CSCI 2110 Data Structures and Algorithms
Fall 2024
Laboratory No. 4
Week of September 30 – October 4

Due (on Brightspace): Sunday, October 6, 11.59 PM

Algorithm Complexity Analysis (cont'd.)

Note: Monday, September 30th is a holiday. There will be no lab session on this day. Students from Monday's lab can attend any other session during the week.

In this lab, you will continue your experimentation with algorithm complexity. Your task is to write, compile and run each program using an Integrated Development Environment (IDE). You may use any IDE of your choice – IntelliJ, Eclipse, NetBeans, VSCode, etc.

Marking Scheme: Throughout this course, we will be focusing on the complexity of algorithms and consequently, the efficiency of programs. In your first-year courses, the focus was on creating runnable code to specifications. This course builds on previous knowledge, but also focuses on efficiency. This will involve reducing unnecessary coding and designing or choosing good algorithms.

Each exercise will list the test cases that you need to test the code. Ensure that the output that you generate covers all the test cases.
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.

Commenting your code: Your code should be clear and easy to understand. While extensive comments are not required, you are expected to include brief, meaningful comments to explain complex logic or any non-obvious sections of your code. Use inline comments to clarify key parts of the code. For methods and classes, use Javadoc to briefly describe their purpose, parameters, and return values. This ensures others can understand your code's intent quickly.

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.

Suggested coding practice: Don't try to tackle the entire exercise in one go. Code a little, test a little, code a little, test a little. You will have better progress if you frequently compile and test your code (every 5-10 lines).

Submission: One zip file containing the source codes (.java files) for all classes, and sample outputs. Please see details at the end of this document.

Submission Deadline: Sunday, October 6, 2024, 11.59 PM

Grace Time: Submissions will be accepted until 4.59 AM on Monday, October 7, 2024 without late penalty.

Late Penalty: Submissions received after the grace time will be subject to a 10% per day late penalty. The submission portal will close on Wednesday, October 9, 4.59 AM.

Dropping of Labs: Up to 2 labs can be dropped during the semester. No SDA submission is required.

<u>Algorithm Complexity Analysis (cont'd.)</u>

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.

<u>Exercise1: Bubble Sort</u>

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 from the book *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.

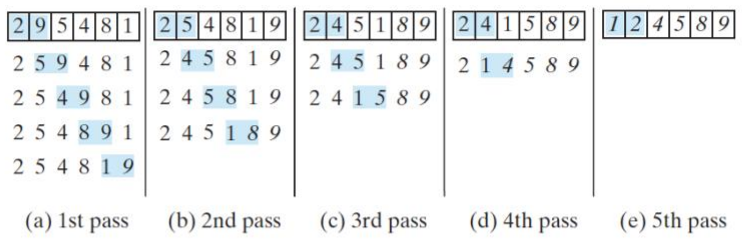

FIGURE 23.3 Each pass compares and orders the pairs of elements sequentially.

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 through the array.
    // With each pass, the largest element gets placed in its final position.
    // Therefore, we can reduce the range of the inner loop by k, as the last k elements are
//already sorted.

    for (int i = 0; i < list.length - k; i++) {
        // Compare adjacent elements in the array.
        // The inner loop iterates through the unsorted part of the array.

        if (list[i] > list[i + 1]) {
            // If the current element is greater than the next element, swap them.
            // This "bubbles" the larger element up towards the end of the array.
            swap(list[i], 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.

```
/*
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    // Get the length of the array

    // Outer loop to traverse the array
    for i from 0 to n-1:
        min_index = i    // Assume the current element is the minimum

        // Inner loop to find the minimum element in the unsorted part of the array
        for j from i+1 to n: // Corrected to go up to n, not n-1
            if arr[j] < arr[min_index]:    // If we find a smaller element
                min_index = j              // Update min_index to the new minimum element's index

        // Swap the found minimum element with the element at index i
        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).

<u>**Submission items**</u>
One zip file containing the following:
1. BubbleSort.java
2. SelectionSort.java
3. Text/word document containing the tables of experimental results and graphs

At the top of each .java file, put the following information in comments.
// Full Name: [Your Full Name]
// ID Number: [Your Banner ID]

You MUST SUBMIT .java files that are readable by the TA. If you submit files that are unreadable such as .class, you will lose points.