, 7, 5, 11, 12, 2, 14, 3, 10, 6], then we choose 6 as the pivot. After partitioning, the subarray might look like [5, 2, 3, 6, 12, 7, 14, 9, 10, 11]. Let

partitioning, the subarray might look like [3, 2, 5, 6, 12, 7, 14, 9, 10, 11]. Let q be the index of where the pivot ends up.

2. Conquer by recursively sorting the subarrays **array[p..q-1]** (all elements to the left of the pivot, which must be less than or equal to the pivot) and **array[q+1..r]** (all elements to the right of the pivot, which must be greater than the pivot).

3. Combine by doing nothing. Once the conquer step recursively sorts, we are done. Why? All elements to the left of the pivot, in **array[p..q-1]**, are less than or equal to the pivot and are sorted, and all elements to the right of the pivot, in **array[q+1..r]**, are greater than the pivot and are sorted. The elements in **array[p..r]** can't help but be sorted!
   Think about our example. After recursively sorting the subarrays to the left and right of the pivot, the subarray to the left of the pivot is [2, 3, 5], and the subarray to the right of the pivot is [7, 9, 10, 11, 12, 14]. So the subarray has [2, 3, 5], followed by 6, followed by [7, 9, 10, 11, 12, 14]. The subarray is sorted.

The base cases are subarrays of fewer than two elements, just as in merge sort. In merge sort, you never see a subarray with no elements, but you can in quicksort, if the other elements in the subarray are all less than the pivot or all greater than the pivot.

Let's go back to the conquer step and walk through the recursive sorting of the subarrays. After the first partition, we have have subarrays of [5, 2, 3] and [12, 7, 14, 9, 10, 11], with 6 as the pivot.

To sort the subarray [5, 2, 3], we choose 3 as the pivot. After partitioning, we have [2, 3, 5]. The subarray [2], to the left of the pivot, is a base case when we recurse, as is the subarray [5], to the right of the pivot.

To sort the subarray [12, 7, 14, 9, 10, 11], we choose 11 as the pivot, resulting in [7, 9, 10] to the left of the pivot and [14, 12] to the right. After these subarrays are sorted, we have [7, 9, 10], followed by 11, followed by [12, 14].

Here is how the entire quicksort algorithm unfolds. Array locations in blue have been pivots in previous recursive calls, and so the values in these locations will not be examined or moved again: