Computer Science 3-4
Problem Set #9: More Array Action!

Sorting is an important part of working with arrays (and really data of any type), so we'll start off by looking at a few different sorting algorithms.

Bubble Sort – In the *bubble sort algorithm*, we make multiple passes through an array. During each pass, we compare adjacent elements of the array to see which is larger. If the elements are out of order, we just swap the elements and then look at the next pair. We sort in terms of increasing order (larger numbers at the end of the array), so if we check a pair and it already is in the proper order we just let it be. Here is an example:

(**5 1** 4 2 8) -> (**1 5** 4 2 8)	Compare first two elements, swap because 5 > 1
(1 **5 4** 2 8) -> (1 **4 5** 2 8)	Compare next two, swap because 5 > 4
(1 4 **5 2** 8) -> (1 4 **2 5** 8)	Compare next two, swap because 5 > 2
(1 4 2 **5 8**) -> (1 4 2 **5 8**)	Compare final two, no swap because 8 > 5

This is "one pass" through the array, you'll notice that it is close, but not all the way sorted. You'll need to run the algorithm at least once more to get the last few elements sorted. The sort stops when it goes through and no elements have been swapped.

Insertion Sort – The bubble sort is an easy to understand sorting algorithm, but honestly not very efficient…it requires a lot of passes through the data and when we have very large sets of data, extra passes really slow us down. This brings us to *insertion sort* which will sort the whole list with one call to the algorithm (though it will have a couple of loops in it). The basic idea is that on our first pass through the loop, we look at the second element of the array and compare it to the first, if it is out of order we "insert" it in front of that first element (you are basically making the second element the first element and pushing all the other numbers up one spot in the array). Our second comparison will look at the 3rd element of the array and "insert" it properly in relation to the first two elements of the array (it will either be in front, in between or just stay put). And then we will continue along until we get to the end of the array.

If you are having a hard time visualizing these sorts, check out www.sorting-algorithms.com where you can see an animation of some of these different sorting techniques (and find some useful pseudocode). Now onto the problems!

1. Write a program that sorts an array of 20 random integers using bubble sort.

2. Bubble Sort Upgrade – You may notice that after the first pass of a bubble sort the largest number is guaranteed to be in the last spot of the array (work this out if you don't believe me!). After the second pass, the two largest numbers will be at the end, etc. Use this info to make your bubble sort a bit more efficient (only check the part of the array that we might have to sort).

3. Implement the insertion sort algorithm to sort a list of 20 random numbers.

4. **Challenge** – add a component to each of your sorts where the user can enter a list of 10 numbers and watch the list sort (print out a list of the numbers as the swaps occur) and lets the user know which items were swapped.

5. **Challenge** – Make a graphical representation of the sorting algorithms (have them sort a list of 10 random numbers from 0 to 15), where you represent each number with a bar of *'s or some other symbol. And have it print out a new version any time the order of the array is changed.

6. Write a program to read an array of 13 integers from a user and compute the median entry of the array.

7. **Challenge –** Write a program that asks the user to type in a series of numbers between 1 and 20 and then computes the mode of the list.