CSCI 2110 Data Structures and Algorithms
Laboratory No. 3
Week of October 2-6, 2023

Due: Sunday, October 8, 11.59 PM

Algorithm Complexity Analysis (Cont'd.)

In this lab, you will continue your experimentation with algorithm complexity by implementing two sorting algorithms. Sorting can take a long time, especially on large or unoptimized data. Two such algorithms with quadratic complexity are Bubble Sort and Insertion Sort. You will implement these two algorithms and test them on arrays with randomly generated integers.

As with Lab No. 3, you can use the following code template to obtain the execution time of your code.

```
long startTime, endTime, executionTime;
startTime = System.currentTimeMillis();

//code snippet (or call to the method) here

endTime = System.currentTimeMillis();
executionTime = endTime - startTime;
```

The above code will give the time for executing the code snippet in milliseconds. You can display the executionTime using System.out.println and/or save it.

Note: The execution times shown in your output file will differ from those in the sample outputs, as well as from the times captured by your peers during their own experiments. Slight differences related to system architecture, computational resources, load, etc. make it very unlikely that two students will submit identical outputs.

Marking Scheme
Each exercise carries 10 points.
Your final score will be scaled down to a value out of 10. For example, if there are three exercises and you score 9/10, 10/10 and 8/10 on the three exercises, your total is 27/30 or 9/10.
Error checking: Unless otherwise specified, you may assume that the user enters the correct data types and the correct number of input entries, that is, you need not check for errors on input

Submission Requirements:
- No submission other than a single ZIP file will be accepted.
- You MUST SUBMIT .java files that are readable by the TA. If you submit files that are unreadable such as .class file, the lab will be marked 0.
- Please additionally comment out package specifiers.

What to submit:
- One ZIP file containing all source code (files with .java suffixes) and a text/word document with sample inputs and outputs, and graphs. The complete list of submission items is given at the end of this document.

You MUST SUBMIT .java files that are executable by your TAs. If you submit files that are unreadable such as .class, you will lose points. Please additionally comment out package specifiers.

Late Submission Penalty: The lab is due on Sunday at 11.59 PM. Late submissions up to 5 hours (4.59 AM on Monday) will be accepted without penalty. After that, there will be a 10% late penalty per day on the mark obtained. For example, if you submit the lab on Monday at 12 noon and your score is 8/10, it will be reduced to 7.2/10. Submissions past two days (48 hours) after the grace submission time, that is, submission past 4.59 AM Wednesday will not be accepted.

Exercise1: Bubble Sort

Bubble Sort makes a number of passes through a given list and sorts the list by comparing pairs of elements at a time. It is a rather inefficient sorting algorithm, but it serves the purpose of experimenting with complexity to see how the run time increases with increase in input size.

Suppose we have an array of integers arr: [2, 9, 5, 4, 8, 1, 9]. Assume that the integers must be sorted in increasing order. Here's a simple description of Bubble Sort and the pseudo code for the algorithm.

Source: Introduction to Java Programming by Daniel Y. Liang, Pearson Publishers, 10th Edition

Compare the elements in the first pair (2 and 9), and no swap is needed because they are already in order. Compare the elements in the second pair (9 and 5), and swap 9 with 5 because 9 is greater than 5. Compare the elements in the third pair (9 and 4), and swap 9 with 4. Compare the elements in the fourth pair (9 and 8), and swap 9 with 8. Compare the elements in the fifth pair (9 and 1), and swap 9 with 1. The pairs being compared are highlighted and the numbers already sorted are italicized in Figure 23.3.

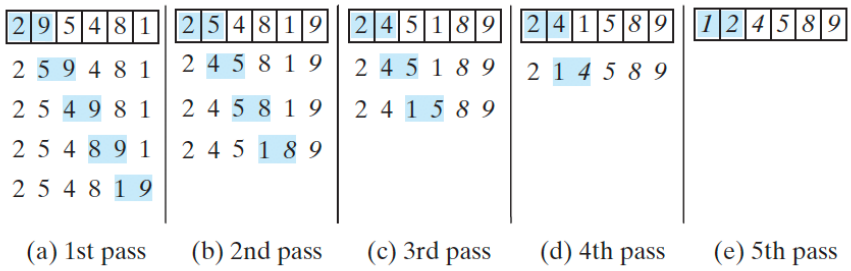

FIGURE 23.3 Each pass compares and orders the pairs of elements sequentially.

As you can see from the example, certain comparisons are not necessary in successive passes. For example, in the second pass, we need not compare arr[5] and arr[6] since that pair is already sorted; in the third pass, we need not compare arr[4] to arr[6] since they are already sorted and so on.

Here's the pseudocode for BubbleSort:

```
for (int k = 1; k < list.length; k++) {
// Perform the kth pass
for (int i = 0; i < list.length - k; i++) {
            if (list[i] > list[i + 1])
            swap list[i] with list[i + 1];
            }
}
```

Write a program that will create an array of size n (where n = 100, 1000, 10000, 100000), generate random integers between 1 and n and store them in the array, and run the BubbleSort algorithm to sort the integers, and determine how long it takes for the program to sort the numbers. Repetition of numbers within the array is OK.

The skeleton of your BubbleSort Class could look something like the code shared below.

```java
/*
Bubble Sort
This class tests the code for Lab4: Exercise1. It calls the sort method to
sort an array of size n and prints information about running time.
*/

import java.util.*;

public class BubbleSort{
  public static void main(String[] args){
    //TODO
    //prompt the user to enter the value of n
   //create an integer array of size n with random integers
  //the range of random integers is from 1 to n

  long startTime, endTime, executionTime;
  startTime = System.currentTimeMillis();

  //call to the sort method

  endTime = System.currentTimeMillis();
  executionTime = endTime - startTime;

//display the executionTIme

}

  public static int[] sort(int[] arr){
    //TODO
   //sort and return the integer arr of size n
  }
}
```

Enter the results in a table (n vs. execution times) and save it in a text/word document, and plot the data on a simple graph. You can use LibreOffice Calc, Microsoft Excel, or a similar program to draw your graph. You have access to a simple but powerful plotting tool called GNUPlot on Unix systems, that will automatically create graphs for you. You may optionally choose to plot your points manually and scan your work for submission.

Exercise2: Selection Sort

Suppose we have an array of integers arr: [12, 19, 5, 14, 8, 1, 9]. Assume that the integers must be sorted in increasing order. Here's a simple description of Selection Sort and the pseudo code for the algorithm.

1. In the first pass, scan the array from arr[0] to arr[n-1] to find the smallest element (1). Swap it with arr[0].
[**1**, 19, 5, 14, 8, **12**, 9]
Now 1 is in the correct position.

2. In the next pass, scan the array from arr[1] to arr[n-1] to find the (next) smallest element. Swap it with arr[1].
[1, **5, 19,** 14, 8, 12, 9]
Now 1 and 5 are in the correct positions.

Repeat the steps similar to the above, in each case finding the smallest element in smaller portions of the array.
3. After the third pass, the first three numbers are in the correct place:
[**1, 5, 8,** 14, 19, 12, 9]

4. After the fourth pass:
[**1, 5, 8, 9,** 19, 12, 14]

5. After the fifth pass:
[**1, 5, 8, 9, 12,** 19, 14]

6. After the sixth pass:
[**1, 5, 8, 9, 12, 14, 19**]

Here's the pseudocode for Selection Sort:

```
SelectionSort(arr):
    n = length of arr
    for i from 0 to n-1:
        min_index = i
        for j from i+1 to n-1:
            if arr[j] < arr[min_index]:
                min_index = j
        swap arr[i] with arr[min_index]
```

Write a program that will create an array of size n (where n = 100, 1000, 10000, 100000), generate random integers between 1 and n and store them in the array, and run the Selection Sort algorithm to sort the integers, and determine how long it takes for the program to sort the numbers. Repetition of numbers within the array is OK.

The skeleton of your SelectionSort class could look something like the code shared below.

```
/*
Selection Sort
This class tests the code for Lab4: Exercise2. It calls the sort method to
sort an array of size n and prints information about running time.
*/

import java.util.*;

public class SelectionSort{
  public static void main(String[] args){
    //TODO
    //prompt the user to enter the value of n
   //create an integer array of size n with random integers
  //the range of random integers is from 1 to n

  long startTime, endTime, executionTime;
  startTime = System.currentTimeMillis();

  //call to the sort method

  endTime = System.currentTimeMillis();
  executionTime = endTime - startTime;
```

```
//display the executionTIme

}

  public static int[] sort(int[] arr){
    //TODO
   //sort and return the integer arr of size n
  }
}
```

Enter the results in a table (n vs. execution times) and save it in a text/word document, and plot the data on a simple graph. You can use LibreOffice Calc, Microsoft Excel, or a similar program to draw your graph. You have access to a simple but powerful plotting tool called GNUPlot on Unix systems, that will automatically create graphs for you. You may optionally choose to plot your points manually and scan your work for submission.

To sort an array with n integers, both the sorting algorithms takes (n-1) passes, and each pass requires a linear search of a max of n items. Therefore, their overall complexity is $O(n^2)$.

What to submit:
One zip file containing the following:
1. BubbleSort.java
2. SelectionSort.java
3. Text/word document containing the tables of experimental results and graphs