CPSC 319 Assignment 2

Youssef Abdel Maksoud
30062891

March 05 2020

Question 1

Justification of sorting algorithms used:

Part A:

To sort the list of input words, a complex sorting algorithm, efficient with large input Merge sort seemed to be a better choice than quicksort for several reasons. merge sort has O(nlogn) runtime, it is easier to implement while quicksort has a worst case runtime of $O(n^2)$. Overall merge sort is difficult to write, but effective with large inputs. It is not favourable for small inputs

Part B

To sort each word and check for Anagrams, a simple sort was required. A sort effective for small inputs ($n < 100$). Selection sort is effective for small inputs, relatively easy to implement, but is less effective for large inputs. selection sort has a runtime of $O(n^2)$, making it more effective than mergesort for smaller inputs.

Question 2 : Program Big-On runtime

To analyze the program, we will look at method main() and analyze calls made within the main() method.

Algorithm / method

```
7   public static void main(String args[]){
8
9       SortingProg ass2 = new SortingProg();
10      Sorter sorter = new Sorter();
11      CheckAndOrganize check = new CheckAndOrganize();
12
13      String wordArr[];
14
15      wordArr = sorter.mergeSort(ass2.prompt());
16
17      LinkedList[] sortedWords = check.organizer(wordArr);
18
19      ass2.printOrg(sortedWords);
20
21
22
23  }
```

```
54  @   public String[] sArrEnlarge(String arr[]){
55          String newArr[] = new String[(arr.length + 1)];
56
57          for(int i = 0; i < arr.length; i++){
58              newArr[i] = arr[i];
59          }
60
61          return newArr;
62      }
```

```
25      public String[] prompt(){
26
27          Scanner scan = new Scanner(System.in);
28          String wordArr[] = new String[1];
29          int wordCount = 0;
30
31          System.out.println("Please enter a list of words type done when list is finished ");
32
33          while(scan.hasNextLine()) {
34
35              String input = scan.nextLine();
36
37              if (input.compareTo("done") == 0){
38                  System.out.println("Thank you for your input");
39                  break;
40              }
41
42              if(wordCount ==  length){
43                  wordArr = sArrEnlarge(wordArr);
44              }
45
46              wordArr[wordCount] = input.trim();
47              wordCount++;
48
49          }
50
51          return wordArr;
52      }
```

$T(n) = n^2 + 8n + 7 \rightarrow O(n^2)$

LINE	COST	TIMES
9	1	1
10	1	1
11	1	1
13	1	1
15	nlogn	1
17	$n^2 L^2$	1
19	n	1
55	2	1
57	3	n+1
58	1	n
61	1	1

$T(n) = 4n + 6 \rightarrow O(n)$

27	2	1
28	1	1
29	1	1
31	1	1
33	1	n+1
35	1	n
37	1	n
38	1	n
39	1	n
42	1	n
43	O(n)	n
46	1	n
47	1	n
51	1	1

```java
public String[] mergeSort(String wordArray[]){

    if(wordArray.length == 1) {
        return wordArray;
    }

    String[] left = new String[wordArray.length/2];
    String right[];

    if(wordArray.length % 2 == 0){
        right = new String[wordArray.length/2];
    }
    else{
        right = new String[(wordArray.length)/2 + 1];
    }
    //if even size creates two equal halves else right side is one larger

    for(int i = 0 ; i < left.length; i ++) {
        left[i] = wordArray[i];
    }

    for(int j = 0; j < right.length; j ++){
        right[j] = wordArray[left.length + j];
    }
    String result[];

    left = mergeSort(left);
    right = mergeSort(right);

    result = merge(left,right);

    return result;
}
```

```java
public String[] merge(String left[], String right[]){
    String merged[] = new String[left.length + right.length];

    int leftHolder = 0;
    int rightHolder = 0;
    int mergedIndex = 0;

    while(leftHolder < left.length || rightHolder < right.length) {

        if (leftHolder < left.length && rightHolder < right.length) {

            if (left[leftHolder].compareTo(right[rightHolder]) > 0) {
                merged[mergedIndex] = right[rightHolder];
                mergedIndex++;
                rightHolder++;
            }
            else if (left[leftHolder].compareTo(right[rightHolder]) < 0) {
                merged[mergedIndex] = left[leftHolder];
                mergedIndex++;
                leftHolder++;
            }
            else {
                merged[mergedIndex] = left[leftHolder];
                merged[mergedIndex + 1] = right[rightHolder];
                mergedIndex += 2;
                rightHolder++;
                leftHolder++;
            }
        }
        else if(leftHolder < left.length){
            merged[mergedIndex] = left[leftHolder];
            mergedIndex++;
            leftHolder++;
        }
        else if(rightHolder < right.length){
            merged[mergedIndex] = right[rightHolder];
            mergedIndex++;
            rightHolder++;
        }
    }
    return merged;
}
```

Constants = 13(1)

loops = 2

mergesort() x 2 $(\frac{n}{2})$

merge x 1 (O(n))

$$T(n) = \begin{cases} 1 & n=1 \\ 2T(\frac{n}{2}) + n + 13 \end{cases}$$

$$T(n) = 2T(\frac{n}{2}) + n$$

$$T(\frac{n}{2}) = 2T(\frac{n}{4}) + \frac{n}{2}$$

$$T(n) = 2\left(2T(\frac{n}{4}) + \frac{n}{2}\right) + n$$

$$T(\frac{n}{4}) = 2T(\frac{n}{8}) + \frac{n}{4}$$

$$T(n) = 4\left(2T(\frac{n}{8}) + \frac{n}{4}\right) + 2n$$

$$T(n) = 2^k \cdot 2T(\frac{n}{2^k}) + kn$$

$$\frac{n}{2^k} = 1 \rightarrow n = 2^k$$

$$k = \log n$$

plugging in:

$$T(n) = 2n + n\log n \rightarrow O(n\log n)$$

23 linear, 5 constants

$$T(n) = 23n + 5 \rightarrow O(n)$$

```java
public LinkedList[] organizer(String arr[]){
    Sorter sort = new Sorter();
    int i = 0;
    int j = 0;
    int elems = 0;

    LinkedList list[] = new LinkedList[arr.length];

    for(i = 0; i < arr.length; i ++){
        for(j = 0; j < elems + 1; j++){
            if(list[j] == null){
                list[j] = new LinkedList();
                list[j].addNode(arr[i]);
                elems ++;
                break;
            }
            else if (isAnagram(arr[i], list[j].getHeadM().getWord()) == true){
                list[j].addNode(arr[i]);
                break;
            }
        }
    }
    return list;
}
```

used selection sort twice so maybe with changed code after

LINE	COST	TIMES
2	1	1
3	1	n+1
4	1	n^2+1
5	1	n^2
7	1	n^2
9	3	n^2
10	3	n^2
11	1	n^2
12	1	n^2
13	1	n^2
14	1	n^2
15	1	n^2
17	$2(L^2)$	n^2
18	1	n^2
19	1	n^2
23	1	1

Array list add()

$T(n) = 2n^2L^2 + 2n^3 + 8n^2 + 3n + 12$

$O(n^2L^2)$

```java
public boolean isAnagram(String s1, String s2){
    Sorter s = new Sorter();
    String word1 = s.selectionSort(s1).toUpperCase().trim();
    String word2 = s.selectionSort(s2).toUpperCase().trim();
    boolean b = false;

    if(word1.compareTo(word2) == 0){
        b = true;
    }
    return b;
}
```

LINE	COST	TIMES
2	1	1
3	$O(L^2)$	1
4	$O(L^2)$	1
5	1	1
7	1	1
8	1	1
10	1	1

$T(n) = 2L^2 + 5 \rightarrow O(L^2)$

```java
public String selectionSort(String toBeSorted){
    char toSort[] = toBeSorted.toCharArray();
    char greatest;
    int j = toSort.length - 1;
    int spot = 0;

    while(j > -1){
        greatest = ' ';
        for (int i = 0; i <= j; i++) {
            if (toSort[i] > greatest) {
                greatest = toSort[i];
                spot = i;
            }
        }

        toSort[spot] = toSort[j];
        toSort[j] = greatest;
        j--;
    }
    String ret = "";
    for(int l = 0; l < toSort.length; l ++){
        ret += toSort[l];
    }

    return ret;
}
```

LINE	COST	TIMES
40	1	1
41	1	1
42	2	1
43	1	1
45	1	L+1
46	1	L
47	3	L^2+1
48	1	L^2
49	1	L^2
50	1	L^2
54	1	L^2
55	1	L^2
56	1	L^2
58	1	1
59	3	L+1
60	1	L
63	1	1

$T(n) = 6L^2 + 5L + 13 \rightarrow O(L^2)$

Overall O runtime approximation:

Method:	big-oh runtime
main()	$O(n^2L^2)$
prompt()	$O(n^2)$
sArrEnlarge()	$O(n)$
mergeSort()	$O(n \log n)$
merge()	$O(n)$
organizer()	$O(n^2L^2)$
selectionSort()	$O(L^2)$
isAnagram()	$O(L^2)$

The big-oh runtime approximation depends on the methods called from the main() method. After analysis of each excecuted method, the program has the following approximate big-oh runtime:

$$T(n) = n \log n + n^2L^2 + n + C$$

The Big-oh runtime depends on the fastest growing term. The program is therefore:

$O(n^2L^2)$ where n is the list size, and L is the maximum word length.

Question 3

The approximate time complexity of the program is:

$$T(n) \sim C_1 n\log n + C_2 n^2L^2 + C_3 n + C_4$$

$$T(2) \sim C_1 2(1) + \underbrace{C_2 4L^2}_{\text{determining term (fastest growing)}} + C_3(2) + C_4$$

big-oh runtime = $O(4L^2)$