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:

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

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

Here is how we will partition:

Start two counters, one at array index 1 and the other at array index 7, (which is the 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:

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

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

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

Now, continue to advance the counters as before:

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

Then swap as before:

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

When both counters cross over each other, swap the value stored in the original right counter with the pivot element.

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

Now, swap the 2 and 5 to yield:

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

Return the index the 5 is stored in to indicate where the partition element ended up in the array.

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

```
// Arup Guha
// 2/3/04
// Code to demonstrate the Partition algorithm.

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void create(int *values, int n );
int partition(int *values, int start, int end);
void swap(int *a, int *b);
void print(int *values, int n);

int main() {

    int *nums, mid;

    srand(time(0));
    create(nums, 30);
    mid = partition(nums, 0, 29);

    printf("The index of the partition element is %d\n", mid);
    print(nums, 30);
}

void create(int *values, int n ) {
    int i;
    values = malloc(n*sizeof(int));
    for (int i=0; i<n; i++)
        values[i] = rand()%100;
    }
}
```

```
int partition(int *values, int start, int end) {

    // Line up left and right counters.
    int i = start;
    int j = end;

    while (i < j) {

        // Move left counter, then the right counter.
        while (i <= end && values[i] <= values[start])
            i++;
        while (values[j] > values[start])
            j--;

        // Swap out of place values.
        if (i < j)
            swap(values+i, values+j);
    }

    swap(values+start, values+j); // Swap in partition element.
    return j;
}

void swap(int *a, int *b) {
     int temp = *a;
     *a = *b;
     *b = temp;
}

void print(int *values, int n) {
     int i;
     for (i=0; i<n; i++) printf("%d ", values[i]));
     printf("\n");
}
```

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 omit the average case analysis due to its difficulty. You'll see it in CS2 however.

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 $O(n^2)$.