CS 3345: Data Structures and Introduction to Algorithmic Analysis

Sorting Techniques Due Date: Sun, Mar 12 at 11:30pm

Overview

One of the most important ADTs is the Dictionary and one of the most studied problems is sorting. In this assignment, you will write multiple implementations of sorting algorithms.

In this assignment, you will explore the performance of several of the sorting algorithms we have learned.

This program can run six sorting algorithms, each described here. More details on the program and the experimental data it reports:

- The algorithms sort numbers stored in a Java `int []`.
- 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 **radix 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.

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** by user input

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 sort returns without the experimental results changing. 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 Sorting.java`

Turn in a written report producing a table for comparing sorting algorithm.

1. Prepare a table (for each sort) showing the runtime, number of comparisons, and number of movements for an array of size 128 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. Prepare another table (for each sort) showing (just) the runtime of the sort for *at least* four different non-trivial array sizes. 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. Input sizes will allow you to see enough variation in the statistics to give you strong evidence to support your answers to questions (3) 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. 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, jus reasonable data.

4. Also submit the source code.