Quick Sort

This is probably the most common sort used in practice, since it is usually the quickest in practice. It utilizes the idea of a partition (that can be done without an auxiliary array) with recursion to achieve this efficiency.

Quick sort relies on the partition. Basically, a partition works like this:

Given an array of n values, you must randomly pick an element in the array to partition by. Once you have picked this value, you must compare all of the rest of the elements to this value. If they are greater, put them to the “right” of the partition element, and if they are less, put them to the “left” of the partition element.

When you are done with the partition, you KNOW that the partition element is in its CORRECTLY sorted location.

In fact, after you partition an array, you are left with all the elements to the left of the partition element in the array, that still need to be sorted, and all of the elements to the right of the partition element in the array that also need to be sorted. And if you sort those two sides, the entire array will be sorted!

Thus, we have a situation where we can use a partition to break down the sorting problem into two smaller sorting problems. Thus, the code for quick sort, at a real general level looks like:

1) Partition the array with respect to a random element.
2) Sort the left part of the array, using Quick Sort
3) Sort the right part of the array, using Quick Sort.

Once again, since this is a recursive algorithm, we need a base case, that does not make recursive calls. (A terminating condition...) Our terminating condition will be sorting an array of one element. We know that array is already sorted.

However, since this is a Divide-and-Conquer technique, it would be better to identify a threshold value of the subarray based on the size of the original array and use one of the sorting algorithms known.

Here is an illustration of Quick Sort:

How to Partition in Place

Consider the following list of values to be partitioned:

```
8     3     6     9     2     4     7     5
^                                   ^
```

Let us assume for the time being that we are partitioning based on the last element in this array, 5.

Here is how we will partition:

Start two counters, one at array index 0 and the other at array index 6, (which is the second to last element in the array.)

Advance the left counter forward until a value greater than the pivot is encountered.

Advance the right counter backwards until a value less than the pivot is encountered.

After these two steps have been performed, we have:

```
8     3     6     9     2     4     7     5
^                         ^
```

Now, swap these two elements, since we know that they are both on the "wrong" side:

```
4     3     6     9     2     8     7     5
^                         ^
```

Now, continue to advance the counters as before:

```
4     3     6     9     2     8     7     5
            ^           ^
```

Then swap as before:

```
4     3     2     9     6     8     7     5
                  ^
```

When both counters line up (corss), swap the last element with the where the counter is to finish the partition:

```
4     3     2     5     6     8     7     9
                  ^
```

Let us take a look at this algorithm:

```
PARTITION(A,i,j )
 x ←A[i]
 k←i
 l←j+1
 while(k<l)
    do k←k+1 until A[k]>=x /first element that is larger than pivot is identified/
    do l←l-1 until A[l]<=x /first element that is smaller than pivot is identified/

     if(k<l)then
          swap A[k]&A[l]
swap A[i]&A[l]
return l+1
```

Once the location of the pivot is captured, we do

```
quicksort ( A, i, l-1);
quicksort (A, i+1, j);
```

Quick Sort Analysis

This is more difficult than Merge Sort. The reason is that in Merge Sort we always knew we were getting recursive calls with equal sized inputs. But in Quick Sort, each recursive call could have a different sized set of numbers to sort. Here are the three analyses we must do:

1) Best case
2) Average case
3) Worst case

In the **best case**, we get a perfect partition every time. If we let T(n) be the running time of Quick Sorting n elements, then we get:

T(n) = 2T(n/2) + O(n), since partition runs in O(n) time.

This is the same exact recurrence relation as we got from analyzing Merge Sort. Just like that situation, here we find that in the ideal case, QuickSort runs in O(nlogn) time.

Now, consider how **bad Quick Sort** would be if the partition element were always the greatest value of the one remaining to sort. In this situation, we have to run partition n-1 times, the first time comparing n-1 values, then n-2, followed by n-3, etc.

This points to the sum 1+2+3+...+(n-1) which is (n-1)n/2. Thus, the worst case running time is $O(n^2)$.

Now, to the **average case** running time. This is certainly difficult to ascertain because we could get any kind of partition. We will assume that each possible partition (0 and n-1, 1 and n-2, 2 and n-3, etc.) is equally likely.

Assume that you run Quick Sort n times. In doing so, since there are n possible partitions, each equally likely, on average, we have each partition occur once. So we have to list all possible form of recurrences and sum up and do simplifications to get O(n log n).

Quickselect

The selection problem is different than the sorting problem, but is related, nonetheless. In fact, one efficient technique to solving the selection problem is very similar to quicksort. The selection problem is as follows:

Given an array of n elements, determine the kth smallest element. (Clearly k must lie in between 1 and n, inclusive.

The idea for the quickselect is as follows:

Partition the array. Let us say the partition splits the array into two subarrays, one of size m with the m smallest elements and the other of size n - m - 1. If $k \leq m$, then we know that the kth smallest element of the original array will be in the first partition. If k=m+1, then we know our first partition element is the correct value, otherwise, we know that the same will be in the second partition of the original array.

Here is a very basic outline of the algorithm, more formally:

Quickselect(A, low, high, k):

1) m = Partition(A, low, high) // m is how many values are less
   // than the partition element.
2) if $k \leq m$, return Quickselect(low, low+m-1, k)
3) else if k=m+1 return the partition element, A[low+m]
4) else return Quickselect(low+m+1, high, k-m-1)