Lab 1

Steve Boucher

October 8, 2022

I'm not that sort!

1 Introduction

The Goal of Lab 2 was to write four algorithms that would take a list of magic items from a text file and a sort them alphabetically. The algorithms required for this lab were; insertion sort, selection sort, merge sort, and quick sort. Once sorted, the code outputs the amount of comparisons required to have completed the sort, as well as the time to sort.

2 The Code

I was thankfully able to pull my file reader from my previous assignment and get to work. To successfully demonstrate sorting a list, its important that the list be adequately shuffled with the shuffle command which runs before each sort.

```
public static void Shuffle(String[] magicalArray) throws FileNotFoundException {
  String temp[];
  temp = new String[2];
  temp[1] = null;
  for (int i = 0; i < magicalArray.length; i++) {
    int random_int = (int) Math.floor(Math.random() * (magicalArray.length));
    temp[1] = magicalArray[random_int];
    magicalArray[random_int] = magicalArray[i];
    magicalArray[i] = temp[1];
  }
}
```

I was wonderfully please at how simple it was to code this shuffle method. It works by going through each element in the list, and randomly swapping it with another element. This command is also quite simple to the computer to. It has a complexity of O(n) as It only has to run through the data set of magicalArray once.

2.1 Selection And Insertion Sort

I had a surprisingly hard time sorting using selection and Insertion sort. My main difficulty came down to the fact we were sorting strings alphabetically as apposed to numbers numerically. I wasn't able to easily wrap my head around the compareTo() function which gave me difficulties in writing functions like this one found on line 94.

```
94 int check = compareOne.compareToIgnoreCase(compareTwo);
```

After looking up a handful of guides, explaining it to my roommates several times, and writing it on the mirror that resides on my closet, I got the concept of it and was able to march on. I did struggle with some of these problems the most oddly enough, as I thought I had completed Insertion sort first, but on double checking at the end, needed to be reworked.Thankfully which cut down on the run time and increased accuracy significantly. Previously, my run time for my first iteration of Insertion sort was over complicated with 3 massive for loops. Giving it an astonishing run time of O(n$^{3"}$)$, onsimplification, IcutthatdowntoO(n(n-1)/2)withselectionsorttakingasimilaramountoftime.$

2.1.1 Merge Sort And Quick Sort

Merge sort and Quick sort took a lot less time to program than I thought they would. Each of these recursive arrays pack a punch of computing using the conquer and divide method. For merge sort, I utilize a command called Merge Sort which is called on line 117. 117 splits my magical array in half again and again until I get back arrays in groups of 2 to 3. I put them back together on line 138 where I invoke the merge function, which stitches them back together in the magical array. to achieve a complexity of O(n*Log(n))

```
136    MergeSort(leftHand, count);
137    MergeSort(rightHand, count);
138    merge(magicalArray, leftHand, rightHand, count);
```

QuickSort splits the array, not into smaller groups, but into two half. My quickSort takes a random element form the array to set as its guide to part the array.

```
int pivotIndex = new Random().nextInt(highIndex - lowIndex) + lowIndex;
```

this partitions the array on either side and counts through the values on the left side of the array. If its smaller, it gets to stay on the left, if the number is larger it gets swapped to the right of the pivot index, and is further sorted.The complexity to Quick Sort is O(n*logn) as its use of recursion and the divide and conquer allows it to not repeat as much data going through loops.

2.1.2 Count

One of the goals of this lab was to print the number of comparisons made in each type of sort. I was able to complete this in a sensible manner for Selection and Insertion sort with a countput command. Count is a variable that counts up for each comparison made.

```
  public static void countput(int count) {

    System.out.println("Number of comparisons: " + count);
```

2.2 Time

Finding the time for each function was done by finding the time right before a sort function is called, and right after. Once these to values were found. I calculated the different and printed it out.

```
System.out.println("Merge Sort");
long timeB = System.currentTimeMillis();
MergeSort(magicalArray, count);
long timeA = System.currentTimeMillis();
System.out.println("This took : " + (timeA - timeB) + " milliseconds ");
Shuffle(magicalArray);
System.out.println("Quick Sort");
timeB = System.currentTimeMillis();
QuickSort(magicalArray, 1, magicalArray.length - 1, count);
timeA = System.currentTimeMillis();
System.out.println("This took : " + (timeA - timeB) + " milliseconds ");
Shuffle(magicalArray);
timeB = System.currentTimeMillis();
SelectionSort(magicalArray, count);
timeA = System.currentTimeMillis();
System.out.println("This took : " + (timeA - timeB) + " milliseconds ");
Shuffle(magicalArray);
timeB = System.currentTimeMillis();
InsertionSort(magicalArray, count);
timeA = System.currentTimeMillis();
System.out.println("This took : " + (timeA - timeB) + " milliseconds ");
```

Due to the recursive natures of Quick and Merge Sort. I was unable to figure out how to successfully count comparisons as each run through its own command would reset the value of count. I currently have my code output the comparisons made, which out puts a large quantity of different integers which could be totaled up to the amount of comparisons, I just couldn't figure out how.

2.3 Results

Sort	Comparisons	Time
Selection Sort	221445	6 milliseconds
Insertion Sort	108613	4 milliseconds
Merge Sort	?	75 milliseconds
Quick Sort	?	91 milliseconds

3 Conclusion

I am proud of myself for being able to get through and code through all the different sort functions.It was a challenge, but very satisfying seeing the correctly ordered list of items.