Assignment 2 - Code Evaluation

Mason Nakamura

October 2021

1 Shuffle Class

```
public class shuffle {
    public String[] shuffle(String[] A){
        int n = A.length;
        for(int i = 0; i < n; i++){
            // pick random number from 0 to i uniformly
            int rand = (int)(Math.random() * (i+1));
            String temp = A[rand];
            A[rand] = A[i];
            A[i] = temp;
        }
        return A;
    }
}
```

The method shuffle() from Shuffle Class shuffles an array based on a uniform distribution.

2 QuickSort Class

```
public class QuickSort {
    int comparisons = 0;
    // set private so not accessible outside of class
    private int partition(String[] A, int start, int end){
        String pivot = A[end];
        int i = start - 1;
        for (int j = start; j < end; j++){
            comparisons += 1;
            if(A[j].compareTo(pivot) < 0){
                i += 1;
                // swap A[i] and A[j]
                String temp1 = A[i];
                A[i] = A[j];
                A[j] = temp1;
            }
        }
        // swap A[i+1] and A[end]
        String temp2 = A[i+1];
        A[i+1] = A[end];
20            A[end] = temp2;
21            return i+1;
22        }
23
24        public void quickSort(String[] A, int start, int end){
25            if(start < end){
26                int q = partition(A, start, end);
27                quickSort(A, start, q-1);
28                quickSort(A, q +1, end);
29            }
30        }
31    }
```

The partition() method partitions the array using a pivot point at line 5 for reference. We then start comparing the pivot value with the elements in the subarray of A and replace them. At the end, line 21, we return the new index for the next recursive partition.

In the quickSort() method, we recursively go through the different partitions based on the pivot value previously defined in the partition() method, sorting all the elements in A.

$O(n \cdot log_2(n)) \rightarrow$ This is the average complexity, as it can reach n^2 if the pivot points are very bad.

3 MergeSort Class

```
1     public class MergeSort {
2         int comparisons = 0;
3         public void merge(String[] A, String[] L, String[] R, int l,
              int r) {
4             int i = 0;
5             int j = 0;
6             int k = 0;
7             while (i < l && j < r) {
8                 comparisons +=1;
9                 //switch the elements in A when comparing L and R
10                if (L[i].compareTo(R[j]) < 0) {
11                    A[k++] = L[i++];
12                }else {
13                    A[k++] = R[j++];
14                }
15            }
16             //these take into account the leftovers
17            while (i < l) {
18                comparisons +=1;
19               A[k++] = L[i++];
20            }
21            while (j < r) {
22                comparisons +=1;
23               A[k++] = R[j++];
24            }
25        }
26        public void mergeSort(String[] A, int end) {
27            if (end < 2) {
```

```
28                 return;
29             }
30             int mid = end / 2;
31             String[] L = new String[mid];
32             String[] R = new String[end - mid];
33 
34             for (int i = 0; i < mid; i++) {
35                 comparisons += 1;
36                 L[i] = A[i];
37             }
38             for (int i = mid; i < end; i++) {
39                 comparisons += 1;
40                 R[i - mid] = A[i];
41             }
42             mergeSort(L, mid);
43             mergeSort(R, end - mid);
44 
45             merge(A, L, R, mid, end - mid);
46         }
47 }
```

Similar to quicksort, we partition the array A into two subarrays L and R. We then compare these two subarrays, switching the elements in A. And after the first while loop, line 7, the proceeding while loops the remaining elements int eh subarray that need to be swapped in A. The mergeSort() returns if end ¡ 2 since it would already be sorted, acting as our base case, line 28. We define out middle index, line 31, then initiate our Left and Right subarrays. We then create the Left and Right subarrays based on A. Then we recursively go through the mergeSort() methods and merge().
$O(n \cdot log_2(n)) \rightarrow$ The $log_2(n)$ derives from the dividing portion of the algorithm while the n portion derives from the conquering.

4 InsertionSort Class

```
public class InserstionSort {
    int comparisons = 0;
    public String[] insertionSort(String[] A){
        int n = A.length;
        for(int i = 1; i < n; i++){
            comparisons += 1;
            String key = A[i];
            int j = i-1;
            // if A[j] does not need to be switched, skip while
            while(j >= 0 && A[j].compareTo(key) > 0){
                comparisons += 1;
                A[j+1] = A[j];
                j = j-1;
            }
            A[j+1] = key;
        }
        return A;
    }
}
```

We initiate the key in line 7, then if the key does not need to be switched with the current indexed array, we skip the while loop. If we enter the while loop in line 11, it shifts the index of the values in A. After the while loop, line 16, we re-initiate the key as a different index, and continue. We return A at the end, sorted.
$O(n) \rightarrow$ Since the while loop helps with not iterating through the entire array, it can be less that n^2, and usually is. However, if the array were perfectly inverted/reversed, we get n^2.

5 SelectionSort Class

```
1  public class SelectionSort {
2      int comparisons = 0;
3      public String[] selectionSort(String[] A){
4          int n = A.length;
5          for(int i=0; i < n-1; i++){
6              comparisons += 1;
7              int smallpos = i;
8              for(int j=i+1; j < n; j++){
9                  comparisons += 1;
10                 // see if A[j] is greater than A[smallpos]
                       Alphabetically
11                 if(A[j].compareTo(A[smallpos]) < 0){
12                     // if so, switch
13                     smallpos = j;
14                 }
15             }
16             //replace A[i] with the smallest value
17             String temp = A[i];
18             A[i] = A[smallpos];
19             A[smallpos] = temp;
20         }
21         return A;
22     }
23
24 }
```

We go through the entire A, line 5, and initiate a smallpos index variable for every i. Then we initiate another for loop, line 8, and compare every element up to index i to see if A[i] ¿ A[smallpos]. If true, we swap A[smallpos] and A[i].
$O(n^2) \rightarrow$ With the two embedded for loops looping through A.length, we get n^2.

6 Test Class

```
1  import javax.swing.plaf.synth.SynthUI;
2  import java.io.*;
3  import java.util.Arrays;
4
5  public class Test {
6
```

```java
    public static void main(String[] args) {
        String[] words = {};
        // Read line by line the txt file using File reader
        String fileName = "Assignment 1/magicitems";
        File file = new File(fileName);
        try {
            FileReader fr = new FileReader(file);
            BufferedReader br = new BufferedReader(fr);
            String line;
            while ((line = br.readLine()) != null) {
                // add strings from txt file line by line into array words
                words = Arrays.copyOf(words, words.length + 1); // extends memory
                words[words.length - 1] = line; //adds word to extra memory
            }
        } catch (FileNotFoundException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        // Start of testing for sorting
        // Testing of selection sort algorithm
        Shuffle shuffler = new Shuffle();
        String[] copy_words0 = words.clone();
        //shuffle the words
        copy_words0 = shuffler.shuffle(copy_words0);
        SelectionSort s_sort = new SelectionSort();
        s_sort.selectionSort(copy_words0);
//        for (String i : s_sort.selectionSort(copy_words0)) {
//            System.out.println(i);
//        }
        System.out.println("Number of Comparisons for Selection Sort: " + s_sort.comparisons);

        // Testing of insertion sort algorithm
        String[] copy_words1 = words.clone();
        //shuffle the words
        copy_words1 = shuffler.shuffle(copy_words1);
        InserstionSort i_sort = new InserstionSort();
        i_sort.insertionSort(copy_words1);
//        for (String i : i_sort.insertionSort(copy_words1)) {
//            System.out.println(i);
//        }
        System.out.println("Number of Comparisons for Insertion Sort: " + i_sort.comparisons);

        // Testing of merge sort algorithm
        String[] copy_words2 = words.clone();
        //shuffle the words
        copy_words2 = shuffler.shuffle(copy_words2);
        MergeSort m_sort = new MergeSort();
        m_sort.mergeSort(copy_words2, copy_words2.length);
//        for (String i : copy_words2) {
//            System.out.println(i);
//          }
        System.out.println("Number of Comparisons for Merge Sort: "
            + m_sort.comparisons);

        // Testing of quick sort algorithm
        String[] copy_words3 = words.clone();
        //shuffle the words
        copy_words3 = shuffler.shuffle(copy_words3);
        QuickSort q_sort = new QuickSort();
        q_sort.quickSort(copy_words3, 1, copy_words3.length-1);
//          for (String i : copy_words3) {
//              System.out.println(i);
//          }
        System.out.println("Number of Comparisons for Quick Sort: "
            + q_sort.comparisons);

//      for (String i : (words)) {
//              System.out.println(i);
//          }

    }
}
```

At every instance of our testing for the sorting algorithms, we shuffle a clone() of the words array derived from the file reader. We then apply the sorting algorithms to these shuffled arrays and output the number of comparisons.

7 Output

	Insertion Sort	Selection Sort	Quick Sort	Merge Sort
Number of Comparisons	≈ 110767	222110	≈ 6576	12604