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.

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 partition 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, swap the last element with the where the counter is to finish the partition:

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

Let's take a look at some code that implements this algorithm:

```
// Arup Guha
// 9/30/02
// Code to demonstrate the Partition algorithm.
import java.util.Random;

public class Sort {

    private int[] values;

    public Sort(int n) {
        values = new int[n];
        Random r = new Random();
        for (int i=0; i<n; i++)
            values[i] = Math.abs(r.nextInt()%100);
    }
```

```
  public int Partition(int start, int end) {

     int i = start;
     int j = end;

     while (i < j) {

        while (i <= end && values[i] <= values[start])
           i++;
        while (values[j] > values[start])
           j--;
        if (i < j)
           swap(i,j);
      }
      swap(start, j);
      return j;
   }

   public void swap(int i, int j) {
       int temp = values[i];
       values[i] = values[j];
       values[j] = temp;
   }
}
```

Median of 3 and 5 Idea for Partition

Finally, since it's important to get a reasonable "split" when doing a quicksort, it's worth going over a couple ideas that ensure a reasonable split of values in the partition step. (I won't show you the code, just the idea. But, you should be able to implement these ideas in code if you ever had to.)

One idea is to randomly pick three elements in the array to be sorted as candidates for the partition element. Then, choose the middle value of these three elements to be the partition. There is some extra expense here - picking three elements and then doing three comparisons to determine the median of the values, but hopefully, if the array being sorted is large enough, this extra expense will be small enough compared to the gains of a better partition element.

Clearly, you would not want to do this if you were only sorting 10 or 20 values. In fact, quicksort is most efficient if you implement some simple sort such as insertion sort when you get down to a few elements, say 10 or 20. (This would be your terminating condition in the recursive method.)

Also, if you wanted to, you could pick 5 random elements to find the median of, and then pick that as the partition element. This can be done in a maximum of 7 comparisons. This will generally give you a better partition element than the median of 3 technique. Depending on the size of the array being sorted, this extra cost may be worth it.

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

We will leave the average case analysis till the end since it's the most difficult.

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 sort 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. One way to work out the math is as follows:

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 the following recurrence relation:

$$nT(n) = T(0)+T(n-1)+T(1)+T(n-2)+...+T(n-1)+T(0) + n*n$$
$$nT(n) = 2[T(1)+T(2)+...T(n-1)] + n^2$$

(The n is for the work done by the partition method, simplified from O(n) to make the analysis easier.)

Now, plug in n-1 in the equation above to get the following one:

$$(n-1)T(n-1) = 2[T(1)+T(2)+...T(n-2)] + (n-1)^2$$

Subtracting these two equations we get:

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n - 1$$
$$nT(n) = (n+1)T(n-1) + (2n - 1)$$
$$T(n) = [(n+1)/n]T(n-1) + (2n - 1)/n$$

Since we are only trying to do an approximate analysis, we will drop the -1 at the end of this equation. Dividing by n+1 yields:

$$T(n)/(n+1) = T(n-1)/n + 2/(n+1)$$

Now, plug in different values of n into this recurrence to form several equations:

$T(n)/(n+1) = T(n-1)/n + 2/(n+1)$
$T(n-1)/(n) = T(n-2)/(n-1) + 2/(n)$
$T(n-2)/(n-1) = T(n-3)/(n-2) + 2/(n-1)$
...
$T(2)/3 = T(1)/2 + 2/1$

Now, adding all of these equations up reveals many identical terms on both sides. In fact, after cancelling identical terms, we are left with:

$T(n)/(n+1) = T(1)/2 + 2[1/1 + 1/2 + 1/3 + ... + 1/(n+1)]$

The sum on the right hand side of the equation is a harmonic number. The nth harmonic number(H_n) is defined as $1 + 1/2 + 1/3 + ... 1/n$.

Through some calculus, it can be shown that $H_n \sim \ln n$. (ln is the natural log. It is a logarithm with the base e. $e \sim 2.718282$.)

Now, we have:

$T(n)/(n+1) \sim 1/2 + 2\ln n$
$T(n) \sim n(\ln n)$, simplifying a bit.

Thus, even in the average case for Quick Sort, we find that $T(n) = O(n \log n)$.

Note, in order analysis, any function of the form $\log_b n = O(\log_c n)$, for all positive constants b and c, greater than 1.

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's 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 was in the first partition. If k=m+1, then we know our first partition element is the correct value, otherwise, we know to search for the kth smallest value 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)

So instead of doing two recursive calls, we only make one here. It turns out that on average, quickselect takes O(n) time, which is far better than it's worst case performance of $O(n^2)$ time.

Simplified Analysis of Quickselect

The analysis of the quickselect is even more involved than quicksort. So, rather than going through all of that, I will go through the best case analysis and a simplified version of the average case analysis. (Worst case is identical to quicksort's worst case.)

In the best case, you find the element right off the bat after doing the partition. Clearly, this is O(n) time.

In the average case, the new subarray you search is about 3/4 the size of the original array. (This is bit of an over estimate, but not by much, simply because if you are given an uneven split, it's far more likely that you'll end up having to search in the larger portion of the array.) Then we get the following recurrence:

$$T(n) = T(3n/4) + O(n)$$

If you look at this carefully, this is a recurrence we can solve. Rewrite as follows:

$$T(n) = T(n/(4/3)) + O(n)$$

Plug in A = 1, B = 4/3, k = 1, thus $B^k = 4/3 > A$. Thus, the solution to this recurrence is O(n).

One way to see this is that if we plug into the recursive formula several times, we see that T(n) = n + 3n/4 + (9n/16) + (27n/64)+... This is bounded by an infinite sum who's value is 4n.