NORTHWEST
MISSOURI STATE UNIVERSITY

CSIS 44-542 Object-Oriented Programming
Spring 2017

Programming Assignment 2

This is an out-of-class programming assignment. It is to be done individually. The assignment is to be submitted as a single zip file via canvas by **5pm Friday, March 17, 2017**.

The purpose of this practical is to help you understand how different algorithms can perform the same task, but at differing levels of efficiency. We will examine tree sorting techniques – selection sort, bubble sort and insertion sort. You are required to write procedures to sort an array of numbers (up to 1,000) and then compare the behaviour of each of the sorting techniques by timing how long each takes to sort a list of numbers. You will run this test several times in order to gain an accurate estimate of the efficiency of each algorithm. In addition, you are required to write a report on your findings and offer suggestions as to why you think these differences occur.

To compare the three sorting routines, you will need to write a driver program that exercises them in precisely the same way. The driver program should create an array of up to 1,000 elements (integers) and then sort the arrat using each of the three sorting algorithms. We will need to time how long it takes to perform the sorting operation in each case. You can find the elapsed time by using the method `System.currentTimeMillis()` before the execution of the sort routine and again after its completion. The difference between them is the number of milliseconds taken to execute the sorting routine. We are also interested in couting the number of elements hat are swapped inside the array as we move data around to sort the information in the list.

Each sorting algorithm is described below. It is recommended that you use pen and paper and manually follow the algorithm to sort a list of, say, 5 numbers in order to make sure you understand each algorithm before you code them.

Sorting by Selection
Scan the list and locate the record with the smallest key. Then swap this record with the first record of the list. The first element is now clearly in its proper place.

Now scan the list for the record with the second lowest key. Swap this with the second element.

Continue by looking for the third lowest element and swapping it with the record in the third position, and so on, up to the second-to-last element. (Why not the last?)

This algorithm may call for you to swap a record with itself. If this occurs, count this as a swap operation. A practical algorithm is likely to make these swaps. It would be quicker to make the occasional wasted swap than to have to check every step to see if it was worthwhile.

Sorting by Insertion

The idea behind insertion sorting is to have a sorted list of records that grows by adding new records one at a time. The sorted list is kept in order by inserting each new record into its proper place in the sequence.

The first record, considered on its own, is a sorted list of one item that is clearly in no need of sorting. Extend the sorted list to two items by including the second element of the original list. If necessary, swap the two items in the sorted list.

Now include the third element of the original list, increasing the size of the sorted list to three items. If the third element has a smaller key than the second, swap them. If the third element has now moved to second place, consider whether it should now be swapped with the first record.

Continue by adding the fourth item to the sorted list. Consider whether it should be swapped with the third item. Then continue comparing and swapping until the new item has reached its proper place.

Carry on until all the records have been included in the sorted list. When the sorted list contains all of the records, exit the program.

This algorithm will never attempt to swap an element with itself as we saw in selection sort.

Bubble Sort

The bubble sort, in its simplest form, always makes the same series of comparisons between records. You should make a series of "passes", as follows:

- Compare the keys of the second-to-last and last records. If they are out of sequence, swap them.
- Then compare the third-to-last and second-tolast records, and swap them if necessary. Continue with the fourth-to-last and third-to-last records, and so on, down to the first and second. At the end of this pass, the smallest key should have bubbled up to the top of the list (first position).
- Make a second pass by starting with the secod-to-last and last records, but this time, continue only until you have compared the second and third keys.
- The third pass begins by comparing the second-to-last and last records, and continues as far as the third and fourth. Continue in this way for a total of $n-1$ passes for a lit of length n.
- The final pass should merely compare the second-to-last and last records.

Swaps should always occur between consecutive items.

In this assignment you are asked to design and write the code to:

1. For the 3 data sets needed:
    a. Populate an array of 1,000 integers with information.
    b. For each sorting algorithm:
        i. Copy the array from step 1 into a second array.
        ii. Sort the second array noting the time taken to sort it and the number of swaps that were perfomed.
        iii. Display the time taken and number of swaps completed

The 3 data sets needed are: in order, reverse order, and random order. That is to say in the first pass we create an array of 1,000 integers and insert the numbers 1 to 1,000 in the array in sorted order A(1) = 1, A(2) = 2, … A(1000) = 1000, and then see how long t takes each algorithm to sort a list that is already sorted. In the second pass we insert the numbers 1 to 1,000 into the array in reverse order so that A(1) = 1000, A(2) = 999, … A(1000) = 1, and then see how long it takes to sort the list with the list is initially in reverse order. Finally, we insert the numbers 1 to 1,000 into the array at random locations and see how long it takes to sort this list. Note that in each case it is important that the sorting algorithm be given exactly the same initial values in the array in order for us to validly compare them.

Complete the following table:

	Selection Sort		Insertion Sort		Bubble Sort	
Data	time	swaps	time	swaps	time	swaps
In order						
Reverse order						
Random order						

You will notice some differences between the algorithms and that their performance is dependent on the order of the input data in some cases. Write a short report (a page or two) describing why you think this is the case, and how knowledge of the behavior of the sort routines would help you select the most appropriate algorithm. For example, if you had to sort a list that was almost sorted alreay, but had a small nuber of values out of order, which is likely to be the best algorithm, of the three considered, to sort the list.

Solutions sourced from the internet or other sources are *not* permitted. The code is to be carefully tested and submitted via Canvas in a single zip file. The zip file is to include:

- The source files.
- The report containing our speculation as to why you observe the difference in execution time.

The source code is to be well written (follow a coding standard), well documented (use appropriate JavaDoc statements and comments), and use meaningful identifier names, All source code must begin with the following comments verifying that the source code is your own work:

```
/**
 * I certify that all code in this file is my own work.
 * This code is submitted as the solution to Assignment 1
 * in CSIS44542 Object-Oriented Programming, 2017, section <Your section number>
 *
 * Due date: 5pm, Friday, March 17, 2017.
 *
 * @author  <Your Name>
*/
```

Michael Oudshoorn
January 2017