Lab 3 - Verification of Sorting Algorithm Performance Across Datatypes
Due (via Bitbucket and hard copy) Wednesday, 04 February 2015
50 points

Lab Goals

- Update/extend the SortCompare class to handle new parameters
- Run a new set of trials to ensure that the performance relationships between sorting algorithms that we noted in class hold true for other datatypes as well

Assignment Details

This lab follows from the experimentation that we have done so far with the SortCompare.java file to compare various sorting algorithms. The SortCompare class currently takes in four inputs from the command line: the two sort algorithms that are being compared, the length of the list to sort, and the number of iterations that each sort is performed (e.g. java SortCompare Shell Merge 1000 100 compares Shell Sort and Mergesort, running 100 trials on a list of 1000 items). The output gives a performance ratio between the two algorithms provided – a ratio smaller than 1 means that the second algorithm is faster, whereas a ratio greater than 1 means that the first algorithm is faster.

In the current state of SortCompare, the class creates a list of doubles between 0 and 1 of the selected length, generating these doubles using a uniform distribution (each number in the list is equally likely to occur). In this lab, you will be modifying (or extending) the SortCompare class to add two new command line parameters. The first will allow a user to run trials using the existing Double lists, but also including Integers and Strings. The second will allow the user to change the distribution from uniform to Gaussian.

After you update the code, you will run a few trials on different combinations of these new options, to ensure that the sorting performance comparisons that we noted in class for Doubles still hold true for Integers and Strings. Thus, a full run of the code may look something like java SortCompare Shell Merge 10000 100 String Gaussian when you are finished.

Part One: Sorting Doubles (5 points)

As noted in the lab introduction, the first parameter that we want to alter in SortCompare is to allow a user to select between Doubles, Integers, and Strings. After we add this command line option into the main() function of SortCompare, we certainly want to make sure that comparing Doubles works as it did before. In this section, you should:
1. Add the command line option as noted above, to allow a user to select between sorting Double, Integer, and String datatypes.

2. Compare the performance of Selection Sort vs. Insertion Sort, Insertion Sort vs. Shell Sort, Shell Sort vs. Top-Down Mergesort, and Top-Down Mergesort vs. Bottom-Up Mergesort using Doubles. Here are some additional details:

   (a) For each of these four algorithm comparisons, you should run 5 “good” trials (throwing out any results more than 25% from the mean) for each of the sizes specified below.

   (b) In each of the trials you run, you should do a minimum of 100 iterations, to average out any outliers that may emerge in any of the individual sortings.

   (c) For Selection Sort vs. Insertion Sort, you should run trials on array lengths of 100, 1,000, and 10,000. You can optionally go one order of magnitude higher (to 100,000) if you have time.

   (d) For Insertion Sort vs. Shell Sort, you should run trials on array lengths of 1,000, 10,000, and 100,000. You can optionally go one order of magnitude higher (to 1,000,000) if you have time.

   (e) For both Shell Sort vs. Top-Down Mergesort and Top-Down Mergesort vs. Bottom-Up Mergesort, you should run trials on array lengths of 10,000, 100,000, and 1,000,000. You can optionally go one order of magnitude higher (to 10,000,000) if you have time.

3. After you finish running the trials, you should (as in the previous lab) create tables and graphs showing the performance comparison between these algorithms. The tables should include your performance ratios, as well as the mean and standard deviation across the 5 trials. The graphs should have array length on the x-axis and performance ratio on the y-axis. The x-axis should be plotted logarithmically, but the y-axis can be linear.

**Part Two: Sorting Integers (10 points)**

Once you have ensured that Doubles still perform as intended, you should add Integer sorting to the SortCompare code. The integers that the code generates should range across all valid 32-bit int values, from \(-2^{31}\) to \(2^{31} - 1\).

Once you have implemented Integer support to the code, you should run the trials as noted in the previous section, with the same parameters and table/graph deliverables.

**Part Three: Sorting Strings (10 points)**

After your Integer trials are complete, it is time to move on to Strings. Each String that you generate should have a randomly generated length between 3 and 20 characters, with each character randomly generated in the string.

Once you have implemented String support in the code, you should run the trials as noted in the first section, with the same parameters and table/graph deliverables.
Part Four: Changing the Distribution (10 point)

The SortCompare class as it existed at the beginning of lab called the StdRandom.uniform() function to generate its double values. Also included in the StdRandom class is the gaussian() function. Rather than generating numbers that follow a uniform distribution, this function will follow a Gaussian distribution – a bell curve. You should examine the StdRandom class to ensure that you have a rough understanding of how this function works.

Now, you should add a second new command line argument to the SortCompare program, allowing a user to select between using the Uniform and Gaussian distributions for generating numbers to sort. When creating Strings, you should still use the Gaussian distribution – the lengths of the strings should follow a Gaussian curve with lengths of 3 and 20 being unlikely and lengths of 11-12 being most likely, and with letters in the middle of the alphabet being more likely than letters at the beginning and ending of the alphabet.

For the trials associated with this section, we will keep a constant array length of 10000. You should run trials for each of the algorithm comparisons from the first section (Selection vs. Insertion, etc.), with 5 trials using the Uniform distribution and 5 trials using the Gaussian distribution options. Your tests should cover all of the datatypes as well – show the performance on Doubles, Integers, and Strings. You should once again create charts and graphs showing the running time difference between these two distributions across the three datatype options. The graphs can be bar charts, with the distribution option on the x-axis and the performance ratio on the y-axis.

Part Five: While You Have Some Downtime (15 points)

While you wait for the above program executions to complete, you should have a bit of downtime to answer some additional questions:

1. Trace how:
   (a) The Selection Sort algorithm will sort the array SELECTIONSORT.
   (b) The Insertion Sort algorithm will sort the array INSERTIONSORT.
   (c) The Shell Sort algorithm will sort the array SHELLSORTISSOMUCHFUNYOUGUYS.
   (d) The Top-Down Mergeshot algorithm will sort the array ISNTDATACOLLECTIONFUN.

2. Which sorting algorithm (Selection or Insertion) will run the fastest on an array where all keys are equal? Why?

3. Which sorting algorithm (Selection or Insertion) will run the fastest on an array that is in reverse order? Why?

4. What is the best case input for Shell Sort? Why?

5. (This question best answered after all test runs are complete.) Did you notice any substantial performance differences between runs of Doubles, Integers, and Strings? What about between uniformly distributed and Gaussian-distributed values? Did you think (before experimentation) that you would see any change?
Submission Details

For this lab, please submit a paper copy of everything listed below. Additionally, please upload all of your files to a folder in your BitBucket repository clearly labeled as “Lab 3.”

1. Your source code for your modified or extended SortCompare.java.

2. A well-written document summarizing:
   - Your tables from Parts 1-4
   - Your graphs from Parts 1-4
   - Your answers to the questions in Part 5.

3. An Assignment Information Sheet filled out for your source files.

Please remember that all files that you submit should be your own work, though you are welcome to discuss high-level topics and algorithms with classmates.