CSE 373, Winter 2013
Homework Assignment #7: Sort Detective (25 points)
Due Friday, March 8, 2013, 11:30 PM

This program focuses on understanding and implementing sorting algorithms. Turn in three files named `answers.txt`, `explanation.txt`, and `Quicksort.java` from the Homework section of the course web site. You will need to download the runnable Sort Detective .JAR archive and other support files from the Homework web page.

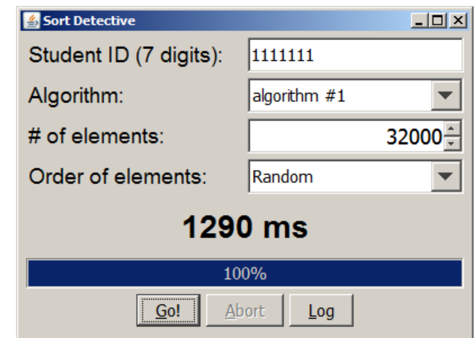
Part A (Sort Detective) Description:

For the first part of this assignment, you will conduct timing experiments to identify anonymous sorting algorithms that we learned in class. We provide you a simulation program called **Sort Detective**. This program lets you run sorting algorithms over an array of integers, but they are listed with generic names like "Algorithm #1". Your task is to figure out which algorithm is which, without seeing the name or source code for the algorithms, based on their external behavior.

The Sort Detective program allows you to specify an unnamed sort algorithm, an input size N for your array of integers, and the ordering of the input. Then you can execute the algorithm, watching to see the running time and behavior of the sort. By altering your input to the Sort Detective, you can make an informed guess which sorting algorithm is which.

The **ten algorithms** in the Sort Detective are the following (details can be found in the lecture slides and textbook):

- *bogo sort*
- *bubble sort* (as implemented in lecture; no optimizations)
- *selection sort*
- *insertion sort*
- *shell sort* (using gaps of $N/2$, $N/4$, ..., 4, 2, 1, as in lecture)
- *merge sort*
- *heap sort* (improved version that builds the heap in-place)
- *quick sort* (using the first element as the pivot for partitioning)
- *bucket sort* (placing element value *i* into bucket *i*)
- *stooge sort* (see below)



The Sort Detective program asks you to type your 7-digit **Student ID** number. You must run the program using your proper student ID number, because it uses the ID to "key" which algorithm is which. The arrangement is different for each student, making sure that there is no incentive to copy other students' answers (since they will be wrong for you). Please double-check your SID number in MyUW to be certain that you are using the right number. If you use the wrong ID number and this causes your answers to be incorrect, you are responsible and will lose points.

Since some sorting algorithms perform differently when given different input, you can choose to have your input array start out in the following **orders** before the algorithm sorts it:

- *Random*: elements are given initial values randomly.
- *Ascending*: elements' values go from smallest to largest; so they are already in sorted order.
- *Mostly Ascending*: a few elements are out of order, but the vast majority (~99%) are in sorted order.
- *Descending*: elements' values go from largest to smallest; so they are in backwards order.

All of the sorting algorithms are ones that you should be familiar with from lecture, except one: **stooge sort**. We're throwing in one new sorting algorithm you haven't seen before to make the discovery slightly more challenging. Stooge sort is a silly sorting algorithm; it compares the first and last element, swapping them if they are out of order; then recursively stooge sorts the first two-thirds of the array, then the last two-thirds, then the first two-thirds again. You can recognize stooge sort by its poor average runtime of $O(N^{2.71})$. Stooge sort can be described by the following pseudocode:

```
function stoogeSort(a, min, max):
    if a[max] < a[min]:
        swap(a, min, max).
    oneThird = (max - min + 1) / 3.
    if oneThird >= 1:
        stoogeSort(a, min, max - oneThird).
        stoogeSort(a, min + oneThird, max).
        stoogeSort(a, min, max - oneThird).
```

Part A Files to Turn In:

Turn in two files containing your solution to Part A:

1) `answers.txt`: This plain text file should contain exactly 11 lines of content. The first line should be the student ID number you used when you ran your tests. The next 10 lines should each contain a name of a sorting algorithm. For example, if you ran the tests with SID #1234567 and you think algorithm #1 is shell sort, algorithm #2 is bogo sort, #3 is insertion sort, etc., your file would contain the following text (abbreviated):

```
1234567
shell sort
bogo sort
...
```

Your file *should not contain any other text or information* of any kind, not even a header with your name. Just list the SID and algorithm names, one per line, exactly as shown, as lowercase strings of the form "***name*** sort". The reason we want you to turn in this file in this format is so that our grading software can check whether you have found the correct answer. To earn full credit for this file, you must have the correct answers and must follow the format specified here.

2) `explanation.txt`: This plain text file should contain a written explanation of how you discovered the answers in your `answers.txt` file. In order to earn full credit for this file, you must include appropriate supporting data and reasoning to back up your claims. The following supporting data is **required, for every sorting algorithm**:

(I) A display of the runtime of each algorithm for several (at least four) nontrivial input sizes. A nontrivial input size means 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 runtime to give you strong evidence to support your answers.

To reduce margin of error, you may want to run the algorithm a few times on each input size and choose the average or median of the three times. You should not necessarily use the same input sizes for every algorithm, since this might not produce the most useful data; but if you intend to compare two algorithms directly by how fast they run, you should use the same input size for that comparison. In order to get the most accurate runtimes possible, you may want to close other programs running on your computer that might be taking a lot of processing power and memory and affecting the Sort Detective runtimes.

The Sort Detective app has a Log button that gives a nice printout of the runs it performs along with the algorithm number, the input size *N*, the order of the input array, and the runtime. You can copy-paste from this log into your report to help support your answers.

Each algorithm should have its own table. Here is an example:

```
Algorithm           N              order    time(ms)
----------------------------------------------------
       #1        1000             Random          79
       #1        2000             Random         219
       #1        4000             Random         681
       #1        8000             Random        1842

       #1        1000          Ascending          50
       #1        2000          Ascending         102
       #1        4000          Ascending         153
       #1        8000          Ascending         198

       #1        1000         Descending         102
       #1        2000         Descending         398
       #1        4000         Descending        1443
       #1        8000         Descending        5017
```

(II) Your estimation of the Big-Oh for each algorithm, based on the data you gathered. 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. Give a tight bound using proper Big-Oh notation, such as `O(N^2)` for $O(N^2)$. Justify your choice based on the runtime data.

If a given algorithm has a **different Big-Oh under different conditions** (for example, with different input orderings), list the best-case, average-case, and worst-case Big-Oh separately as applicable.

(III) Your conclusion of which sorting algorithm it is, along with your reasoning. You should base your reasoning on attributes such as the following:

- growth rate (Big-Oh) of the algorithm's runtime as input size changes
- relative speed of the algorithm compared to the other algorithms
- changes in behavior of the algorithm, if any, for different input orderings
- odd behavior or errors that occur in certain cases (such as Stack Overflow or Out of Memory errors)

If an error caused by the algorithm helped cause you to deduce that algorithm, explain why you suspect that algorithm would cause the error. Do not base your choice of a sorting algorithm entirely on an error; also back up your decision with discussion of the runtime or other properties of the algorithm.

If all of your claims are based only on relative speeds of the algorithms ("Algorithm 5 was fastest, so therefore it's X-Sort"), you will not receive full credit. You will also not receive full credit if you rely on "process of elimination" to identify algorithms ("I already figured out the other nine algorithms, so the last one must be X-Sort").

Please format your file for readability by using spacing and blank lines between paragraphs. For example, place at least one blank line between each algorithm that you discuss, and place a blank line before and after any tables of data included in your file. When we are grading your work, we will view your `explanation.txt` file in a window that is **80 characters** wide. If your file has lines that are wider than 80 characters, the lines will be wrapped at word boundaries. Please keep this in mind as you write your document and avoid lines longer than 80 characters where line wrapping would damage the readability of your work. For example, don't turn in a file containing a table of data that is 150 characters wide, because its appearance will be mangled when we try to view it. Many editors like Eclipse and Notepad++ have an option to see the width of the current line and/or to show a vertical marker for the line width.

This is not an English course, so you generally will not be graded on your grammar and spelling. But if the writing style or formatting of your document makes it so that we are not able to understand your answer, you might not receive full credit. See the class web site for an example stub of an `explanation.txt` file with proper formatting.

Part B (Improved Quick Sort) Description:

In Part B of this assignment, you will write an **improved version of the quick sort algorithm** that was shown in class. Recall that quick sort partitions an array based on a chosen *pivot* value, then recursively sorts the partitions. The version of quick sort written in class sorts an array of integers and always chooses the first element as the pivot. You should write and submit a modified version of this quick sort code that contains the following modifications:

1. **`String[]` instead of `int[]`:** The algorithm should now accept an array of `String`s, rather than an array of integers. Sort the strings alphabetically by their natural ordering as defined in their `compareTo` method.
2. **Median-of-3 pivot:** Rather than choosing the first element as the pivot, you should choose the median value between the first, middle, and last value in the relevant range of the array. For example, if you are sorting an array of length 100, you would examine elements 0, 49, and 99, and choose the median (middle) of these three values as the pivot. For the same example range, if a[0] is "rat", a[49] is "apple" and a[99] is "card", the median (middle) value is "card", so you would use a[99] as the pivot. This pivot selection technique will substantially improve your algorithm's runtime when it is passed an ascending (sorted) or descending (reverse sorted) input. While you are examining the first, middle, and last values to find the median, arrange those three values into ascending relative order in the array. See the textbook and lecture slides for more information.
3. **Insertion sort on small ranges:** If the range of the array to be sorted is **100** (one hundred) elements or fewer, rather than performing an actual quick sort with partitioning, you should instead invoke insertion sort over that portion of the array. This turns out to provide a speed improvement because it is not worth the overhead of all the partitioning and recursive calls when the array is small. You were shown a version of insertion sort in class that sorts an array of integers. You will need to modify this code to process an array of strings, and also to accept a range of indexes and sort only that portion of the array. This way you can call it on a sub-portion of your array without having to copy that range into its own array and waste time and memory.

Specifically, your `Quicksort.java` file should contain the following method, with this exact header:

```
public static void quickSort(String[] a)
```

(You can write additional `private` helper methods to help you implement your quick sort, of course.)

You may assume that the array passed to your quick sort method, and the strings in the array, are not `null`.

We provide a **skeleton file `Quicksort.java`** that contains the headers of the quick sort code you need to implement. You can look at the Lectures page to find the quick sort and insertion sort code that work on ranges of integers. You can use that code as a starting point and modify it to add the improvements previously listed. We also provide a file named `ArrayMethods.java` that contains basic array utility methods, such as a method to create an array of integers or strings to be sorted, and a method to swap values at two indexes of an array.

We suggest that you make one improvement at a time and then test your program, rather than trying to make all three improvements at once. The provided `Quicksort` class contains methods for generating a randomized, ascending, or descending array of Strings. You can use these to test your code. The ascending and descending arrays will run very slowly or crash for large inputs before you add the median-of-3 improvement to your code.

You may want to run your code to watch the changes in runtime as you make each improvement. But you don't need to turn in any gathered data about runtimes for this part; simply turn in the finished Java code file for grading.

Style Guidelines and Grading:

For Part A, we will grade you partly on whether you discover the correct mapping of which sorting algorithm is which, and partly on the soundness of your justification for how you came to this conclusion. Please follow the guidelines in that part of the spec to justify your reasoning using data about the runtimes and behavior of each sorting algorithm.

For Part B, we will grade your code's behavior to give you an **External Correctness** score. We will test you by running your sort algorithm on a variety of input arrays and checking that the array is sorted, as well as checking the runtime and memory usage of the sort.

We will also grade your code quality (**Internal Correctness**). There are many general Java coding styles that you should follow, such as naming, indentation, avoiding redundancy, etc. For a list of these, please see the **Style Guide** document posted on the class web site. You should follow all of its guidelines as appropriate on this and future assignments.

You should implement your sorting routine as a proper implementation of the quick sort algorithm as taught in class. If you base your code on the lecture code provided, you will be fine. You should not use any of Java's built-in data structures (such as the ones from `java.util`) to implement your quick sort code. For example, declaring a field of type `ArrayList` or `HashMap` would be unacceptable. A solution that disobeys these restrictions will receive a deduction. You also may not call any Java class library methods related to sorting, such as `Arrays.sort`.

Your `Quicksort` file should not need any fields or any static global variables.

Include **comment headers** on each class and each method, and also make sure to include inline comments next to any complex or tricky code, briefly explaining the purpose of that code.

This document and its contents are copyright © University of Washington. All rights reserved.