CMPT-435-Assignment-2

Ahmed Handulle

October 11, 2022

In this document, I will be explaining my code from Assignment 2 in detail. It will be divided into different sections by Functions and talk about what each part will do.

1 External Java Packages

Below is a list of external Java packages that I have used to create my program.

```java
import java.io.File; // This line will import the Java file utility package and it will give us access to other files in our computer. We will also use it to read data from as well as writing data to those files.
import java.util.Scanner; // This line imports the Java Scanner package which is a text scanner that can parse data (strings/numbers) using regular expressions.
import java.util.ArrayList; // This line also imports the ArrayList package which will give us many functionalities of dynamic arrays in Java.
import java.util.Random;  // Importing this random class will help us create objects of the Random class to generate random numbers.
```

2 Class Level Variables

The following code listing describes the class level variables (global) for the Main class of this program. The two variables will be used to keep truck of the number of comparisons that happen inside the merge and quick sorts respectively. The will be accessed from inside of future methods.

```java
public class Main {
    private static int mergeSortComparisonsCount=0;
    private static int quickSortComparisonsCount=0;
```

3 Shuffling Method

The below functions takes an ArrayList as an input. The functions then shuffles all the elements in the ArrayList using a technique known as the Knuth Shuffle. The way this technique works is that it will start iterating over the list elements from the first position (i=0) to the last (n-1). At any given position, a random integer(index) between the (i)th position and the last position of the list will be generated. After that, the element at the position of the random index will be swapped with the element at the (i)th position. This technique eliminates all biases as all elements in the list have equal chances of being chosen as random.

```
public static void shuffleArrList(ArrayList<String> strList) {
    int n = strList.size();
    //Creating an object of the Random class in order to computer the random integer
    Random ran = new Random();
    //Iterating over the ArrayList
    for(int i=0; i<n; i++){
    //Generating a random integer; nextInt(int n) method is used to generate a random integer from the sequence of 0(inclusive) to the number passed as an argument.
        int randomIndex = i + ran.nextInt(n-i);
        //Then the values at the (i)th position is swapped with the one at the randomIndex.
        String temp = strList.get(randomIndex);
        strList.set(randomIndex, strList.get(i));
        strList.set(i, temp);
    }
}
```

4 Selection Sort

In the section, I will be implementing a selection sort algorithm. The function takes an ArrayList of strings as an input. This is comparison-based algorithm where the list of (n) elements will traversed and for each iteration, the minimum element of the list will be selected and swapped with the element at the (i)th position where (i) starts from 0-(n-1). The average and the worst case time complexities of this algorithm is of big O(n2) where (n) is the number of items.

```
public static ArrayList<String> selectionSort(ArrayList<String> str){
//We check if the input is empty before we start doing anything else so we can save time and computing power
    if (str == null) {
        return null;
    }
// Then we are making a call to the shuffling method to shaffle the list before we start sorting.
    shuffleArrList(str);
    int comparisonCount = 0;
    int n = str.size();
// This outer loop will start iterating from the first position of the list (i=0) up to (n-1) where n is the size of the list.
```

```
    for(int i=0; i<n-1; i++){
// We are now considering the element at the first position as the minimum element.
        int minPosition = i;
// The inner loop will start iterating from the element at the second position of the list up to the last element.
        for(int j= i+1; j<n; j++){
            comparisonCount++;
//For each (j)th iteration, we are comparing the element at the (j)th position with the element at our minimum positon. Since we are using an ArrayList, we use the obj.compareTo(obj) method to compare the two strings. This method will return a positive integer if the first (obj) is greater than the second (obj) alphabetically, otherwise it will return a negative integer. Also, this method will return 0 if both the strings are the same. Then we will save the outcome in an int variable called result.
            int result = str.get(j).compareTo(str.get(minPosition));
            if(result<0){
//If the element at the (j)th position is smaller than the element at the minimum position, we then update the minimum position to the (j)th position.
                minPosition = j;
            }
        }
// After the minimun element is found, then it will be swapped with the element at the (i)th position.
        String temp = str.get(i);
        str.set(i, str.get(minPosition));
        str.set(minPosition, temp);
    }
    System.out.println("Number of Comparisons: "+ comparisonCount);
    return str;
}
```

5 Insertion Sort

Insertion sort is another comparison-based algorithm which is slightly different than selection sort. For insertion sort, a sub-list is sustained which is always sorted. The sorted list first starts with one element which is usually the first element, and the rest of the list is considered to be unsorted. The elements in the unsorted list will be selected one by one and then inserted into their proper position in the sorted list, which usually start at the left end of the list by swapping with the next element. The following function takes an ArrayList as a list of strings.

```
public static ArrayList<String> insertionSort(ArrayList<String> str){
    if (str == null) {
        return null;
    }
    shuffleArrList(str);
//Here, I'm using two variables to count the number of comparisons.
```

```java
    int swapComparison = 0;
    int nonSwapComparison = 0;
    int n = str.size();
// This outer loop will iterate over all the elements in the list sequentially starting from the second position, since a sub-list of one element is already sorted, all the way up to the last element.
    for(int i=1; i<n; i++){
//Here we are keeping truck of both the position and the value of the current element we want to insert into its proper position, since we are swapping with other elements until we find its position.
        String currentValue = str.get(i);
        int currentPos = i;
// For this loop, we are checking two conditions. The first one is that we want to iterate over the all the elements in the list except the first one at indext 0, since we know that element is already in its sorted position. The second condition is where we compare the element at our current position with the element with its preceding element.
        while((currentPos>0) && (str.get(currentPos-1).compareTo(str.get(currentPos))>0)){
// We swap the elements if they satisfy both conditions and keep repeating this process until we find it is proper position or we reach the last position at index 0.
            str.set(currentPos, str.get(currentPos-1));
            str.set(currentPos-1, currentValue);
// Here we increment the number of comparisons when the condition in the while loop is true
            swapComparison++;
// We decrement the current position index since we are looking for the correct position in the sorted sub-list , which is at the left side of the list.
            currentPos--;
        }
// Here we also increment the number of comparisons in case the while is condition is false because we are doing comparisons in the while loop condition.
        nonSwapComparison++;
    }
    System.out.println("Number of Comparisons: "+ (swapComparison+nonSwapComparison));
    return str;
}
```

6 Merge Sort

Merge sort is a sorting algorithm that is based on divide and conquer technique with worst time complexity being big O(nlogn). In merge sort, the list is divided into halves each time until each sub-list becomes one single element. Then the sub-lists of single elements are combined in a sorted manner until we get one sorted list. Merger sort also uses recursion for both dividing and conquering the list. The following function (mergeSort) takes an ArrayList of strings as an argument.

```
//The following function is an extra step and somehow  unnecessary but I used it to make the program readable and easy to follow. So the function just takes an ArrayList of Strings as an input.
public static ArrayList<String> mergeSort(ArrayList<String> str) {
    if (str == null) {
        return null;
    }
    shuffleArrList(str);
// After shuffling the input, We are calling the actual merger sort and passing three arguments. The first one is the inputer (ArrayList of strings), the indext of the first element is the second argument and finally the index of the last element of our list.
    mergeSort(str, 0, str.size()-1);
    System.out.println("Number of Comparisons: "+ (mergeSortComparisonsCount));
    return str;
}
// This funcition is being called from the preceding function with arguments passed to it.
public static void mergeSort(ArrayList<String> str, int leftEndIndex, int rightEndIndex) {
// Checking to see if we have more than one element in our input list by comaring the indices of the first element element to the last index of the last element of the list.
    if(leftEndIndex<rightEndIndex){
// To divide the list in two halves, we have to find the mid point
        int mid = (leftEndIndex+rightEndIndex)/2;
// We are then calling the mergeSort recursively by passing three arguments, the input list, the leftmost index and the mid point index of the list. We continue calling the mergeSort until the left halve of the list is broken down into a sub list of single element.
        mergeSort(str, leftEndIndex, mid);
// The samething is done for the right half of the list.
        mergeSort(str, mid+1, rightEndIndex)
// Most of the work happens here by calling the merge sort function.
        merge(str, leftEndIndex, mid, rightEndIndex);
    }
}
// The Merge sort function combines the sub-lists into a single sub-list in a sorted manner. It takes an ArrayList of Strings as an input as well as leftmost, rightmost and mid indices.
private static void merge(ArrayList<String> str, int leftEndIndex, int mid, int rightEndIndex) {
// Here we create an empty ArrayList data structure for merging the sub-lists since we can't use our original list to merge the sub-lists because there are more than two sub-lists. We then copy the elements of the original list into the new temporary list because we can't access the indices of an empty list.
    ArrayList<String> tempArray = new ArrayList<String>(str);
    int i = leftEndIndex;
    int j = mid+1;
    int tempArrayIndex = leftEndIndex;
// Here, we are using a while loop to iterate over every two sub-lists by comparing the their values to each other and placing
```

```java
each value in the sub-list onto the temporary ArrayList we have created for storing sub-lists.
    while((i<= mid) && (j<=rightEndIndex)){
// We check if the left left sub-list at indext (i) is smaller than the one in the right.
        if(str.get(i).compareTo(str.get(j))<0){
// Since the condition is true, we copy the value at index (i) in the sub-list into the temporary ArrayList of strings.
            tempArray.set(tempArrayIndex, str.get(i));
// We then increment the index of the left sub-list
            i++;

        } else {
// Since the condition is false, it means the element at index (j) in the right sub-list is smaller and that element will be stored in the temporary ArrayList at index (j)
            tempArray.set(tempArrayIndex, str.get(j));
// We then increment the index of the right sub-list
            j++;
        }
// We are also incriminating the index in the temporary ArrayList since we place an element at the current index
        tempArrayIndex++;
        mergeSortComparisonsCount++;
    }
// In the following loop, we check if there any visited elements left in the left sub-list in case the right sub-list traversal finished before the left-sub-list.
    while (i<= mid){
// Since the condition is true, we finish iterating over the rest of the elements int he left sub-list and just put them in the temproray list since there are no element to compare with fromt he right sub-list.
        tempArray.set(tempArrayIndex, str.get(i));
        i++;
        tempArrayIndex++;
    }
    while (j<=rightEndIndex) {
// We do the same thing for the right sub-list in-case the left sub-list are visited before the right sub-list iteration is finished and just put them in the temporary list.
        tempArray.set(tempArrayIndex, str.get(j));
        j++;
        tempArrayIndex++;
    }
// Now, we are iterating over the temporary ArrayList, which stored the sorted list and copy element back into their original ArrayList.
    for(int x=leftEndIndex; x <rightEndIndex+1; x++){
        str.set(x, tempArray.get(x));
    }
}
```

7 quickSort Sort

Quick Sort is another sorting algorithm that is based on partitioning of list of data into smaller lists. At first step, a larger list will be partitioned into two sub-lists with one sub-list holding values smaller than a specific value, a pivot, and the other sub-list holding values greater than the pivot value. Then, the algorithm uses recursion to sort the two sub-lists recursively. Also, this algorithm takes big O(nlogn) on an average case but it can go down to big O(n2).

```java
// The following method takes an ArrayList of strings as an argument and I'm using it to make an easy call to the actual quick sort function.
public static ArrayList<String> quickSort(ArrayList<String> str) {
    if (str == null) {
        return null;
    }
    shuffleArrList(str);
// After shuffling the list, we make a call to our actual sort function and pass three arguments as the ArrayList of strings, position of the first element of the list, and the position of the last element of the Arraylist respectively.
    quickSort(str, 0, str.size()-1);
    System.out.println("Number of Comparisons: "+ (quickSortComparisonsCount));
    return str;
}
public static void quickSort(ArrayList<String> str, int leftEndIndex, int rightEndIndex) {
//Here, I'm making sure that the left most position of the list is always less than right most position, if not then we know that we have partitioned all the elements of the list.
    if (leftEndIndex<rightEndIndex){
// Once the above condition is true, we are going to divide the partition the larger Arraylist around a specific value. To do that we are making a call to the partition function which will return the position of the specific value that we partitioned around the elements.
        int partitionIndex = partition(str, leftEndIndex, rightEndIndex);
// Then, we keep partitioning the sub-lists until we get to sub-list of one element.
        quickSort(str, leftEndIndex, partitionIndex-1);
        quickSort(str, partitionIndex+1, rightEndIndex);
    }
}
public static int partition(ArrayList<String> str, int leftEndIndex, int rightEndIndex) {
// Since we are shuffling the list before we start sorting, then I have decided to pick the first element of the ArrayList as the pivot and store it in a string variable called pivot.
    String pivot = str.get(leftEndIndex);
    int i = leftEndIndex+1;
    int j = rightEndIndex;
// I'm using a while loop here that will always execute until we find the position of the partitioned element. Then we will
```

```java
    break out the loop once we find that.
    while(true){
// I'm also using another while loop to increment the the (i)th index which is on the left sub-list and will compare its values with the pivot. This loop will terminate if the value at the (i)th index is greater than pivot.
        while((i<=j) && ((str.get(i).compareTo(pivot))<=0)){
            i++;
            quickSortComparisonsCount++;
        }
// I'm also using another while loop to increment the (j)th index which is on the right sub-list and will compare its values with the pivot. This loop will terminate if the value at the (j)th index is less than pivot. Because, we want to keep all the value greater than the pivot on the right as well as keeping the ones less than the pivot on the left side.
        while((i<=j) && ((str.get(j).compareTo(pivot))>0)){
            j--;
            quickSortComparisonsCount++;
        }
// When both while loops terminate, we need to swap the element at the (i)th index with the one at (j)th index since they didn't satisfy the conditions in the above two while loops.
// Before we swap the elements, we have to check if there are still some elements that we didn't visit using this condition in the following if statement (i<=j)
        if(i<=j){
            String temp = str.get(i);
            str.set(i, str.get(j));
            str.set(j, temp);
        } else {
// When all elements are traversed, then we breaking out the outer loop, which also means that we have found the position of the portioning element
            break;
        }
    }
// Then we swap the element at the position of the portioned element with the first element in the ArrayList which we chose as our pivot.
    String tempVar2 = str.get(leftEndIndex);
    str.set(leftEndIndex, str.get(j));
    str.set(j, tempVar2);
// we just then return the position of the partitioned element
    return j;
}
```

8 Results

The following table shows the number of comparisons and their asymptotic time complexities.

Selection Sort O(n2):
Number of Comparisons: 221445
Number of Comparisons (with Shuffling): 221445
Insertion Sort O(n2):
Number of Comparisons: 114925
Number of Comparisons (with Shuffling): 109656
Merge Sort O(nlog):
Number of Comparisons: 5444
Number of Comparisons (with Shuffling): 5407
Quick Sort O(nlogn)–O(n2):
Number of Comparisons: 6388
Number of Comparisons (with Shuffling): 7686