COSC 311: ALGORITHMS
HW1: SORTING

Due Friday, September 22, 12pm

In this assignment you will implement several sorting algorithms and compare their relative performance. The sorting algorithms you will consider are:

1. Insertion sort
2. Selection sort
3. Heapsort
4. Mergesort
5. Quicksort

We will discuss all of these algorithms in class.

You should run your experiments on the department servers, `remus/romulus` (if you would like to write your code on another machine that is fine, but make sure you run the actual timing experiments on `remus/romulus`). Instructions for how to access the servers can be found on the CS department web page under "Computing Resources." If you are a Five College student who has previously taken an Amherst CS course or who enrolled during preregistration last spring, you should have an account already set up (you may need to change your password; go to https://www.amherst.edu/help/passwords). If you don't already have an account, you can request one at https://sysaccount.amherst.edu/sysaccount/CoursePetition.asp. It will take a day to create the new account, so please do this right away.

Please type up your responses to the questions below. I recommend using LaTeX, which is a typesetting language that makes it easy to make math look good. If you're not already familiar with it, I encourage you to practice!

Your tasks:

1) Theoretical predictions. Rank the five sorting algorithms in order of how you *expect* their runtimes to compare (fastest to slowest). Your ranking should be based on the asymptotic analysis of the algorithms. You may predict a tie if you think that multiple algorithms have the same runtime. Please give a brief justification for your ranking.

2) Implementation. Begin by copying some files from my directory:

```
$ cp ~kgardner/courses/f17/cs311/sorting/code/* .
```

This will give you a starting point. In particular, I am providing you with:

1. A file called `Sort.java` that includes the following:
   (a) An insertion sort implementation (we looked at this code in class), so you really only need to implement four algorithms.

   (b) Stub methods for each of the remaining four sorting algorithms.
   (c) A main method that provides an example of how to time your sorting algorithms (see part 3 of this assignment).

2. A file called `Heap.java` that gives an implementation of a heap. You may have your own heap implementation from COSC211; please use mine for this assignment.
3. A file called `Generate.class`. This is a compiled Java file that provides the methods that you will use to generate various types of arrays on which to test your sorting algorithms.

After you compile `Sort.java` and `Heap.java`, run the `Sort` program using the command

```
$ java Sort 10000
```

where the parameter (in this case, 10000) is the number of items you want to sort. You can enter any positive integer you want, but if you do not pass a positive integer as a parameter, you'll get a runtime error.

Your job: Fill in the missing methods to implement the remaining four sorting algorithms. You may (and probably should) write some helper methods in addition to the methods whose headers I have provided. You might also want to write a method to check that your sorts are producing correct output (i.e., that you are in fact sorting the arrays).

Note: Implementing mergesort involves creating a temporary array that is used to store data during the merge operation. Repeated creation and garbage collection of arrays every time you recursively call mergesort can significantly add to the running time. It might help you to find ways in which you can reuse arrays.

3) Timing. The file `Sort.java` contains a method called `test()`. This method generates an array of random data of length `N` and times how long it takes to sort the data using insertion sort. The line

```
Generate.randomData(my_array);
```

calls a method that I have written for you to fill an array, passed as an argument, with random unsorted data. The lines

```
long start_time = System.nanoTime();
insertionSort(my_array);
long end_time = System.nanoTime();
```

record the system time before calling the `insertionSort` method, run the method to sort the data, and record the system time after calling the method.

You will notice that in the `test()` method, I have written a loop and timed `insertionSort` several times. This is because a single run on random data can produce significantly different results, depending on the particular initial ordering of the input data. This is especially true when `N` is small. Try setting `num_iters` to 1, and then run the program several times. You likely will see several fairly different timing results. We can reduce this noise by timing the method several times and averaging the results.

Important: Note that I call `Generate.randomData(my_array)` *inside* the loop that runs `insertionSort` several times. This is so that we fill the array with a new set of random data each time we call the method. If we were to call `Generate.randomData(my_array)` before the loop, we would not reset the array in between sorts, and so after the first iteration the data would already be sorted!

Your job: Add code to time all five of the sorting algorithms on different size inputs. You should find that different algorithms are capable of handling very different size inputs. Make a graph comparing the runtimes of the five algorithms. Your x-axis should be `N` (the number of elements you are sorting) and your y-axis should be the average time to sort an array of size `N`, in milliseconds. Be patient, but not too patient. There should be no need to run your code for hours.

Write a short summary of what you observed in your experiments. What did you notice about the relative performance of the different algorithms? Did the actual performance always match up to the ranking you predicted in part (1) based on the asymptotic analysis? Why or why not?

4) Sorting k-sorted data. We say that an array A is *k-sorted* if it has the property that for every element i in array A, i's position in array A is at most k away from i's position in a sorted array of the same data. For example, the array

```
1 4 2 3 5 7 6 8
```

is 2-sorted because every element is within 2 positions of its "correct" position in a sorted array. Any array that is k-sorted is also j-sorted for all $j > k$.

In this part of the assignment, you will study the question of whether it is faster to sort k-sorted data than fully random data. The line

```
Generate.almostSortedData(my_array);
```

fills the array that is passed as an argument with 10-sorted data.

Your job: Repeat your timing experiments from part (3), this time with 10-sorted data instead of random data. Again, make a graph comparing the runtimes of the five algorithms.

Write a short summary of what you observed in your experiments. Did the algorithms have the same ranking on 10-sorted data as on random data? Which algorithm(s) performed significantly differently? (Note: by "significantly," I mean an orders of magnitude difference.) Why?

This assignment is due on Friday, September 22, at 12pm. Please submit your `Sort.java` through the online submission system (www.cs.amherst.edu/submit), and bring a hard copy of your typed responses to submit in class.