CSE 373, Winter 2011
Written Homework #2: Fun A-SORT-ment (40 points)
Due Wednesday, January 26, 2011, Beginning of Class

This assignment explores several of the different sorting algorithms we have learned. In Part A of the assignment you gain experience with the performance of different sorting algorithms by using a program called the Sort Detective. The Sort Detective allows you to specify an unnamed sort, an input size, and the order of the elements and returns to you the running time of the sort. By altering your input to the Sort Detective, you can make an informed guess which sort is being used. In Part B of this assignment you will gain experience with implementing a variation of a sorting algorithm we learned about in lecture.

This assignment should be worked on individually. Both parts of the assignment should be turned in on paper in class.

Part A: Sort Detective (25 points)

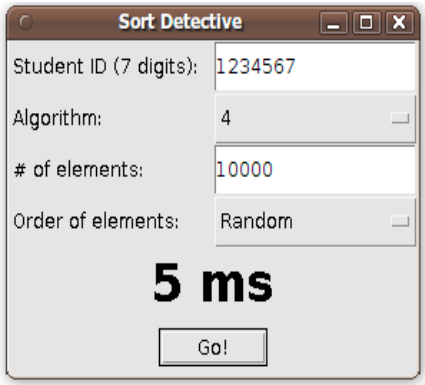
Background Information

This part of the assignment asks you to evaluate anonymous sorting algorithms, based only on their external behavior, to figure out which is which. The algorithms are the following:

- Bogo sort
- Bubble sort (not optimized – bubbling even when no swaps were made during the previous pass)
- Selection sort
- Insertion sort
- Shell sort (using Shell's increments; much like Figure 7.4 on pg. 251 in the textbook)
- Merge sort
- Quick sort (using the first element in the range as the pivot for partitioning)

You will be provided with a testing program that you can run. The program will let you run any of the algorithms. It doesn't tell you which algorithm is which; it only refers to them by a number. Your goal is to run the program several times with different parameters, observe the behavior of the sorting algorithms, and discover which is which.

When you run the testing program, you will see a user interface asking for several pieces of information. You may enter the following information:



- **Student ID**: You must enter your 7-digit student ID number before running any of the sorts.
- **Algorithm**: You can choose any of the anonymous sorting algorithms to run.
- **# of elements**: Type in how many elements should be in the array to sort. (You must type a non-negative number in this box before running any of the sorts.)
- **Order of elements**: Since some sorting algorithms perform differently when given different input, you can choose to have your input array in the following orders:
    - *Random*: elements are given initial values randomly.
    - *Ascending*: elements' values go from smallest to largest; so they are already in sorted order.
    - *Descending*: elements' values go from largest to smallest; so they are in backwards order.

What to turn in

Turn in a written report explaining which sort is which, including supporting data to back up your claims. The following supporting data is required, for every sorting algorithm:

1. A table 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 (2) and (3) below.

   To reduce margin of error, run the algorithm a few times on each input size and choose the median of the three times. You need not 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.

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

	Average runtime for three runs (ms)		
Input Size	Random	Ascending	Descending
10000	79	50	102
20000	219	102	398
30000	681	153	1443
40000	1842	198	5017

2. 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.
3. Your statement of which sorting algorithm it is, along with your reasoning. You should base your reasoning on the following:
   - growth rate (Big-Oh) of the algorithm's runtime as input size changes
   - speed of the algorithm compared to the other algorithms
   - changes in behavior of the algorithm, if any, 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.

If all of your claims are based only on relative speeds of the algorithms ("Algorithm 5 was the fastest, so therefore it's 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 ("I have already figured out the first six algorithms, so the last algorithm must be X-Sort").

Part B: Selection Sort Variation (15 Points)

There are many different tweaks that can be made to standard sorting algorithms to increase their performance. In this part of the assignment you will explore one such tweak to the selection sort algorithm that was presented in lecture.

Write a Java method `dualSelectionSort` that takes an array of integers and sorts their values by using a variation of selection sort that takes into account both the smallest and the largest values in the list on each pass. More specifically, your code should implement the following algorithm:

Find the minimum and the maximum value in the list.

Swap the minimum value with the value in the first position.

Swap the maximum value with the value in the last position.

Repeat the steps above for the remainder of the list (starting at the second position and ending at the second to last position and narrowing the range of positions examined from both ends of the array each time).

For example, if `arr = {27, 63, 1, 72, 64, 58, 14, 9}`, then the call `dualSelectionSort(arr)` would make the following passes over `arr` to sort it:

Index	0	1	2	3	4	5	6	7
Value	27	63	1	72	64	58	14	9
1st Pass	**1**	63	**27**	**9**	64	58	14	**72**
2nd Pass	1	**9**	27	**63**	**14**	58	**64**	72
3rd Pass	1	9	**14**	**58**	**27**	**63**	64	72
4th Pass	1	9	14	**27**	**58**	63	64	72

After implementing your method in Java and ensuring that it is correct, compare your `dualSelectionSort` method and the `selectionSort` method presented in lecture by running timing tests with arrays of different sizes. To help you create random sorted arrays and run your timing tests, use the method `createRandomArray` (shown below) and `System.nanoTime()` (as seen in lecture). Answer the following questions:

- What array sizes did you use to compare the two algorithms and why?
- What were the runtimes of each array size with each sort? It might be useful to organize your data in a table.
- Does `dualSelectionSort` provide an improvement to `selectionSort`? If so, describe the improvement that you see.
- Does `dualSelectionSort`'s growth rate (i.e. Big-Oh) appear to be the same as or different than `selectionSort`'s? Explain your answer based on the runtimes you observed.

What to turn in: Turn in your `dualSelectionSort` method by printing it out and attaching it to your homework or by writing it on your homework pages. For the analysis, you only need to turn in the answers to the four questions above; you don't need to turn in your time testing code.

```
import java.util.Random;
...
public static int[] createRandomArray(int size) {
    int[] array = new int[size];
    Random rand = new Random();

    // fill it with random data in [0, size]
    for (int i = 0; i < size; i++) {
        // pick random numbers (subtract a bit so that some
        // are negative)
        array[i] = rand.nextInt(size * 3) - size / 4;
    }

    return array;
}
```