Course: Data Structures, Algorithms, and Their Applications – CPSC 319
Assignment 2 Complexity Analysis

Instructor: Leonard Manzara
T.A.: Xi Wang
Tutorial Section: T04

Student Name: Danielle Jourdain
UCID: 30114566

Question 1

```
                                                                    # Of Operations
Line 1   public static String sortWord(String word) {
Line 2           char[] array = word.toCharArray();                                1
Line 3
Line 4           if(array.length < 2) {                                            2
Line 5               return word;                                                  1
Line 6           }
Line 7
Line 8           return quickChar(array, 0, array.length - 1);
Line 9       }
Line 10
Line 11      private static String quickChar(char array[], int low, int high) {
Line 12          int[] stack = new int[high - low + 1];                            3
Line 13
Line 14          int top = -1;                                                     1
Line 15
Line 16          stack[++top] = low;                                               3
Line 17          stack[++top] = high;                                              3
Line 18
Line 19          while (top >= 0) {                                              log k
Line 20              high = stack[top--];                                          3
Line 21              low = stack[top--];                                           3
Line 22
Line 23              int p = partition(array, low, high);                         1
Line 24
Line 25              if (p - 1 > low) {                                           3
Line 26                  stack[++top] = low;                                      3
Line 27                  stack[++top] = p - 1;                                    4
Line 28              }
Line 29                                                                           3
Line 30              if (p + 1 < high) {                                          4
Line 31                  stack[++top] = p + 1;                                    3
Line 32                  stack[++top] = high;
Line 33              }
Line 34          }
Line 35          return String.valueOf(array);                                    1
Line 36      }
Line 37
Line 38      private static void swapChar(char[] array, int i, int j) {
Line 39          char temp = array[i];                                            3
Line 40          array[i] = array[j];                                             3
Line 41          array[j] = temp;                                                 3
Line 42      }
Line 43
Line 44  private static int partition(char array[], int low, int high) {
Line 45          char pivot = array[high];                                        2
Line 46
Line 47          int i = (low - 1);                                               2
Line 48
Line 49          for (int j = low; j <= high - 1; j++) {                          k
Line 50              if (Character.compare(array[j], pivot) <= 0) {               2
Line 51                  i++;                                                     1
Line 52                  swapChar(array, i, j);
Line 53              }
Line 54          }
Line 55          swapChar(array, i + 1, high);
Line 56
Line 57          return i + 1;                                                    1
Line 58      }
```

To begin the complexity analysis, a few assumptions must be made. Firstly, all declarations, initializations, arithmetic operations, value fetching and *if* statements are considered a single operation with $O(1)$ complexity where k is the number of letters in the word. The second large assumption is that each *for* or *while* loop has a $O(k)$ complexity where k is the number of letters in a word. The worst case will be the partition is chosen as the first or last letter alphabetically in a word.

This version of Quick Sort begins in the sortWord method in Line 1 by converting a String into a character array to simplify the sorting. Then the length of the array is checked to avoid unnecessary sorting. These two operations both have $O(1)$ complexity. The final operation in sortWord is a call to the quickChar function on Line 8.

The quickChar method is where the bulk of the work is done. This implementation of the Quick Sort algorithm is an iterative version. There were issues with a recursive implementation, so an iterative one ensures the algorithm works. It begins by creating a new stack using an array on Line 12. It pushes the initial high and low values to the stack. These operations all have $O(1)$ time complexity. Then the program enters a while loop (Line 19). In this loop, several $O(1)$ operations are performed. The partition function is called which will be discussed in the next paragraph. As the program moves through the loop, it pushes more indices to the stack. These indices will provide the indices for the sub-arrays for Quick Sort to move through. Each time the loop starts over, the new values of high and low will be half the size of the old values. This gives a $O(\log k)$ complexity even for the worst case.

A partition method is used for this implementation of the Quick Sort algorithm. This method starts on Line 45. In this method, the last element of the array is selected as the partition. Then the array is iterated through to move elements smaller than the pivot to the left side and larger elements to the right. This *for* loop has an $O(k)$ complexity. Since the partition method is called from within the *while* loop in quickChar, the product rule will be applied, giving a $O(k \log k)$ for the sorting algorithm so far. From within the *while* loop starting on Line 19 and after it, another helper method, called swapChar is called. This is the final method used and will be discussed in the next paragraph.

Finally, the swapChar method has a $O(1)$ complexity. It begins on Line 39 of the code above, and only contains the basic operations mentioned in the first paragraph giving it the $O(1)$ complexity. Since the swapChar method is called from within the *for* loop of the partition method, the product rule must be used. This gives $O(k \log k)$ complexity. Applying the sum rule for the call to swapChar on Line 56 gives a complexity of $O(k \log k)$.

Finally, going back to the sortWord function, the sum rule can be used on the call to quickChar. This gives a final complexity of $O(k \log k)$ for sorting a word.

Question 2

Theoretical Analysis

For this program, two extreme cases will be considered. The first will be the case where all the words in the input text file are anagrams of one another, and the words are in alphabetical order, so the output will be all the words on one line of a new text file. The second extreme case will be the case where none of the words in the input text file are anagrams, so the output will be the words sorted into alphabetical order. To complete this complexity analysis, several assumptions must be made. The first assumption is that all basic operations including declarations, initializations, arithmetic operations, *if* statements, and value fetching have $O(1)$ complexity. The second assumption is that all *while* or *for* loops have $O(N)$ complexity unless otherwise specified.

Case 1 – All Anagrams

The program begins in the main method, starting on Line 15 of the Assign2.java file. First, the command line arguments are checked, starting with the number of arguments using a simple *if* statement which will have $O(1)$ complexity. Then, the function checkTXT is called twice. This function begins on Line 54 of the Assign2.java file. In this function, an *if* statement is used to check the extension on the filenames entered. This would give an $O(1)$ time complexity. Going back to the main method, and applying the sum rule, we get a $O(1)$ complexity for the program so far.

After checking the command-line arguments, the input text file is read into a String array using the readFile method, called on Line 27 of main. This method begins on Line 74 of Assign2.java. Here, the file is read using a BufferedReader object. It goes through the input file line by line and adds the words to a String. This is done using a *while* loop, which would give a $O(N)$ complexity. After this loop, the long String is turned into a String array using the split method. This would have an $O(1)$ complexity. Applying the sum rule to readFile gives an overall $O(N)$ complexity for the function. Going back to main and applying the sum rule again gives a current complexity of $O(N)$.

Next, the array of references to. LinkedLists is created using the createList method, which begins on Line 116 of Assign2.java. In this method, the array of LinkedLists is created, and the first Node of the first list is inserted specially. Then the program enters a set of nested *for* loops. One of these loops iterates through the String array holding the words, and the other iterates through the LinkedList array. There is an *if* statement inside the inner loop to find the check if the current spot in the LinkedList array is null, and another *if* statement inside to check if the words are anagrams.

To check if two words are anagrams of one another, the method checkWord is used. In this function, two words are given as arguments, then they are sorted into alphabetical order using the sortWord method that starts on Line 9 of the Sort.java file. This is an altered version of the Quick Sort algorithm that was used in Assignment 1, and it was analyzed above in Question 1. From that analysis, it was found that the complexity is $O(L \log L)$ where L is the maximum length of any word. Back in checkWord, the sum rule can be applied with all the other simple operations to give an overall complexity of $O(L \log L)$ overall for checkWord.

Since this is the case where all the words are anagrams, the program will always enter the *if* statement on Line 126 of Assign2.java. Then, the word will be inserted into the LinkedList using the insertNode function. This function can be found on Line 48 of the LinkedList.java file. In this method, a new Node is created that holds the word as data. Since this case assumes the input file had the words in alphabetical order, the new Node will have to be inserted at the end of the LinkedList. To do this, a *while* loop is used, which will give $O(N)$ complexity in this case. The rest of the insertNode function consists of simple operations that will have $O(1)$ time complexity. Using the sum rule on the function gives $O(N)$ complexity. Back in the createList function, there is a break statement that will stop the inner *for* loop from running further. Since this is the case where every word is an anagram of all others, the inner *for* loop will always be broken on the first iteration. This means the complexity for the *for* loops will only be $O(N)$. Using the product rule on the *for* loops, *if* statement, and the insertNode function gives a time complexity of $O(N^2 L \log L)$. Before returning to the main function, extra null space in the array of references is trimmed out using the shortenList method. This method uses two separate *for* loops to remove all null space in the array. This would give $O(N)$ complexity. Applying the sum rule to the rest of insertNode gives an overall complexity of $O(N^2 L \log L)$.

Back in the main method, applying the sum rule on all the work so far gives a complexity of $O(N^2 L \log L)$. The newly created array of LinkedLists is sorted using the quick method found starting on Line 116 of the Sort.java file. Since all the words are stored in one LinkedList, the length of the array of references will be 1. This means the array is already sorted, so the complexity for quick is $O(1)$. The program returns to Line 41 of the main method, where it will write the LinkedList array to a text file using the writeToFile method.

This method starts on Line 201 of Assign2.java. In this method, a *while* loop is nested within a *for* loop. The *for* loop iterates through the array of references, and the *while* loop iterates through the Nodes of the LinkedList. Since all the words are stored in a single LinkedList, the inner *while* loop will have $O(N)$ complexity, and the outer *for* loop has $O(1)$ complexity. Going back to the main method and applying the sum rule one last time gives an overall complexity of $O(N^2 L \log L)$ for the case where all the words are anagrams of one another.

Case 2 – No Anagrams

This analysis starts off the same as the case when all words are anagrams. Up to Line 27 of Assign2.java it is identical to the other case. This gives a current complexity of $O(N)$, the same as Case 1 at this point.

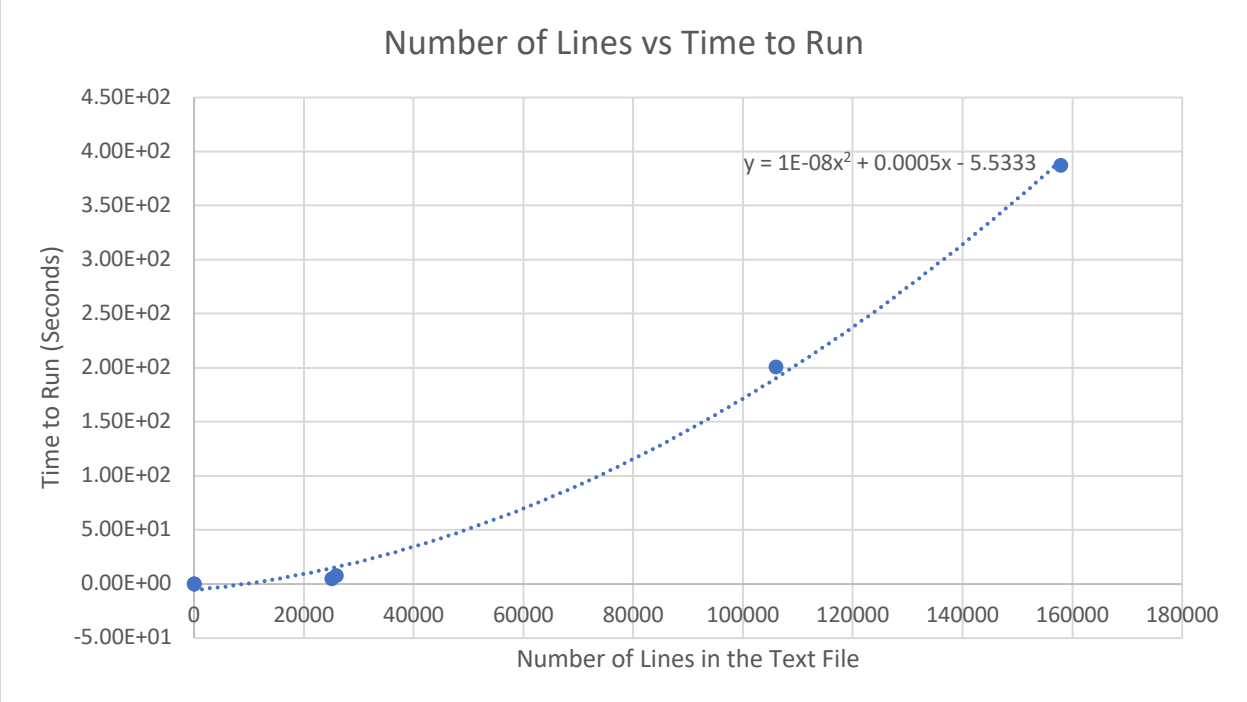
The difference in the two cases begins in the createList method which starts on Line 116 of Assign2.java. Once again, there is a set of nested *for* loops. In this case, the current word will be checked against each of the LinkedLists. Since there are no anagrams, all of these will not be a match. Instead, the inner *for* loop will continue to compare and check until an open space is found in the LinkedList array, and the word will be inserted as the first Node in a new LinkedList. The words are checked to be anagrams the same way as they are in Case 1, which gives a complexity of $O(L \log L)$. In this case however, both *for* loops are completely iterated through, which will give a current complexity of $O(N^2 L \log L)$ using the product rule.

The shortenList method works the same as it does for Case 1. However, in this case, there is no null space to remove. This will still give $O(N)$ complexity for the function. Applying the sum rule back in createList gives a current complexity of $O(N^2 L \log L)$.

Returning to Line 33 of main, the next step is to sort the LinkedList array. This is done using the quick method found in Line 116 of Sort.java. Unlike Case 1, this array will need to be sorted. The sorting is done using an adapted version of the Quick Sort used in Assignment 1. The only major change is the data type being sorted. Otherwise, the details of the implementation are very similar. From Assignment 1, it is known that the complexity of the Quick Sort algorithm is $O(N \log N)$ in best, worst, and average cases.

Using the sum rule on all the work done so far gives a complexity of $O(N^2 L \log L)$. To write the LinkedList array to a text file, the same function is used as in Case 1 – writeToFile. In this case however, the outer *for* loop will be executed N times while the inner *while* loop will be executed only once for each element. This will give a complexity of $O(N)$ for the function. Returning to the main method and applying the sum rule a final time gives a complexity of $O(N^2 L \log L)$.

Experimental Analysis



From this graph it shows that the actual trendline gives a $O(N^2)$ complexity, which matches the first part of the complexity found in the theoretical analysis. However, this graph does not consider the length of the words being compared (the L values) in the theoretical analysis. This graph only considers the number of words being sorted (the N values). To see if experimental results align with theoretical ones, more tests should be run, keeping the number of words constant, but changing the average size of the words being sorted from test to test. The experimental results fall somewhere between the two cases considered above. In all the files

tested, some words had anagrams, and some did not. However, both cases considered in the theoretical analysis gave $O(N^2)$ complexity, so it follows that any "in-between" cases would also have $O(N^2)$ time complexity.

Conclusion

Since the theoretical and experimental analysis both gave a $O(N^2)$ time complexity where N is the number of words in the input text file, it can be concluded that this is accurate for the number of words. There were no experimental tests to see what the time complexity is for the size of the words being analyzed, so the theoretical results are the only indicator of this complexity. Overall it can be concluded that the time complexity of this program will be $O(N^2 L \log L)$.