Irwin Frimpong –September 16th, 2018 – CS 150 Lab 3

Lab Notes

Introduction

The purpose of this lab was to introduce different sorting and searching algorithms which included insertion and bubble sort as well as binary and linear search. These algorithms were performed on an array list, where the goal was to measure the time it took for each algorithm to run. These results are then written to a CSV file using a printWriter class to be further analyzed graphically. I hypothesize that insertionSortofUnsorted and bubbleSortofUnsorted (which is the worst case) would have runtime of O(n^2) and insertionSortofSorted as well as bubbleSortofSorted should have a run time of O(n) since the array list is already sorted and no inversions are made, but it will need to iterate through to ensure that the array list is indeed sorted. Both linearSearch methods in the Experiment controller should have a runtime of O(n) because the array list would still have to be iterated through to determine whether a string exists. Lastly both binary Search methods should have a runtime of O(logn).

Method

This program consisted of three classes: String Container, Random String Generator, and Experiment Controller.

The String Container consisted of one array list manipulation method, addtoEnd which is responsible for adding a new string to the end of a defined array list. This method is further called in the Experiment Controller class for the time calculations. Insertion sort (algorithm taken from the textbook) and Bubble sort (algorithm taken from lecture) are the two sorting algorithms present in this class which are responsible for alphabetically sorting the strings that are inputted in array.
Linear search and Binary search were two methods employed to find specific strings that may or may not be in the array list. These methods performed the same function but differed in the sense that Linear Search is iterating over the entire array list, hence the for loop in my program, until a match is found. If there is no match made, the index of -1 is returned. In the case of the Binary Search, the target string entered by the user is compared to the middle most element in the array list, if a match is made, that index is returned. In the case of a match not being made in the first iteration of the while loop, the target string is either compared to string to the left or right of itself depending on the values returned by the if statements in the while loop. The middle is recalculated, and the process repeats itself until a match is made.

The RandomStringGenerator class consists of a next String method which is responsible for generating random integers that are further casted to char to be stored in an array of chars. The formula *random.nextInt(((max-min)+1))+ min;* was derived with the help of a CS.gethelp() tutor which ensured that the random integer values generated remain between 97 & 122 inclusive to ensure that the characters generated fell between lowercase a and z. I originally created an algorithm to achieve this result, however this was very inefficient in the sense that the random function kept generating numbers until it fell between the range I specified with the if statement, then will it be added to the array of chars. This program can be found commented in the RSG class.

The Experiment Controller class consists of eight methods that calculate the time it takes for each of the methods defined in String Container to run. Each of these methods have two parameters, numofi(which is the number of inputs) and seed (used to essentially control the random function). In each method, an instance of RandomStringGenerator and StringContainer is created. A for loop loops to the numofi index, populating the array list with random strings, hence the call exp1.nextString(). The code that follows is specific to each method.

Unit tests

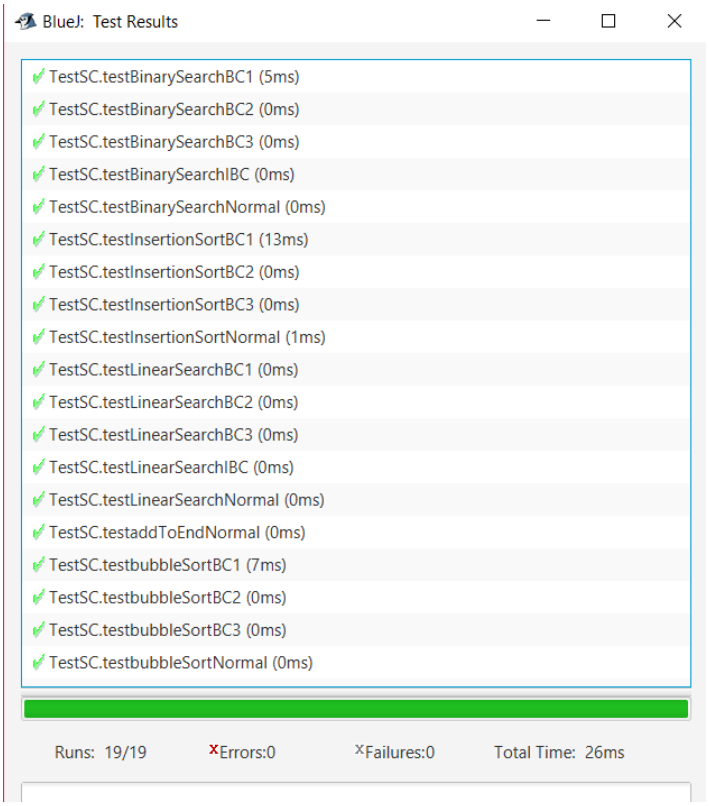

Figure 1: Unit Test of Methods In String Container: insertionSort(), bubbleSort(), linearSearch(), binarySearch()

Figure 1 details the unit test results run on the four methods in the String Container class, testing the normal cases, boundary conditions, and the invalid boundary conditions.

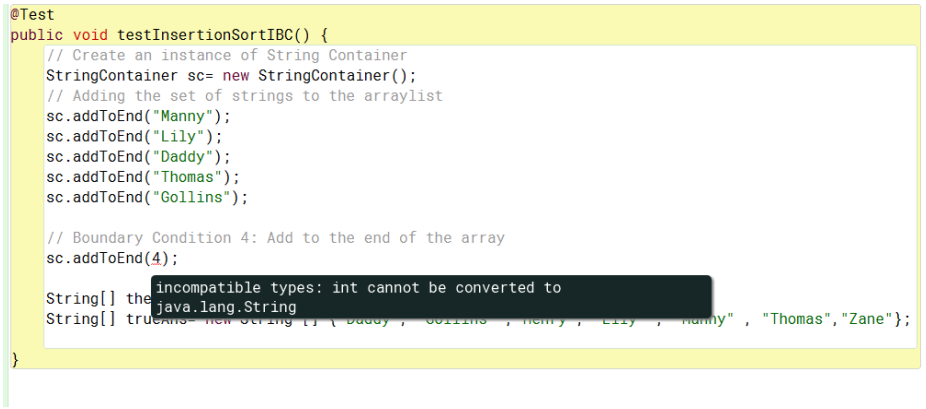

Figure 2 : Unit Test Result for IBC of insertionSort()

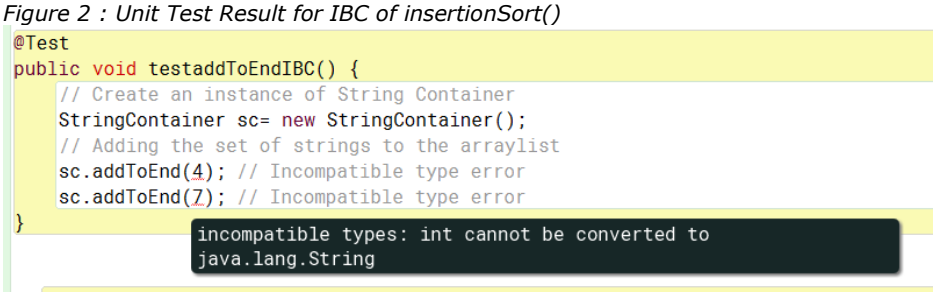

Figure 3 : Unit Test Result for IBC of addToEnd()

Figure 2 and 3 depict the error messages received while trying to test the Invalid Boundary Case for the insertionSort() and addToEnd. This was the same for bubbleSort.

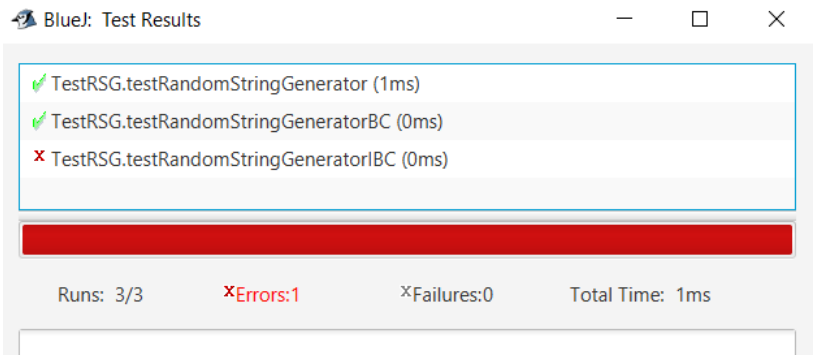

Figure 4: Unit Test Result For Random String Generator

The Random String Generator unit tests were essentially testing to see if the specified string length equaled the string length of the random word generated. The first unit test consisted of testing for a string length of 6, the second was 0, and the last was a negative integer value. The last test failed because it is not possible to define a string length with negative length.

Experiments conducted

In ExperimentController to test the run times of the eight methods defined in the class, the user had to the ability to choose the number of input values by typing

them in from the command line. For each number of input, five different trails are conducted and the run times are transcribed in an excel document with printWriter. When it came to the searching methods(linearSearchFound()… binarySearchFoun()…) the query value that I used was set to 3% of the number of inputs.
The numbers that I chose for the 'number of inputs' values were: 5000,7000,10000. These average of the five trails for each method is displayed graphically in the next section.

Results (analysis of the data)

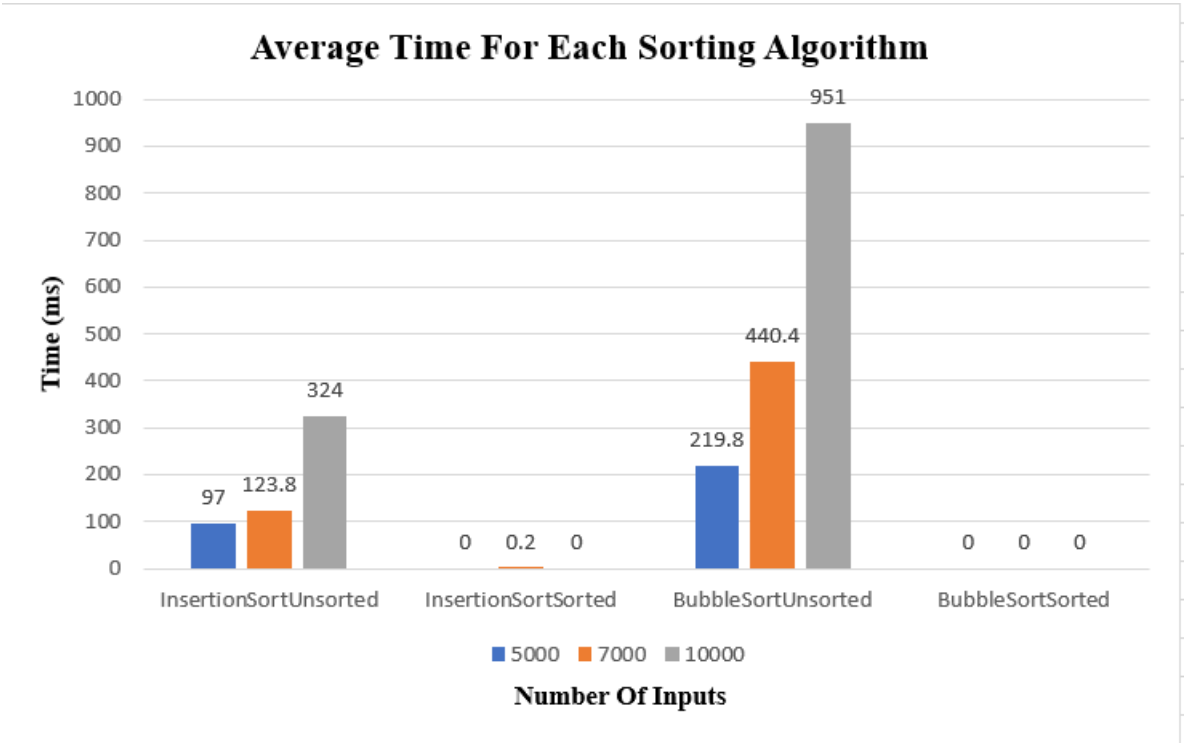

Figure 4 : Graph of the Average Time for each sorting algorithm method

Data Analysis:
As seen in Figure 4 , as the number of inputs increase the amount of time used for each sorting method increases. Specifically the insertionSortUnsorted() method has a runtime of 97 ms ,123.8 ms, 324 ms as the number of inputs increase from 5000 to 7000 and finally to 10000. The same trend is seen with the bubbleSortUnsorted() method with runtimes of 219.8 ms, 440.4, and 951 ms. InsertionSortofSorted() and BubbleSortSorted() both have runtimes that are zero or close to zero, which makes sense because it there is no sorting being done if the arraylist is already sorted, regardless of the number of inputs.

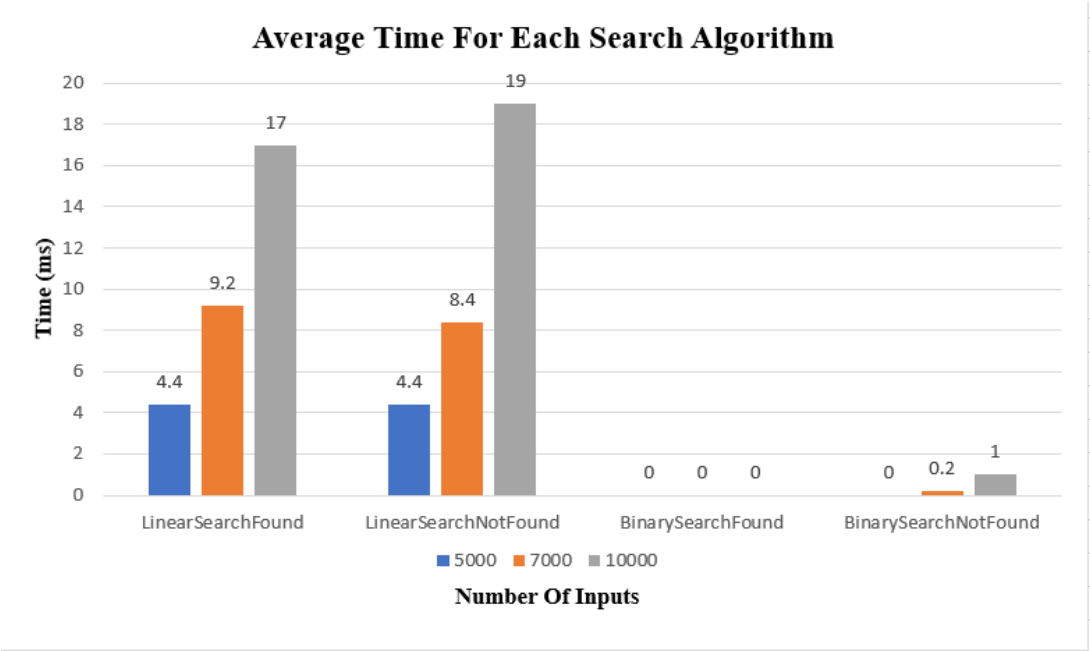

Figure 5 : Graph of the Average Time for each search algorithm method

Data Analysis:

As seen in Figure 5, as the number of inputs increase, run time for the first two methods (LinearSearchFound() & LinearSearchNotFound()) increase nearly identically . The linearSearchFound() method's run time goes from 4.4 ms, to 9.2ms, 17ms, While the linearSearchFound() runtimes are 4.4 ms, 8.4ms, and 19. The runtime for binarySearchFound() are zero for all the input values and it can be seen that runtimes of BinarySearchNotFound() are close to zero. This is strange because this essentially is saying that it takes no time to find a string match using the binarySearch method.

Screenshot Of Experiment Controller in Command Line :

```
C:\Users\irwin\Documents\FALL 2018\CS 150\Lab 3>java ExperimentController
Enter the name of your outpit file. Be sure to put .csv at the end test5.csv

 Your output file name is: test5.csv
 Enter your first number of items: 5000

 Enter your second number of items: 7000

 Enter your third number of items: 10000

 The number of items you entered are: 5000 7000 10000
 Please standby. Your Output file is being created
```

Figure 6 : Command Line Run of ExperimentController class

Conclusion

This lab was key in familiarizing one with the searching and sorting algorithms performed on array list, while taking heed to the runtime of each of these algorithms. Revisiting the hypothesis made in the introduction, insertionSortofUnsorted() and bubbleSortofUnsorted() did have a runtime of $O(n^2)$. After analyzing the datapoints of two methods at their respective input values, it can be said to produce a quadratic looking graph when plotted. (*See Figure 6 & 7 below.)* My hypothesized runtimes for the linearSearch() and linearSearchNotFound() proved to be true, although it can be seen that there is some slight disruption to the linearity of the graphs of their datapoints when plotted.

My hypothesized runtime for insertionSortSorted() & bubbleSortSorted() was disproved. According to the data, the increase of the number of inputs did not increase the time in which it took these two methods to run; they should hence have a O(1) run time based on the data. It is hard to conclude as to whether my hypothesis for the binarySearch methods were valid or invalid since my data points for binarySearch() were potentially skewed.

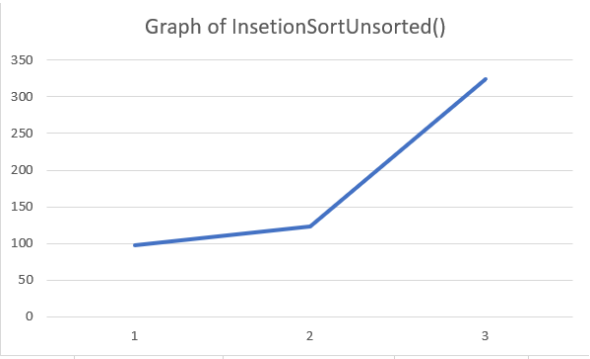

Figure 6 : Graph of InsertionSortUnsorted()

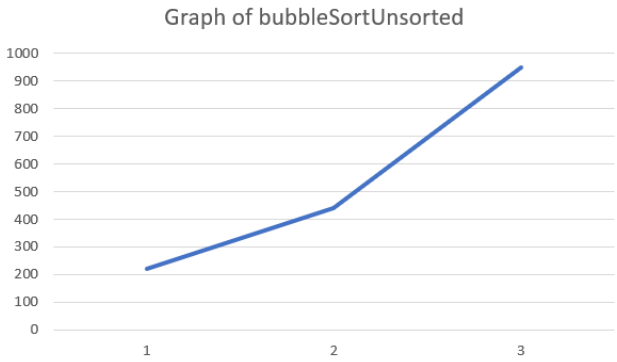

Figure 7 : Graph of bubbleSortUnsorted()

Trouble report

The binarySearchFound() and binarySearchNotFound() return a runtime of 0 for all the number of input values entered. This might be a result of having a small query search value (3% of number of inputs). To see if the query affected BinarySearch() runtime, I altered the query value to 50% of number of inputs. However, after compiling and running the program, I received an error shown in Figure 8 below.

```
Your output file name is: testingfinal.csv
Enter your first number of items: 5000

Enter your second number of items: 7000

Enter your third number of items: 10000

The number of items you entered are: 5000 7000 10000
Please standby. Your Output file is being createdException occured java.lang.IndexOutOfBoundsException: Index: 5000, Size: 5000
```

Figure 8 : Error Message Received After Query Value Change

References

- Weiss, Mark Allen. Data Structures & Problem Solving Using Java. Pearson Education, 2010. Chapter 8: Sorting Algorithms pg 354
- Frank Xia Lecture PowerPoint (Bubble Sort)
- T. (n.d.). Data Structures and Algorithms Binary Search. Retrieved from https://www.tutorialspoint.com/data_structures_algorithms/binary_search_algorithm.htm Binary Search Algorithms