Quicksort will, often, fail to divide the array into equal parts. This is because the whole process depends on how we choose the pivot. We need to choose a pivot so that it's roughly larger than half of the elements, and therefore roughly smaller than the other half of the elements. As intuitive as this process may seem, it's very hard to do.

Think about it for a moment - how would you choose an adequate pivot for your array? A lot of ideas about how to choose a pivot have been presented in Quicksort's history - randomly choosing an element, which doesn't work because of how "expensive" choosing a random element is while not guaranteeing a good pivot choice; picking an element from the middle; picking a median of the first, middle and last element; and even more complicated recursive formulas.

The most straight-forward approach is to simply choose the first (or last) element. This leads to Quicksort, ironically, performing very badly on already sorted (or almost sorted) arrays.

This is how most people choose to implement Quicksort and, since it's simple and this way of choosing the pivot is a very efficient operation (and we'll need to do it repeatedly), this is exactly what we will do.

Now that we have chosen a pivot - what do we do with it? Again, there are several ways of going about the partitioning itself. We will have a "pointer" to our pivot, and a pointer to the "smaller" elements and a pointer to the "larger" elements.

The goal is to move the elements around so that all elements smaller than the pivot are to its left, and all larger elements are to its right. The smaller and larger elements don't necessarily end up sorted, we *just* want them on the proper side of the pivot. We then recursively go through the left and right side of the pivot.

A step by step look at what we're planning to do will help illustrate the process. Using the array shown below, we've chosen the first element as the pivot (29), and the pointer to the smaller elements (called "low") starts right after, and the pointer to the larger elements (called "high") starts at the end.

* *29* is the first pivot, *low* points to *99* and *high* points to *44*

29 | **99 (low)**,27,41,66,28,44,78,87,19,31,76,58,88,83,97,12,21,**44 (high)**

* We move high to the left until we find a value that's lower than our pivot.

29 | **99 (low)**,27,41,66,28,44,78,87,19,31,76,58,88,83,97,12,**21 (high)**,44

* Now that our *high* variable is pointing to *21*, an element smaller than the pivot, we want to find a value near the beginning of the array that we can swap it with. It doesn't make any sense to swap with a value that's also smaller than the pivot, so if *low* is pointing to a smaller element we try and find one that's larger.
* We move our *low* variable to the right until we find an element larger than the *pivot*. Luckily, *low* was already positioned on *99*.
* We swap places of *low* and *high*:

29 | **21 (low)**,27,41,66,28,44,78,87,19,31,76,58,88,83,97,12,**99 (high)**,44

* Right after we do this, we move *high* to the left and *low* to the right (since *21* and *99* are now in their correct places)
* Again, we move *high* to the left until we reach a value lower than the *pivot*, which we find right away - *12*
* Now we search for a value larger than the *pivot* by moving *low* to the right, and we find the first such value at *41*

This process is continued until the *low* and *high* pointers finally meet in a single element:

29 | 21,27,12,19,**28 (low/high)**,44,78,87,66,31,76,58,88,83,97,41,99,44

* We've got no more use of this pivot so the only thing left to do is to swap *pivot* and *high* and we're done with this recursive step:

**28**,21,27,12,19,**29**,44,78,87,66,31,76,58,88,83,97,41,99,44

As you can see, we have achieved that all values smaller than *29* are now to the left of *29*, and all values larger than *29* are to the right.

The algorithm then does the same thing for the *28,21,27,12,19* (left side) collection and the *44,78,87,66,31,76,58,88,83,97,41,99,44* (right side) collection.

**Implementation**

**Sorting Arrays**

Quicksort is a naturally recursive algorithm - divide the input array into smaller arrays, move the elements to the proper side of the pivot, and repeat.

Let's go through how a few recursive calls would look:

* When we first call the algorithm, we consider all of the elements - from indexes *0* to *n-1* where *n* is the number of elements in our array.
* If our pivot ended up in position *k*, we'd then repeat the process for elements from *0* to *k-1* and from *k+1* to *n-1*.
* While sorting the elements from *k+1* to *n-1*, the current pivot would end up in some position *p*. We'd then sort the elements from *k+1* to *p-1* and *p+1* to *n-1*, and so on.

Bottom of Form

That being said, we'll utilize two functions - partition() and quick\_sort(). The quick\_sort() function will first partition() the collection and then recursively call itself on the divided parts.