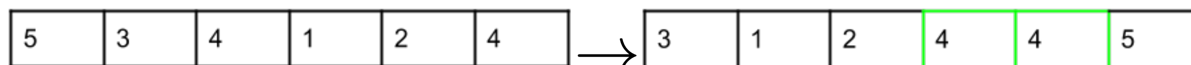
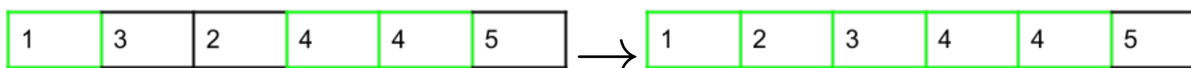
Quicksort

1 Quicksort

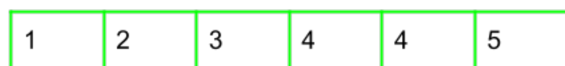
We will now discuss one of the fastest sorts ***Quicksort*** (who would've guessed this one would be fast). The idea of quicksort is that we choose some random pivot and create 3 subarrays. One subarray would contain all the elements less than the pivot (traditionally to the left), one subarray would contain elements equal to the pivot, and one subarray would contain elements greater than the pivot (usually to the right of the pivot). Here is an example of quicksort.



We started off with the unsorted array, we randomly choose a pivot of 4. We then move all the elements, in the order that they appear, to the left or right of the pivot based off their quantity. Note that our sorted array contains 2 elements. This is because there are 2 4's. From now on, the position of the 4's will not change.



We start off by going to the left unsorted array and choose 1 as a pivot. Nothing in its subarray is less than it, so we move both elements to its right. Following this, we choose 2 as a pivot and move 3 to the right of it. We no call quicksort on the last element 3 and recognize that there are no more elements, so quicksort has completed on this end.



Now that we have finished with the left side of the original pivot, we will go to the right side of the pivot. The only element on that side is 5. Thus we go to that element and then we are done with our quicksort.

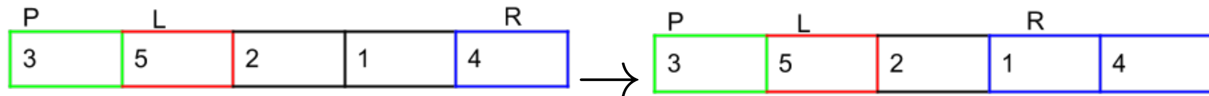
The runtime for quicksort is $\Theta(Nlog(N)$ assuming we have a random pivot that is good (basically that we do not always end up with a pivot that has everything to the left of right of it).

Quicksort will use, on average, $(log(N)$ extra space because there will be a call stack/recursive calls.

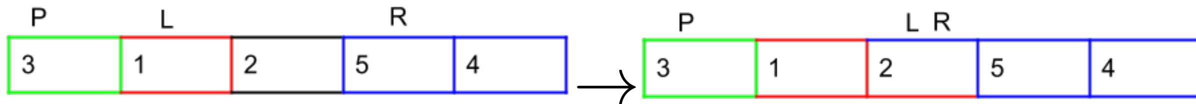
Quicksort is unstable because items will not keep their relative position to elements of the same type. The pivot is randomized, thus it is impossible for there to be stability.

The previous way that we did quicksort was through making new arrays; however, there is a method that is almost always guaranteed to run in $\Theta(Nlog(N)$ time that is done in-place. This method is called **Hoare Partioning**. Hoare Partitioning works in the following way, you have a chosen pivot and 2 pointers, one that starts on the left end and one that starts on the right end.

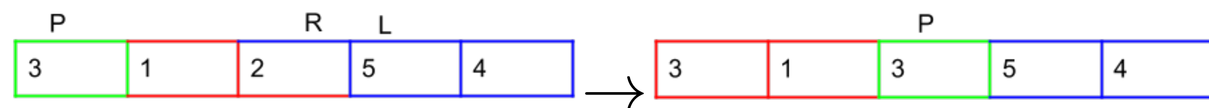
The left pointer moves to the right until it finds an item that is greater than or equal to the pivot. The right pointer moves left until it finds an item that is less than or equal to the pivot. Once both pointers have stopped, the items that they are on swapped and the process repeats. This process terminates once the pointers pass each other. We will illustrate this by going through 1 partition.



In this step, we started off with a pivot of 3 and we made our left pointer start on the element 5 and our right pointer start on the element 4. Immediately, the left pointer halts on the element 5 since 5 is greater than the pivot. The right pointer moves down to the element to the right of it.



Since our right pointer was stopped on 1, because it is less than 3, we swap 1 and 5. After this, we move both our pointers to 2. At this point, the right pointer is halted since 2 is less than 3.



We move our left pointer one more item to the right. Here we realize that our pointers have crossed paths and our partitioning is done, well almost. As our very last step, we swap the item the right pointer is on with the pivot. Now we have a great in-place partition!