


### 3. Quicksort

Quicksort takes a different approach to sort the array in place. Its first job is to identify a pivot element in the array. We'd like this element to be in the middle range of the array, so we often check the first element, the middle element, and the last element and take their median (middle) value. For us, that's 3. We'll remember that value.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 8 | 4 | 5 | 2 | 1 | 7 | 6 | 3 |
|---|---|---|---|---|---|---|---|


The main work of quicksort is a process called partitioning, which puts the pivot in the correct place and puts all the items  $\leq$  pivot on the left hand side of the pivot, and all the items  $>$  pivot on the right hand side.

First in partition, we run a loop that starts at the front of the array and continues forwards as long as the items it finds are less than or equal to the pivot element. When we find an item that's bigger than the pivot, we will stop at that index. Here, that's the first item (the 8).



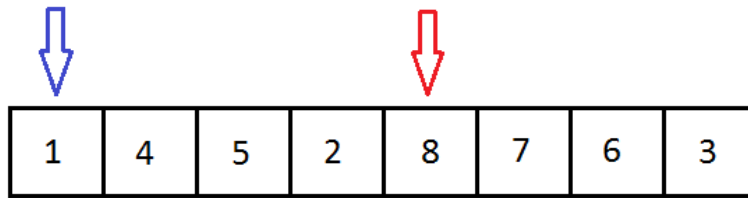
|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 8 | 4 | 5 | 2 | 1 | 7 | 6 | 3 |
|---|---|---|---|---|---|---|---|

We also run a loop from the end of the array backwards that continues as long as the items it finds are greater than the pivot element. This loop stops at the 1, which is less than the pivot.

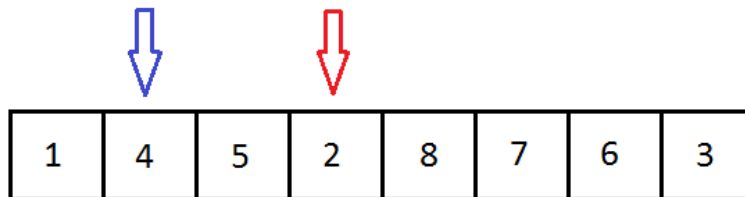


|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 8 | 4 | 5 | 2 | 1 | 7 | 6 | 3 |
|---|---|---|---|---|---|---|---|

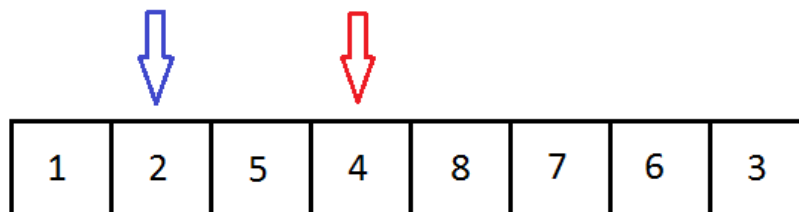
Now, we swap the 8 and the 1:



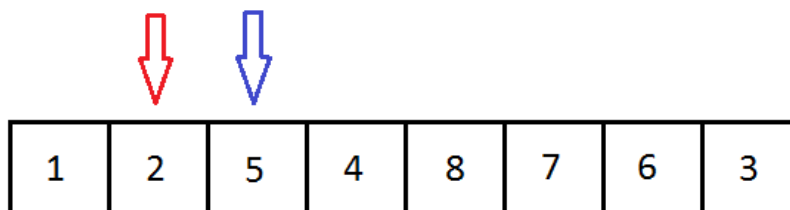
This leaves us free to continue the loops. The forward loop will now stop at the 4, and the backward loop will now stop at the 2:



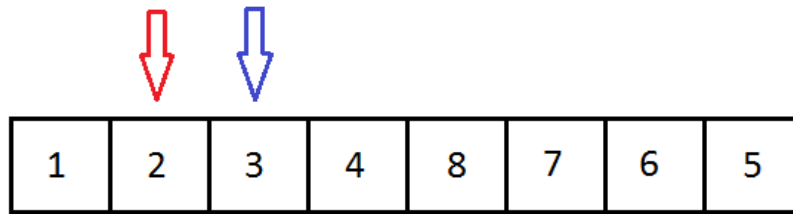
Then we swap these items:



Now, if we loop forward again until we find a too-big item, and backward again until we find a too-small item, our indexes cross:



This crossing tells us that we are nearly done with one partition. All that remains is to swap the pivot element, the 3, with the last item we found that was too big for the left hand side (the 5):



With this done, notice that the pivot element (the 3) is in place, and everything to the left of the pivot is less than or equal to the pivot, and everything to the right of the pivot is greater than the pivot.

What comes next? Recursively sort the subarrays [1 2] and [4 8 7 6 5] by partitioning each of these around a new pivot element. (For the right hand side, 5 will be the next pivot.) When all of these little arrays are sorted, the whole array is sorted as well.

Quicksort is usually  $O(n \log n)$ , but it can do worse. It depends on the pivot choice. What would partition have done if we'd chosen the 8 as the pivot? Or the 1? We make this choice carefully because if we're unlucky in every pivot selection, the algorithm degenerates to  $O(n^2)$  again.

Notice finally that the book recommends that when your quicksort's sub-arrays become small, you sort them with something quadratic and easy, like insertion sort. We can get away with this as long as we only do it on arrays with size less than or equal to some constant.