Cosc 2P03: Advanced Data Structures in Java
Assignment #3
Due date: June 22nd, 5:00pm

This assignment covers sorting and light complexity analysis.

Part 1: Sorting

For this part, you will implement five sorting algorithms: Heapsort, Merge Sort, Quicksort (with a catch), Radix sort, and some other sort (see below). Since Radix Sort is not a general purpose sort, generics aren't for a fair comparison in the second part. As such, you can simply implement all five as integer sorters. Thus, each will receive an array of integers (possibly with indices to indicate the range to sort, but that's up to your own preference), and sort the values within the array.

For the Quicksort, you're free to use whichever variant you like. However, you must have it use a different sort when the sub-problems break down to a sufficiently small size. It's up to you whether you want to have that threshold user-selected, or simply hard-coded. I'd suggest Insertion Sort for this part.

For the fifth sort, you may implement whichever deterministic sort that's *guaranteed* to finish/work that you like (that isn't already one of the four listed, of course). Suggestions include: Bubble Sort, Selection Sort, or Insertion Sort. There's no need to choose a 'good' sort, as a 'bad' one will simply make the second part of this assignment more interesting. Yes, you can reuse your code from the sub-problem sorting in the QuickSort.

It is suggested that you simply have your sorts as separate functions within the same class (though a separate class from your test harness). However, you may put each sort into a separate class if you prefer (you may find that this improves readability).

You *must* implement your own *maxheap* for the Heapsort (though you're free to base it on the code from the course textbook).

Bonus: The radix sort mentioned above will use integers. However, it can also be used with letters. For a **small** bonus, implement a radix sort that takes an array of Strings, and sorts them lexicographically. You may assume that only letters will be used (eg. a, j, L, z. No numbers, punctuation, or spaces), but you *must* treat uppercase and lowercase as being equivalent (e.g. *a* doesn't have a higher priority than *A*, and vice versa).
Note: This is more complicated than it initially seems!

Part 2: Analysis

For this part, you are to create a test harness that can use the sorts you implemented in Part 1. You are mostly free to arrange the interface however you like with the following restrictions:

- It must be command-line (no BasicIO)
- It must permit the user to choose the size of the array of values
- It must be able to randomly generate integers to fill those arrays

    - You may ask the user the lower and upper bounds of numbers to generate, or you can hard-code your own bounds (Use numbers between 0 and 10 million if it's hardcoded)
- It should have an option for receiving several integers from the user, sorting them, and then displaying the sorted values.
- For each sorting method being tested for analysis:
    - It should sort the randomly-generated values
    - It should then re-sort the already sorted values
    - You must keep track of how long ***both*** operations took
    - Repeat multiple times (with several random generations) so enough time passes to make the results significant (eg. generate X values, then sort, then re-sort, then generate X values, then sort, then re-sort, etc. repeated enough times for the whole operation to take a couple seconds. Then, repeat the entire procedure for another sorting method)
    - Do your tests with two different array-sizes (eg. 5,000 and 10,000; 1,000 and 10,000; or whatever you feel gives you 'useful' results) so you can see how computation time 'grows'
    - Try to hypothesize as to whether sorting an already-sorted array improves, harms, or has no effect on speed
- Once you've done all your tests, compare the perceived complexity growth against what you expected (ie. against what the algorithms are listed as having for complexity)
- Include some form of writeup, including tables of values

Note: If you did the bonus in Part 1, your test harness will, of course, need to have an option for entering sequences of strings to sort (eg. I should be able to enter "Fred", "free", and "fr", which should sort to "fr", "Fred", and "free"). However, only the radix sort on integers needs to be included in the analysis mentioned above.

Submission Guidelines:
Electronic submission is required. Slap your files into a dedicated directory on sandcastle, ssh (or putty) in, and run 'submit2p03'.
Make sure to include a .pdf (or .txt, if it's readable) copy of your writeup (including complexity analysis), along with sample executions.