Assignment Two - Sorting Magic Items

Charlie Schmitz

September 30th 2020

---

1 Asymptotic Running Time Table

	Selection Sort	Insertion Sort	Merge Sort	Quick Sort
Best Case (Swaps) :	O(1)	O(1)	O(n log n)	O(n log n)
Worst Case (Swaps) :	O(n)	$O(n^2)$	O(n log n)	$O(n^2)$
Best Case (Comparisons) :	$O(n^2)$	O(n)	O(n log n)	O(n log n)
Worst Case (Comparisons) :	$O(n^2)$	$O(n^2)$	O(n log n)	$O(n^2)$
Average Performance (Swaps) :	$O(n^2)$	$O(n^2)$	O(n log n)	O(n log n)
Average Performance (Comparisons) :	O(n)	$O(n^2)$	O(n log n)	O(n log n)

These case running time values can vary based on how the code is implemented, such as methods of partitions for merge/quick sorts.

---

2 Run Time Explanations/Analysis

Selection Sort is derived into three parts:
1. The running time for all the calls of the minimum index
2. The running time for all the swaps
3. The running time for the rest of the loop in the selection sort function.
1 and 2 are called n times and is consistent, however step 3 so that takes constant time n (for 1 and 2) iterations multiplied by $O(n)$ iterations for step 3.

Insertions Sort on average takes $O(n^2)$ time, but inserting can cause every element in the array to slide over one, or each element to not slide at all. Insertion causes each element to slide over if the "key" being inserted is less than every element to the left. This means that the worst case would be $O(n^2)$ assuming each element has to slide. At its best every element is nearly or already sorted and no elements have to be moved. The average still remains $O(n^2)$ because here constant coefficients do not matter even if a random sorted array finishes somewhere in-between times. $O(1)$ represents the constant time if all items are sorted already.

Merge Sort runs $O(n)$ times when merging n elements and runs in $O(nlogn)$ time. This can also be broken down into 3 parts:
1. The divide step takes a constant n time regardless of size of the array. This is indicated with $O(1)$ which is used to represent a constant time.
2. This is the stage in which recursion happens, in which the array is "divided and conquered" in which there are $n/2$ elements that should be sorted.
3. Combining/Merge step in which a n items are sorted in $O(n)$ time. Dividing and conquering can be depicted with cn or $logn$ times. This added all together averages out the run time to $O(n) * O(logn) = O(nlogn)$.

Quick sort can depend on the efficiency of the partitions. The worst case means the partitions are unbalanced and take either a constant cn time, each recursive call takes $c(n-$element number within array). Through 1 + 2 + 3 + ... + n number of iterations. The worst case run time would be $O(n^2)$. The best case occurs when partitions are as balanced as possible meaning there sizes are about equal. This places the pivot point right in the middle splitting the array evenly. This gives a run time of $O(nlogn)$, the same as merge sort. The constant factors here are what enable the average run time to also be $O(nlogn)$ as the coefficients can be ignored, the time taken to run depends on the partition and how well sorted the array before being quick sorted.

3 Code

```java
import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.util.Arrays;

public class AssignmentTwoMain {

    private static int size = 0;
    private static int selectionCounter = 0;
    private static int insertionCounter = 0;
    private static int mergeCounter = 0;
    private static int quickCounter = 0;


    public static void main(String[] args) {

        //Use magicitems.get(index) to get values in ArrayList magicItems

        /* Goals:
         * Sort using your  selection sort. Print the number  of comparisons.
         * Sort using your  insertion sort. Print the number  of comparisons.
         * Sort using your  merge sort. Print the number  of comparisons.
         * Sort using your  quick sort. Print the number  of comparisons.
         */
        //read in the array list each time so that it can be resorted and counted
        ArrayList<String> magicitems = readFromFile("magicitems.txt");
        System.out.println(Arrays.toString(magicitems.toArray()));
        System.out.println("\n");

        selectionSort(magicitems);

        ArrayList<String> magicitems1 = readFromFile("magicitems.txt");
        insertionSort(magicitems1);

        ArrayList<String> magicitems2 = readFromFile("magicitems.txt");
        mergeSort(magicitems2);
        //Merge sort done in different way than insertion and selection since utilizes 2 functions
        // to divide and conquer so prints comparisons and list out here to show sorted
        System.out.println("Merge Sort : Comparisons = " +
```

```java
mergeCounter);
        System.out.println(Arrays.toString(magicitems2.toArray()));

        //Quick sort also done in differnet way than insertion and selection because quicksort function called multiple times
        //similarly to that of merge sort so printing done in main
        ArrayList<String> magicitems3 = readFromFile("magicitems.txt");
        int n = magicitems3.size();
        quickSort(magicitems3,0, n-1);
        System.out.println("Quick Sort : Comparisons = " + quickCounter);
        System.out.println(Arrays.toString(magicitems3.toArray()));


    }

    //this is the function that takes in the file name and adds it to an arrayList
    public static ArrayList readFromFile(String filename) {
        ArrayList<String> myList = new ArrayList<String>();
        try {
            Scanner sc = new Scanner(new File(filename));
            while(sc.hasNextLine()) {
                String value = sc.nextLine();
                myList.add(value.substring(0,1).toUpperCase() + value.substring(1));
                size++;
            }
            sc.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }

        return myList;

    }//readFromFile

    public static void selectionSort(ArrayList<String> magicitems) {

        //check to see if the array list is null, if it is return null value
        //OR if it empty, OR if only one item in the array
        if(magicitems == null || magicitems.size()== 1 || magicitems.size()== 0) {
            return;
        }//if

        //loop through, compare, and sort
        for(int headIndex = 0; headIndex < magicitems.size(); headIndex++) {
            int smallestIndex = findSmallestFrom(headIndex, magicitems);
            if (smallestIndex != headIndex) {
                String head = magicitems.get(headIndex);
                magicitems.set(headIndex, magicitems.get(
```

```
    smallestIndex));
                magicitems.set(smallestIndex, head);
                selectionCounter++;
            }
        }

        System.out.println("Selection Sort : Comparisons = " +
    selectionCounter);
        System.out.println(Arrays.toString(magicitems.toArray()));
    }//selectionSort

    private static int findSmallestFrom(int i, ArrayList<String>
    magicitems ) {
        int smallestIndex = i;
        String smallest = magicitems.get(i);
        for (int j = i; j < magicitems.size(); j++) {
            String value = magicitems.get(j);
            if (value.compareToIgnoreCase(smallest) < 0) {
                smallest = value;
                smallestIndex = j;
                selectionCounter++;
            }
        }
        return smallestIndex;
    }

    public static void insertionSort(ArrayList<String> magicitems)
    {

        //list sorted one items at a time, not very efficient
    compared to merge and quicksort
        //loops through the size of the array, within that another
    while loop is used to compare the element to the other elements
        //and sort into proper location
        for(int i = 1; i < magicitems.size(); i++) {
            String value1 = magicitems.get(i);

            int j = i-1;
            while(j >= 0 && value1.compareToIgnoreCase(magicitems.
    get(j)) < 0) {
                magicitems.set(j+1, magicitems.get(j));
                j--;
                insertionCounter++;
            }//for
            magicitems.set(j+1, value1);

        }//for


        System.out.println("Insertion Sort : Comparisons = " +
    insertionCounter);
        System.out.println(Arrays.toString(magicitems.toArray()));
    }//insertionSort



    public static ArrayList<String> mergeSort(ArrayList<String>
```

```java
magicitems) {
        int center;
        ArrayList<String> left = new ArrayList<String>();
        ArrayList<String> right = new ArrayList<String>();

        if (magicitems.size() <= 1) {
            return magicitems;
        }//if list only one item in array

        else {
            //take the center of the array and break down into left and right, sort left and right halves and continue
            //divide and conquer then merge together. Merge Function is used for actual merges of the right, left, and full arrays.
            //It compares and makes sure that all is merged in proper order checking for remaining elements and those that have been used up
            //as well as there index's

            center = magicitems.size()/2;

            //add items to each sections
            for (int i = 0; i < center; i++) {
                left.add(magicitems.get(i));
            }
            for (int i = center; i < magicitems.size(); i++) {
                right.add(magicitems.get(i));
            }

            //mergeSort the sections themselves
            left =  mergeSort(left);
            right = mergeSort(right);

            //merge together the sorted sections
            merge(left, right, magicitems);

        }//else begin sorting array

        return magicitems;
    }//mergeSort

    public static void merge (ArrayList<String> left, ArrayList<String> right, ArrayList<String> magicitems){
        int leftIndex = 0;
        int rightIndex = 0;
        int magicItemsIndex = 0;

        //As long as the left and right elements of the array remain or have not been "used up"
        while (leftIndex < left.size() && rightIndex < right.size()) {
            if ( (left.get(leftIndex).compareToIgnoreCase(right.get(rightIndex))) < 0) {
                magicitems.set(magicItemsIndex, left.get(leftIndex));
                leftIndex++;
```

```java
            }
            else {
                magicitems.set(magicItemsIndex, right.get(rightIndex));
                rightIndex++;
            }
            magicItemsIndex++;
            mergeCounter++;
        }
        ArrayList<String> remain;
        int remainIndex;
        //if the left array has been fully used up remaining is the right, else the remaining is the left
        if (leftIndex >= left.size()) {
            remain = right;
            remainIndex = rightIndex;
        } //if
        else {
            remain = left;
            remainIndex = leftIndex;
        }//else

        // Copy the remaining array items that have not been fully used up
        for (int i = remainIndex; i < remain.size(); i++) {
            magicitems.set(magicItemsIndex, remain.get(i));
            magicItemsIndex++;
        }
    }//merge

    public static void quickSort(ArrayList<String> magicitems, int low, int high) {

        //low/high and smallerthan/greaterthan are same variables being passed around just named differently
        //low is the starting index, high is the ending index. Each is either smallerthan or greaterthan the pivot point
        if(low < high){
            //part is the partition index
            int part = partition(magicitems, low, high);

            //Sort the elements before and after the partition
            quickSort(magicitems, low, (part - 1)); //before
            quickSort(magicitems, (part+1), high); //after
        }


    }//quickSort

    public static int partition(ArrayList<String> magicitems, int smallerThan, int greaterThan){
        String pivot = magicitems.get(greaterThan);
        int i = (smallerThan-1); //represents the index of the smaller element
        for(int j = smallerThan; j < greaterThan; j++){
            //if the current element is smaller than the pivot, performs a swap
```

```
            if(magicitems.get(j).compareToIgnoreCase(pivot) < 0){
                i++;

                String temp = magicitems.get(i);
                magicitems.set(i,magicitems.get(j));
                magicitems.set(j, temp);

                quickCounter++;
            }
        }
    //swaps magicitems[i+1] and magicitems[greaterThan](or pivot)
        String temp = magicitems.get(i+1);
        magicitems.set(i+1, magicitems.get(greaterThan));
        magicitems.set(greaterThan, temp);

        return i+1;
    }
    //partition - this function takes the last element as a pivot point and places the pivot element
    //as the correct position in the final sorted magicitems array.
    //It then places all elements smaller than the pivot to the left and all greater than to the right



}//AssignmentTwoMainClass
```