CSE373 Spring 2015: Homework 6 - Sort Explorer

In this assignment, you will use a program called Sort Explorer to explore the performance of several of the sorting algorithms we have learned. You will observe the performance of various "secret" implementations and use the results to decide what sorting algorithm is used in each implementation.

Outline

Due Dates, Turn-In, and Rules

This is an individual assignment — no partners.

Assignment Due 11:00PM Wednesday June 4, 2014

Turn in your assignment here

Provided Program

Download SortExplorer.jar, which is a (compiled) Java program that you should be able to run on any computer that has Java installed:

- Most likely you can just (double-)click on the file and it will run.
- You can also run it on a command-line using `java -jar SortExplorer.jar` assuming the terminal where you type commands is in the directory/folder holding the file.
- For (much) more information on running `.jar` files, see here.

The Sorting Algorithms

The Sort Explorer program can run six sorting algorithms, each described here. It does not tell you which algorithm is which, identifying them only by Greek letters. When running one of the algorithms, it tracks the number of comparisons performed, the number of times data items are moved, and the actual time elapsed.

More details on the program and the experimental data it reports:

- The algorithms sort numbers stored in a Java `int []`. Details on how to control the contents and size of the array being sorted are described in the "Using the Program" section.
- The program reports the number of **comparisons** performed. Every time an element in the array is compared to something else (a temporary value, another location in the array), this counts as a comparison.
- The program reports the number of **movements** performed. Any instruction that copies or moves a value from the array either to a temporary location or to another location in the array itself counts as a movement. Note this means doing a "swap" of two values, both in the array, would require three movements: `temp=a[i]; a[i]=a[j]; a[j]=temp;`.
- The program reports the **total time** to do a sort. Time is measured in milliseconds. We recommend you do several runs of a sort on a given array and take the median value. To get more accurate runtimes, you may want to close other programs that might consume a lot of processing power or memory and therefore affect the Sort Explorer runtimes.
- For the comparison and movement counters, you might encounter **overflow** when sorting large arrays. You can notice this by trying increasingly larger array sizes until you see negative values for these counters.

Details on the sorting algorithms:

- One algorithm is **insertion sort**.
- One algorithm is **selection sort**.
- One algorithm is **heap sort**, using a *build heap* operation to create a binary maxheap and writing the values removed from the heap back into the original array from end to beginning. This code uses the original array for everything, like we discussed in class.
- One algorithm is **merge sort**. For each merge of two sub-arrays, it creates a new temporary array, merges into the new array, and then copies back into the original array.
- One algorithm is **quick sort (simple)**. When sorting a range of the array, it uses the first element in the range of the array as the pivot for partitioning. It does *not* use median-of-three for pivot selection and does *not* use a cut-off below which it switches to a different algorithm.
- One algorithm is **quick sort (optimized)**. It uses median-of-three for pivot selection and switches to insertion sort below a cutoff value.

Using the Program

The first step when using Sort Explorer is to enter your 7-digit UW student ID (including any leading zeros) in the box at the top. The assignment of sorting algorithms to Greek letters is different for different student ID numbers and only the course staff knows the mapping. If you do not enter your student ID correctly, you will likely not have the mapping we expect you to, which will lead you to the wrong

conclusions and therefore hurt your grade. **Do not forget to enter your Student ID correctly every time you start the program.**

The second step is to **create an array to sort** following these steps:

1. Choose **how the elements are ordered** from these choices:
    - *InOrder:* Array elements are already in sorted order, i.e., from smallest to largest.
    - *ReverseOrder:* Array elements are in the exact opposite order from sorted, i.e., from largest to smallest.
    - *AlmostOrder:* Array elements are almost in sorted order, but a few elements are in the wrong place.
    - *Random:* The initial order of the elements is chosen at random, so it is probably far from being sorted (or reverse-sorted).
2. Specify the **array size** either using the slider bar or typing directly into the box holding the number.
3. **Click on the Create The List button**. This array is not created until you click the button. When a new array is created, the program fills in the N and DataType fields in the Experimental Results section.

The third step is to **select a sorting algorithm to run**. Clicking one of the buttons with a Greek letter will run one of the sorting algorithms and fill the rest of the experimental results.

For each array you create, you can then sort it many times, using one or more algorithms. Every time you select a sorting algorithm to run, it will run starting with the array in the same configuration it was in when the array was originally created. So while you will want to create many different arrays to sort, for each one you create, you can try all the sorting algorithms (multiple times to get better timing information).

Stack overflow: For large input sizes and/or particular input patterns, some algorithms may run out of call-stack space. You will notice this because you click on the name of a sort and it returns without the experimental results changing. This should return fairly quickly (as opposed to waiting a while for something to finally return), allowing you to hit other buttons (in contrast, you will not be able to hit other buttons while waiting for a long-running algorithm to finish). If you run the `.jar` file from the command line, then you will see stack overflow messages reported there. If you run it by just clicking on the `.jar file`, then you will not see the messages. If you are interested in changing the runtime stack size, then you can do so when running from the command line. For example, to increase the maximum call-stack size to 2MB, type: `java -jar -Xss2m SortExplorer.jar`.

Other output: If you run Sort Explorer from the command-line, then whenever you create an array with 32 elements or fewer (including when you start the program), the contents of the array will be printed below your command line. This is just debugging output that you can ignore.

What to Turn In

Create a file called **answer.txt**. Inside that file, put your student ID in the first line and then the actual names of the sorting algorithms in the order you have identified from the 2nd line to 7th line. (In order from Alpha to Zeta)
Here is the example:

```
0000000
insertion sort
selection sort
heap sort
merge sort
quick sort (simple)
quick sort (optimized)
```

This file is used for auto-grading, so **make sure** the file has the right name and right format.

Turn in a written report, **writeup.pdf**, explaining which sort is which, with strong supporting evidence, by answering *all* the questions below for *all* the implementations of sorting. Make sure your answer in this written report is consistent with the answer.txt.

Questions (1) and (2) require you to produce tables for each sorting algorithm. You will probably find it useful to create your tables in a spreadsheet program and just cut and paste values from the Sort Explorer directly into the spreadsheet. You can create the tables first with blank entries for experiments you need to run; this is a good way to make sure you understand the questions before running experiments. Later, you can transfer your tables into your written report.

1. **Full statistics for input size 16:** Prepare a table (for each sort) showing the runtime, number of comparisons, and number of movements for an array of size 16 and with separate results for each choice of how the elements are initially ordered. The full answer for this question is just one table per sorting implementation, no explanation required.
2. **Runtimes for at least four nontrivial input sizes:** Prepare another table (for each sort) showing (just) the runtime of the sort for *at least* four different non-trivial array sizes for Random, InOrder, and ReverseOrder element orderings. A non-trivial array size is one where the runtime is more than just a few milliseconds. When you can, increase the input size until the runtime takes at least one second. If input sizes are chosen well, they will allow you to see enough variation in the statistics to give you strong evidence to support your answers to questions (3) and (4) below. You do not need to use the same input sizes for every algorithm, since this might not produce the most useful data. The full answer for this question is just one table per sorting implementation, no explanation required.
3. **Worse-case asymptotic run-time:** Your estimation of the *worst case* asymptotic ("big-O") runtime for each sorting implementation. You must gather appropriate data that clearly shows the relevant patterns of the algorithm's runtime. This likely means measuring different algorithms at different input sizes, since some

algorithms are much faster than others. No explanation is required here. However, if the data you collected in question (2) is in conflict or in no way suggesting the runtimes you give for question (3), then you should consider collecting more runtime data. We do not expect curves that show a perfect n log n or anything like that though, just reasonable data.

4. **Identifying the algorithms:** State which sorting implementation uses which algorithm, *along with your reasoning and supporting evidence*. You should base your reasoning on the following:
    - growth rate of the algorithm's runtime as input size grows
    - speed or number of comparisons and data movements of the algorithm compared to the other algorithms
    - changes (or lack thereof) in behavior of the algorithm for different input orderings

    If errors caused by the algorithm caused you to deduce that algorithm, explain why you suspect that algorithm would cause the error.

For question (4), if all your claims are based only on relative speeds of the algorithms ("sort Beta was the fastest, so therefore it is X-Sort"), you will not receive full credit. You will also not receive full credit if you use process-of-elimination to state which algorithm is being used for the last one or two sorts.

A good answer to question (4) is a strong argument. Try for two or, if possible, three strong points that support each algorithm claim. A reason can be based on any of the criteria that are listed above. But if you are using the growth rate as one of your arguments, then include an explanation of *why* you think the growth rate is what you claim it is. For example, *why* do you think a growth rate is quadratic instead of n log n? But there are other things you can use beyond trying to match your runtime data to a particular curve. Look at how the number of comparisons and movements vary both for different types of inputs (InOrder, ReverseOrder, etc.) and across different algorithms. You can also look at how speed varies between different types of inputs. One reason we asked that you collect data for N=16 is that this is a small enough problem size that you may be able to walk through the algorithm and come up with a fairly close idea of how many comparisons or data movements would be required in some cases, which you can then use as an argument.