INTRODUCTION TO COMPUTER SCIENCE I
LAB 7: SORTING
Friday, March 23, 2018

1 Introduction and Setup

In class on Wednesday, we saw that the runtime for insertion sort is proportional to n^2, where n is the number of elements in the array that we're sorting. I proposed mergesort as an alternative sorting algorithm. Recall that the idea behind mergesort is that we split our unsorted array in half, sort each half, and then merge the two sorted halves back together.

Your goal in this lab is to determine which algorithm—insertion sort or mergesort—is more efficient. Specifically, you will write some code to count up the number of comparisons required for each algorithm, and then run some experiments to see how each algorithm performs on different values of n.

Begin by making a new directory for this lab, changing into that directory, and copying a java file:

```
$ mkdir lab7
$ cd lab7
$ wget -nv --trust-server-names https://goo.gl/Hz8hAi
```

You will see several methods in the starter code. Some of them are complete, others you will need to modify. The `print` and `fillRandom` methods should look familiar to you from Lab 6; I have included these methods (which you should not change in any way) to help you test your code. The `mergesort` and `mergesortHelper` methods are also complete; there is no need to change this code. You will need to make modifications, detailed below, to the `merge` and `insertionSort` methods, as well as write an additional method to test your code.

Note that in several places, there are variables declared that are of type `long`. A `long` is very similar to an `int`, except that its internal representation uses 64 bits (binary digits; that is, 0s and 1s) instead of the 32 bits used to represent an `int`. The consequence of this is that while `int` variables can only store values in the range -2^{31} through $2^{32}-1$, `long` variables can store values in the range -2^{63} through $2^{64}-1$. We're using `long` variables here because as n (the number of elements we are sorting) gets very large, the number of comparisons we need to do could exceed the maximum value we can store in an `int`.

2 Your Tasks

1. **Fill in the missing parts of the `merge` method.** On Wednesday, we discussed how to merge two sorted halves of an array. The idea was to walk down the two halves of the array simultaneously, copying values into a new temporary array as we go. If the value at our current location in the left half is smaller than the value at the current location in the right half, we copy that

value into the next position in the temporary array. Otherwise, we copy the value at our current location in the right half. The `merge` method in `Sort.java` contains most of the code needed to do this. Read through the code and make sure you understand what it is doing (we will talk about the `mergesortHelper` method in class on Monday). Ask questions if you are confused! Then **fill in the missing pieces**, which are marked in the code with the comment `FILL THIS IN`.

2. **Count comparisons for `insertionSort`.**The `insertionSort` method works as it is, and looks almost identical to the code we wrote in class. Your job is to add a few lines of code to count up the number of comparisons that are performed while the `insertionSort` method is running. In the end, instead of returning 0 (as the method currently does), you should return the total number of comparisons.

3. **Run an experiment to compare `insertionSort` and `mergesort`**. Write a method called `experiment` that has no input parameters and no return value, and that does the following:

For each multiple of 10000 up to and including 100000, do:

1. Create an array of length n and fill it with random numbers (you can use the `fillRandom` method that I have provided).
2. Run `insertionSort` on this array.
3. Re-fill the array with a new set of random numbers.
4. Run `mergesort` on this array.
5. Print out n, the number of comparisons required for insertion sort, and the number of comparisons required for mergesort.

For example, your output might look something like:

```
2 1 1
4 5 4
8 14 12
16 62 32
```

where the first column is n (your values of n will be different from this example), the second column is the number of comparisons for insertion sort, and the third column is the number of comparisons for mergesort.

4. **Report the results of your experiment.** You'll need to add some code to your `main` method to actually run your experiment by calling your `experiment` method. After you've done this, answer the following questions in a comment at the top of your `Sort.java` code. Your answers don't need to be longer than a few words or a sentence.

(a) Which algorithm, `insertionSort` or `mergesort`, is more efficient? That is, which requires fewer comparisons to sort the same number of elements?

(b) As n gets very large, does the difference in number of comparisons required for `insertionSort` and `mergesort` increase, decrease, or stay the same?

(c) We said in class that the number of comparisons required for insertion sort is proportional to n^2. Take a guess (based on the results you saw in your experiment, on looking at the code, or on something else) at the order of the number of comparisons required for mergesort. Is it proportional to n^2? To n? To something else?

3 Submit your work

Submit your modified `Sort.java` using either the submission web site or the `cssubmit` command.

This assignment is due on Thursday, March 29, 11:59 pm.

4 Totally Optional Challenge

If you finish this week's lab early and want some extra practice, here is a totally optional challenge problem for you to think about: what other algorithms, besides insertion sort and mergesort, could we use to sort an array of `int`s? We've mentioned a couple of ideas in class. One idea was to repeatedly compare pairs of elements in our array, swapping them if they're out of order, until the entire array is sorted. Another idea was to find the smallest element in the array, move it to the front, and repeat until all elements are in place. There are many others!

Come up with an idea for a sorting algorithm (it can be one of the two suggestions above) and write some code to sort an array of `int`s using your algorithm. Put your new sorting method in a **new java file** (i.e., not in `Sort.java`). Run some experiments to try to figure out how your idea compares to insertion sort and mergesort.

You can submit this new code using the submission web site or the `cssubmit` command; choose the `Lab 7 (OPTIONAL PART)` option.