**Quick Sort**

* Quick Sort is a **divide-and-conquer** algorithm.
* It picks an element as **pivot** and partitions the given array around that pivot.

**Selecting A Pivot**

* Note that there are many different versions of quick sort that pick pivots in different ways.
  + Always pick first element as pivot.
  + Always pick last element as pivot (implemented below)
  + Pick a random element as pivot.
  + Pick median as pivot.

**Median-of-Three Pivot Selection**

* Ideally, the pivot should be the **median** value in the array, so that the subarrays *S*1 and *S*2 each have the same—or nearly the same—number of entries.
* One way to find the median value is to sort the array and then get the value in the middle. But sorting the array is the original problem, so this circular logic is doomed.
* So instead of getting the best pivot by finding the median of all values in the array, we will at least simply try to avoid a bad pivot.
* We will take as our pivot the **median** of three entries in the array:
  + the first entry,
  + the middle entry,

and

* + the last entry.
* One way to accomplish this task is to **sort** only those three entries and use the middle entry of the three as the pivot.
* Figure 11-11 shows an array both before and after its first, middle, and last entries are sorted.
* The pivot is the 5. This pivot selection strategy is called **median-of-three pivot selection.**

Table

Description automatically generated with medium confidence

* This pivot selection scheme assumes that the array has at least three entries.
* If you have only three entries, the pivot selection sorts them, so there is no need for the partition method or for a quick sort.
* Thus, we now assume that the array contains at least four entries.
* The following pseudocode describes how to sort the first, middle, and last entries in an array of at least four entries.
* It will transform the array in Figure 11-11a to the one in Figure 11-11b.

Text

Description automatically generated

***Last Entry***

* During quicksort partition, we move all array entries larger than the pivot to the right side of the array, and all entries less than the pivot to the left side (divide and conquer).
* Here, we just sorted the first, middle, and last entries. Therefore, we know that the last entry is at least as large as the pivot, so it can stay on the right side of the array.
* Thus, we can simply leave the last entry in its place.

***Pivot Entry***

* To get the pivot out of the way, we can swap it with the next-to-last entry, a[last - 1], as Figure 11-12 shows.
* Therefore, the partition algorithm can begin its search from the right at index last - 2.

***First Entry***

* Also notice that the first entry is at least as small as the pivot.
* Thus, we can leave the first entry in place and have the partition algorithm begin its search from the left at index first + 1.
* Figure 11-12b shows the status of the array at this point, just prior to partitioning.

A picture containing table

Description automatically generated

**Partitioning An Array**

* Partitioning an array section about a pivot item is the most difficult part of quick sort.
* The partition function will receive an array segment theArray[first..last] as an argument.
* The function must arrange the items of the array segment into two regions:

1. *S*1 contains the items less than or equal to the pivot *p*

and

1. *S*2 contains the items greater than or equal to the pivot *p*

* Each region has the following properties:

1. All items in *S*1 = theArray[first..pivotIndex - 1] are less than or equal to the pivot *p*.

and

1. All items in *S*2 = theArray[pivotIndex + 1..last] are greater than or equal to the pivot *p*.

* Though these properties do not imply that the array is sorted, they do imply useful facts:

1. The items within *S*1 remain within *S*1 when the array is properly sorted, although their positions relative to one another may change.

similarly,

1. The items within *S*2 will remain within *S*2 when the array is sorted, although their relative positions may change.

finally,

1. The pivot item remains in its position in the final, sorted array.

* Arranging the array items around the pivot *p* generates two smaller sorting problems:

1. Sort the left section of the array (*S1*)

and

1. Sort the right section of the array (*S2*)

* The relationships between the pivot and the array items imply that once you solve the left and right sorting problems, you will have solved the original sorting problem.
* That is, partitioning the array before making the recursive calls places the pivot in its correct position and ensures that when the smaller array segments are sorted their items will be in the proper relation to the rest of the array.
* Also, the quick sort algorithm will eventually terminate: The left and right sorting problems are indeed smaller problems and are each closer than the original sorting problem to the base case—which is an array containing one item—because the pivot is not part of either *S*1 or *S*2.

**Partitioning Algorithm**

* This example assumes that we swap the pivot with the last entry theArray[last] to get it out of the way while we partition the array. Figure 11-10a shows an array after this step.
* Starting at the beginning of the array and moving toward the end (left to right in the figure), look for the first entry that is greater than or equal to the pivot. In Figure 11-10b, that entry is 5 and occurs at the index indexFromLeft.
* In a similar fashion, starting at the next-to-last entry and moving toward the beginning of the array (right to left in the figure), look for the first entry that is less than or equal to the pivot. In Figure 11-10b, that entry is 2 and occurs at the index indexFromRight.
* Now, if indexFromLeft is less than indexFromRight, swap the two entries at those indices. Figure 11-10c shows the result of this step. The 2, which is less than the pivot, has moved toward the beginning of the array, while the 5, which is greater than the pivot, has moved in the opposite direction.
* Continue the searches from the left and from the right.
* Figure 11-10d shows that the search from the left stops at 4 and the search from the right stops at 1. Since indexFromLeft is less than indexFromRight, swap 4 and 1.
* The array now appears as in Figure 11-10e. Entries equal to the pivot are allowed in either piece of the partition.
* Continue the searches again. Figure 11-10f shows that the search from the left stops at 6, while the search from the right goes beyond the 6 to stop at 1. Since indexFromLeft is not less than indexFromRight, no swap is necessary and the searches end.
* The only remaining step is to place the pivot in between the subarrays *S*1 and *S*2 by swapping a[indexFromLeft] and a[last], as Figure 11-10g shows.
* The completed partition appears in Figure 11-10h.

Note that the previous searches must not go beyond the ends of the array.

Text

Description automatically generated with medium confidence

**Entries equal to the pivot**

* Notice that both of the subarrays *S*1 and *S*2 can contain entries equal to the pivot. This might seem a bit strange to you.
* Why not always place any entries that equal the pivot into the same subarray?
* Such a strategy would tend to make one subarray larger than the other.
* However, to enhance the quick sort’s performance, we want the subarrays to be as nearly equal in size as possible.
* Notice that both the search from the left and the search from the right stop when they encounter an entry that equals the pivot.
* This means that rather than leaving such entries in place, they are swapped. It also means that such an entry has a chance of landing in each of the subarrays.

**A function for the quick sort**

* Before creating a function for quick sort, we need to think about small arrays.
* You have seen that the array should contain at least four entries before you call the partition method.
* But simply agreeing to use the quick sort only on large arrays is not enough.
* The code for the quick sort needs to screen out small arrays and use another way to sort them.
* An insertion sort is a good choice for small arrays. In fact, using it instead of the quick sort on arrays of as many as ten entries is reasonable.

The function in Listing 11-5 implements the quick sort with these observations in mind

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated