Prepare your answers to the following questions either in a plain text file or in a file that can be opened with Microsoft Word. Submit your file to the Curator system (www.cs.vt.edu/curator) under the heading HW2, by the posted deadline for this assignment. No late submissions will be accepted.

1. [25 points] Many operations can be performed faster on sorted data than on unsorted data. For each of the following operations, state whether it could be performed faster if the data values were sorted (do not take the cost of the sorting into account). No justification is required.

   a) Checking whether a data set contains two words that are an anagrams of each other (e.g., *plum* and *lump*)

   The question does not specify whether the words would be stored in an array or a linked structure.

   If you are given a specific word, say "plum", having the words in a sorted list would offer at least a small advantage. If you generated all 4! different permutations of "plum", you could then quickly see whether each of them did occur within an array of words.

   However, if the question is broader, that is, are there any two words in the set that are anagrams of each other, the computational complexity is going to be enormous in any case.

   As usual, if the values were stored in a linked list, having them sorted would provide at most a tiny advantage.

   b) Finding an item that is at least as large as some specified value.

   Yes, there is a huge advantage if the values are in a sorted array. You only have to consider whether the last element in the list is at least as large as the specified value. (Nothing was said about finding the closest such value.)

   c) Computing the average of a set of integers.

   No. You still need to add them all up, count them, and divide. That's Theta(N) regardless of the ordering.

   d) Finding the median of a set of integers.

   Yes, obviously. If there are an odd number of values, you can just compute the middle index and take that value. If there are an even number of values, you compute the closest-to-middle index and average the two middle-most values.

   Note: even if the data is in a linked list there is still an advantage since you don't have to perform a partial sort to find the median.

   e) Finding the mode of a set of integers (e.g., the value that occurs the largest number of times).

   This can be done by a modified binsort where you increment a counter for each value as you traverse the list (whether or not it's sorted or even in an array); but only if the number of bins is reasonable.

   Otherwise, if the data is sorted we can just make a single linear traversal and count the longest run of identical values; that's Theta(N) and doesn't depend on having a reasonable range for the values, so being sorted will offer an advantage in that case.

2. [25 points] As described in the notes, insertion sort goes sequentially through the array when making comparisons to find the proper place for the element that is currently being processed. Suppose that the sequential search is replaced by a binary search. Will the change decrease the big-Θ cost of insertion sort? Explain.

Well, obviously this would decrease the cost of finding the location to which the "current" element should be moved. The search to find the correct location for the K-th element would average to log K instead of K/2.

But… there would still be an average of K/2 element moves when placing the K-th element, and therefore the total cost will still be Θ(K^2).

3. [25 points] Suppose that two arrays of N integer values, A and B, are in sorted order. Implement a Java method that will find the median value of $A \cup B$ in $O(\log N)$ time; your solution should conform to the interface:

```
int unionMedian(int[] A, int[] B);
```

The obvious approach is to merge the two lists and then take the average of the middle two elements (note the union always contains an even number of values). BUT, the merge would be theta(N), so that doesn't satisfy the problem statement.

The key to doing this in log N time is to make it look like a binary search, and that hinges on the following observations:

Let `medianA` and `medianB` be the medians of the respective lists (which are easily found since both lists are sorted). If `medianA == medianB`, then that's the overall median of the union and we are done. Otherwise, the median of the union must be between `medianA` and `medianB`.

Suppose that `medianA < medianB` (opposite case is entirely similar). Then we need to find the median of the union of the following two sets:

```
{x in A | x >= medianA}     {x in B | x <= medianB}
```

So, we can do this recursively by resetting the "boundaries" of the two arrays. I used a recursive helper function with the following interface:

```
int unionMedian(int[] A, int ALo, int AHi, int[] B, int BLo, int BHi)
```

Applying the logic above, I reset `ALo`, `AHi`, `BLo` and/or `BHi` before each recursive call. And the base cases would be when the two medians are equal and when the two lists are reduced to length 1.

I'm not going to include the actual code here…

4. [25 points] Consider applying radix sort to a list of 10,000 integers in the range from 1,000,000 to 9,999,999.

   a) Not counting the space needed to store the integers themselves, how many bytes of memory would be required? Assume pointers/references occupy 4 bytes. Note: radix sort ALWAYS uses a linked structure to store the elements in a bin.

As described in class, you'd probably use two sets of N bins, N depends on the radix you use and each bin is a singly-linked list.

Each of the 10,000 integers would always have to be in the list in some bin, and singly-linked list nodes are sufficient, so there'd be 10,000 "next" pointers, plus a head pointer for each list.

That makes for a total of 10,000 + N pointers or 40,000 + 4N bytes of memory.

And, if you count a pointer to the array of bins, that would add another 4 bytes.

Note: there is not a unique acceptable answer to this question, and there may be acceptable alternatives that I did not list above…

   b) How many times would radix sort examine each value in the list?

Assuming you use radix 10, since the values all have 7 digits, radix sort would make 7 passes. Each value is examined once per pass.

In general, the number of passes is $\log_b(\text{Max})$ where b is the radix you use and Max is the largest value in the set.