CPSC 319
Assignment 2

Aleksander Berezowski
Tutorial Section: T03
TA Name: Shopon
Email: aleksander.berezowsk@ucalgary.ca

Contents

Figures

Tables

Equations

Data collected

Below is all the data collected during experimental runs. A separate driver class was used to run the *Assign2* program with all the text files needed automatically without needing to manually rerun the program every time. Tabel 1 shows how long each input text took to run, including time conversions. Figure 1 shows the output of the driver class, which run *Assign2* seven times automatically. Figure 2 is a scatter plot with the number of elements as the independent variable on the x axis and the runtime as the dependant variable on the y axis; on this scatter plot is a line of best fit generated by Excel. Finally, Figure 2’s trendline has equation shown in Equation 1.

Name	Elements	Time (ns)	Time (s)	Time (min
input	8	14963401	0.014963	0.00025
example_1	9	5945500	0.005946	0.00010
example_2	11	1646099	0.001646	0.00003
example_3	25125	5005888499	5.005888	0.08343
large	157858	6.98659E+11	698.6593	11.64432
medium	105990	1.64966E+11	164.9662	2.74944
small	24868	7692470001	7.69247	0.12821

Table 1 - Program Run Time

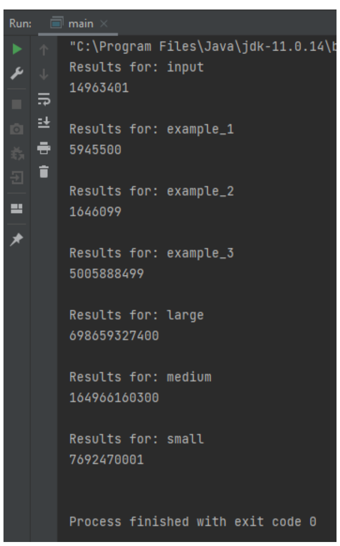

Figure 1 - Program Output

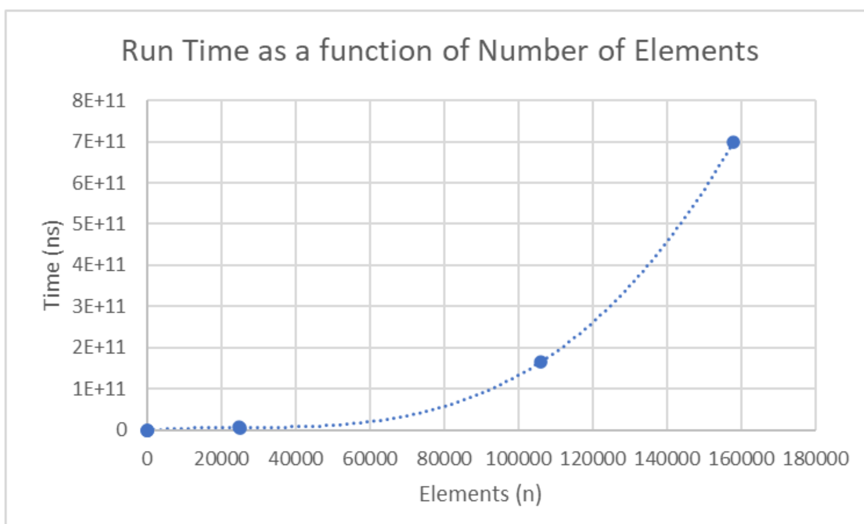

Figure 2 - Run Time Graph

$$t(n) = 0.0003x^3 - 22.601x^2 + 634\,102x + 5\,000\,000$$

Equation 1 - Trendline Equation from Figure 2

Question 1

What is the worst-case complexity of your algorithm when checking if two words are anagrams of each other? Express this using big-O notation and use the variable k to represent the number of letters in each word. Support this with a theoretical analysis of your code.

The algorithm for checking if two words are anagrams of each other has two functions within it: *sortCharArray* and *checkAnagram*. The first function, *sortCharArray*, is used within the *checkAnagram* function, therefore *sortCharArray* will be analyzed first and then used to analyze *checkAnagram*.

Line 1	private char[] sortCharArray(char[] arr){	
Line 2	char temp;	1
Line 3		
Line 4	int i = 0;	2
Line 5	while (i < arr.length) {	k
Line 6	int j = i + 1;	2
Line 7	while (j < arr.length) {	k
Line 8	if (arr[j] < arr[i]) {	3
Line 9		
Line 10	temp = arr[i];	2
Line 11	arr[i] = arr[j];	2
Line 12	arr[j] = temp;	2
Line 13	}	
Line 14	j += 1;	1
Line 15	}	
Line 16	i += 1;	1
Line 17	}	
Line 18		
Line 19	return arr;	1
Line 20	}	

Mathematically:

$$1 + 2 + k(2 + k(3 + 2 + 2 + 2 + 1) + 1) + 1 = 4 + 3k + 10k^2 = O(k^2)$$

Intuitively:

This function has two *while* loops, each of which are *O(k)*. When multiplying these two *while* loops together, you get *O(k)*O(k) = O(k*k) =* $O(k^2)$, which confirms the mathematical proof above.

Line 1	private boolean checkAnagram(String str1, String str2) {	
Line 2	if (str1.length() != str2.length())	3
Line 3	return false;	1
Line 4		
Line 5	char[] a = str1.toCharArray();	2

Line 6
Line 7
Line 8
Line 9
	`char[] b = str2.toCharArray();`
	
	`return Arrays.equals(sortCharArray(a), sortCharArray(b));`
	`}`
		2
		
		$1 + 2(4+3k+10k^2)$

Mathematically:

$$3 + 1 + 2 + 2 + 1 + 2(4 + 3k + 10k^2) = 17 + 6k + 20k^2 = O(k^2)$$

Intuitively:

This function doesn't have any loops or iterations, so this function's big-O notation is the sum of its constants plus the big-O notation calculation for *sortCharArray* multiplied by two due to *sortCharArray* being called twice, therefore the function should still be $O(k^2)$, which again confirms the mathematical proof above.

In conclusion, the worst-case time complexity of this algorithm when checking if two words are anagrams of each other is $O(k^2)$. This is proven due to both the mathematical proof as well as the intuitive inspection leading to the same conclusion.

Question 2

Let N be the number of words in the input word list, and L be the maximum length of any word. What is the big-O running time of your program? Justify your answer using both a theoretical analysis and experimental data (i.e. timing data).

Theoretical Analysis

Node.java

```
public class Node {
  private String data;
  private Node next;

  public Node(String data){
    this.setData(data);
    this.setNext(null);
  }

  public void setData(String data){ this.data = data;}

  public void setNext(Node next){ this.next = next;}

  public String getData(){ return this.data;}

  public Node getNext(){ return this.next;}
}
```

Node constructor: *O(1)*

This constructor has a big-O notation of *O(1)*, meaning it is constant. This is due to the constructor having 2 constant commands, which sum up to a constant big-O of *O(1)*.

setData Method: *O(1)*

This method has a big-O notation of *O(1)*, meaning it is constant. This is due to the method having 1 constant command, which sum up to a constant big-O of *O(1)*.

setNext Method: *O(1)*

This method has a big-O notation of *O(1)*, meaning it is constant. This is due to the method having 1 constant command, which sum up to a constant big-O of *O(1)*.

getData Method: *O(1)*

This method has a big-O notation of *O(1)*, meaning it is constant. This is due to the method having 1 constant command, which sum up to a constant big-O of *O(1)*.

getNext Method: *O(1)*

This method has a big-O notation of *O(1)*, meaning it is constant. This is due to the method having 1 constant command, which sum up to a constant big-O of *O(1)*.

LinkedList.java

```
public class LinkedList {
   private Node head;

   public void setHead(Node newHead){this.head = newHead;}

   public Node getHead(){ return this.head;}

   public LinkedList(String data){insert(data);}

   public void insert(String data) {
      Node new_node = new Node(data);
      new_node.setNext(null);

      if (this.head == null) { this.head = new_node;
      }
      else {
         Node last = this.head;
         while (last.getNext() != null) {
            last = last.getNext();
         }

         last.setNext(new_node);
      }
   }

   public void push(String new_data) {
      Node new_Node = new Node(new_data);

      new_Node.setNext(head);

      head = new_Node;
   }

   public void printList() {
      Node currNode = this.head;

      System.out.print("\nLinkedList: ");

      while (currNode != null) {
         System.out.print(currNode.getData() + " ");
```

```
      currNode = currNode.getNext();
    }
  }

  public String toString() {
    Node currNode = this.head;
    StringBuilder returnString = new StringBuilder(3000);
    String appendString = "";

    while (currNode != null) {
      appendString = currNode.getData() + " ";
      returnString.append(appendString);

      currNode = currNode.getNext();
    }
    returnString.setLength(returnString.length() - 1);
    return returnString.toString();
  }

  public static LinkedList deleteByKey(LinkedList list, String key) {
    Node currNode = list.head, prev = null;

    if (currNode != null && currNode.getData() == key) {
      list.head = currNode.getNext(); // Changed head

      System.out.println(key + " found and deleted");

      return list;
    }

    while (currNode != null && currNode.getData() != key) {
      prev = currNode;
      currNode = currNode.getNext();
    }

    if (currNode != null) {
      prev.setNext(currNode.getNext());
      System.out.println(key + " found and deleted");
    }
    if (currNode == null) {
      // Display the message
      System.out.println(key + " not found");
    }
    return list;
  }

  public static LinkedList deleteAtPosition(LinkedList list, int index) {
```

```
        Node currNode = list.head, prev = null;

        if (index == 0 && currNode != null) {
            list.head = currNode.getNext(); // Changed head

            System.out.println(
                    index + " position element deleted");

            return list;
        }

        while (currNode != null) {

            if (counter == index) {
                prev.setNext(currNode.getNext());

                System.out.println(
                        index + " position element deleted");
                break;
            }
            else {
                prev = currNode;
                currNode = currNode.getNext();
                counter++;
            }
        }

        if (currNode == null) {
            // Display the message
            System.out.println(
                    index + " position element not found");
        }

        return list;
    }

    public void swapNodes(String x, String y) {
        if (x == y)
            return;

        Node prevX = null, currX = head;
        while (currX != null && !Objects.equals(currX.getData(), x)) {
            prevX = currX;
            currX = currX.getNext();
        }

        Node prevY = null, currY = head;
        while (currY != null && !Objects.equals(currY.getData(), y)) {
```

```
        prevY = currY;
        currY = currY.getNext();
      }

      if (currX == null || currY == null)
        return;

      if (prevX != null)
        prevX.setNext(currY);
      else // make y the new head
        head = currY;

      if (prevY != null)
        prevY.setNext(currX);
      else // make x the new head
        head = currX;

      Node temp = currX.getNext();
      currX.setNext(currY.getNext());
      currY.setNext(temp);
    }

    public Node insertionSort(){
      Node dummy = new Node(null);
      Node curr = this.head;

      while (curr != null) {
        Node prev = dummy;

        while (prev.getNext() != null &&
prev.getNext().getData().compareTo(curr.getData()) < 0) {
          prev = prev.getNext();
        }

        Node next = curr.getNext();
        curr.setNext(prev.getNext());
        prev.setNext(curr);

        curr = next;
      }

      return dummy.getNext();
    }
}
```

setHead Method: *O(1)*

This method has a big-O notation of *O(1)*, meaning it is constant. This is due to the method having 1 constant command, which sum up to a constant big-O of *O(1)*.

getHead Method: *O(1)*

This method has a big-O notation of *O(1)*, meaning it is constant. This is due to the method having 1 constant command, which sum up to a constant big-O of *O(1)*.

LinkedList Constructor: *O(n)*

This constructor is *O(n)* because it calls the *insert* method, which is defined below as *O(n)*, therefore making this constructor linear.

insert Method: *O(n)*

This method has a big-O notation of *O(n)*, meaning its time-complexity is linear. This is due to the method having 1 *while* loop on line 18 that iterates over the linked list, which can be a maximum of *n-1* iterations, therefore making this method *O(n)*.

push Method: *O(1)*

This method is a constant big-O notation of *O(1)* because the method is made of 3 constant commands, which sum up to a constant run time not based on any list length.

printList: *O(n)*

The big-O notation for this method is linear, as in *O(n)*, due to the *while* loop on line 39 that iterates over the entire list.

toString: *O(n)*

The big-O notation for this method is also linear for the same reason as printList: the *while* loop on line 51 iterates over the entire list.

swapNodes: *O(n)*

The big-O notation for this method is linear due to the worst-case scenario that line 130 could loop through the entire list, making it a big-O notation of *O(n)*.

insertionSort: $O(n^2)$

This method has 2 *while* loops: one that iterates through the entire loop, and one loop that could iterate through the entire loop. The loop on line 163 will iterate through the entire loop, so *n* times, while the loop on line 166 could iterate *n-1* times. The nested loops results in a $O(n^2)$ big-O notation.

Assign2.java

Line 1	import java.io.*;
Line 2	import java.util.*;
Line 3	
Line 4	public class Assign2 {
Line 5	private String inputFileName = "";
Line 6	private String outputFileName = "";

```
    private boolean debuggingMode = true;
    private String[] inputWordList;
    private LinkedList[] linkedLists;
    private long startTime;

    private void printHeader(String methodName) {
        if (debuggingMode) {
            System.out.println("\n\t - - - - " + methodName + " - - - -");
        }
    }

    private void readFile() {
        printHeader("readFile");
        List<String> listOfStrings = new ArrayList<String>();
        BufferedReader bf;

        try {
            bf = new BufferedReader(new FileReader("src/"+inputFileName));
        } catch (FileNotFoundException e) {
            throw new RuntimeException(inputFileName + " not found, " + e.getMessage());
        }

        try {
            String line = bf.readLine();

            while (line != null) {
                listOfStrings.add(line);
                line = bf.readLine();
            }
        } catch (Exception e){
            throw new RuntimeException("Error reading file, exception caught " +
e.getMessage());
        }

        try {
            bf.close();
        } catch (IOException e) {
            throw new RuntimeException("Error closing file, exception caught " +
e.getMessage());
        }

        inputWordList = listOfStrings.toArray(new String[0]);

        if(debuggingMode) {
            System.out.println("\nPrint input file's unsorted contents:");
            int lineNumber = 1;
            for (String str : inputWordList) {
                System.out.println("Line " + String.valueOf(lineNumber) + ": " + str);
```

```
                lineNumber++;
            }
        }
    }

    private boolean checkAnagram(String str1, String str2) {
        printHeader("checkAnagram");

        if (str1.length() != str2.length())
            return false;

        char[] a = str1.toCharArray();
        char[] b = str2.toCharArray();

        return Arrays.equals(sortCharArray(a), sortCharArray(b));
    }

    private char[] sortCharArray(char[] arr){
        char temp;

        int i = 0;
        while (i < arr.length) {
            int j = i + 1;
            while (j < arr.length) {
                if (arr[j] < arr[i]) {

                    temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                }
                j += 1;
            }
            i += 1;
        }

        return arr;

    }

    private void quickSortLinkedListArray(){
        printHeader("quickSortLinkedListArray");
        quickSortLinkedListArrayAlg(linkedLists, 0, linkedLists.length - 1);

        if(debuggingMode) {
            System.out.println("\nPrint input file's sorted contents:");
            int lineNumber = 1;
            for (LinkedList list : linkedLists) {

```

```
                System.out.println("Line " + String.valueOf(lineNumber) + ": " +
list.getHead().getData());
                lineNumber++;
            }
        }
    }

    private void quickSortLinkedListArrayAlg(LinkedList[] arr, int l, int h) {
        printHeader("quickSortLinkedListArrayAlg");

        int[] stack = new int[h - l + 1];
        int top = -1;
        stack[++top] = l;
        stack[++top] = h;

        while (top >= 0) {
            h = stack[top--];
            l = stack[top--];
            int p = partitionLinkedListArray(arr, l, h);

            if (p - 1 > l) {
                stack[++top] = l;
                stack[++top] = p - 1;
            }

            if (p + 1 < h) {
                stack[++top] = p + 1;
                stack[++top] = h;
            }
        }
    }

    private int partitionLinkedListArray(LinkedList[] arr, int low, int high) {
        printHeader("partitionLinkedListArray");

        LinkedList pivot = arr[high];

        int i = (low - 1);
        for (int j = low; j <= high - 1; j++) {
            if (arr[j].getHead().getData().compareTo(pivot.getHead().getData()) < 0) {
                i++;

                Node temp = arr[i].getHead();
                arr[i].setHead(arr[j].getHead());
                arr[j].setHead(temp);
            }
        }

```

```
        Node temp = arr[i + 1].getHead();
        arr[i + 1].setHead(arr[high].getHead());
        arr[high].setHead(temp);

        return i + 1;
    }

    private void createLinkedListArray(){
        printHeader("createLinkedListArray");

        linkedLists = new LinkedList[inputWordList.length];

        linkedLists[0] = new LinkedList(inputWordList[0]);

        if(debuggingMode){
            System.out.println("List after first position populated");
            printLinkedLists();
        }

        for(int j = 1; j < inputWordList.length; j++){
            for(int i = 0; i < linkedLists.length; i++) {
                if (linkedLists[i] != null) {
                    if (checkAnagram(inputWordList[j], linkedLists[i].getHead().getData())) {
                        linkedLists[i].insert(inputWordList[j]);
                        if(debuggingMode){
                            System.out.println(inputWordList[j] + " is an anagram of "
                                    + linkedLists[i].getHead().getData() + ", inserting into list" );
                        }
                        break;
                    } else if (debuggingMode){
                        System.out.println(inputWordList[j] + " is not an anagram of "
                                + linkedLists[i].getHead().getData());
                    }
                } else {
                    linkedLists[i] = new LinkedList(inputWordList[j]);
                    if(debuggingMode){
                        System.out.println("Didn't find list with " + inputWordList[j] +", making new
list");
                    }
                    break;
                }
            }
        }

        if(debuggingMode){
            System.out.println("List before insertion sort");
            printLinkedLists();
        }
```

```
        for (LinkedList linkedList : linkedLists) {
            if (linkedList != null) {
                linkedList.setHead(linkedList.insertionSort());
            }
        }

        if(debuggingMode){
            System.out.println("List after insertion sort");
            printLinkedLists();
        }


    }

    private void printLinkedLists() {
        for (LinkedList s : linkedLists) {
            if (s != null) {
                if (s.getHead() != null) {
                    s.printList();
                } else {
                    System.out.println("null");
                }
            } else {
                System.out.println("nullptr");
            }
        }
    }

    private void shrinkLinkedListArray(){
        printHeader("shrinkLinkedListArray");

        if(debuggingMode){
            System.out.println("List with nulls");
            printLinkedLists();
        }

        ArrayList<LinkedList> list = new ArrayList<LinkedList>();
        for (LinkedList s : linkedLists)
            if (s != null)
                list.add(s);
        linkedLists = list.toArray(new LinkedList[list.size()]);


        if(debuggingMode){
            System.out.println("List without nulls");
            printLinkedLists();
        }
```

```
  }

  private void printFile(){
    printHeader("printFile");

    try {
      PrintWriter myFile = new PrintWriter(outputFileName + ".txt", "UTF-8");
      for (LinkedList list : linkedLists) {
        if(debuggingMode){
          System.out.println("Printing to file: " + list.toString());
        }
        myFile.println(list.toString());
      }
      myFile.close();
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }

  public void startOfProgram(){
    startTime = System.nanoTime();
  }

  public void endOfProgram(){
    System.out.println(System.nanoTime() - startTime);
  }

  public Assign2(String[] args) {
    startOfProgram();

    if(args.length < 2){
      throw new IllegalArgumentException("Wrong number of input arguments: too few arguments");
    } else if(args.length > 3){
      throw new IllegalArgumentException("Wrong number of input arguments: too many arguments");
    }

    if(args.length == 3){
      try{
        Integer.parseInt(args[2]);
      } catch (NumberFormatException e){
        throw new IllegalArgumentException("Debugging mode argument is not a number");
      }

      if (Integer.parseInt(args[2]) == 0){
```

```
Line 295            debuggingMode = false;
Line 296        } else if (Integer.parseInt(args[2]) != 1){
Line 297            throw new IllegalArgumentException("Debugging mode argument is not 0 or 1");
Line 298        }
Line 299      } else {
Line 300        debuggingMode = false;
Line 301      }
Line 302
Line 303      inputFileName = args[0] + ".txt";
Line 304      outputFileName = args[1];
Line 305
Line 306      readFile();
Line 307
Line 308      createLinkedListArray();
Line 309      shrinkLinkedListArray();
Line 310      quickSortLinkedListArray();
Line 311
Line 312      printFile();
Line 313
Line 314      endOfProgram();
Line 315    }
Line 316
Line 317    public static void main(String[] args){
Line 318      new Assign2(args);
Line 319    }
          }
```

printHeader: *O(1)*

> This method is just *O(1)* because this method is just an *if* statement followed by a *println* statement.

readFile: *O(n)*

> This method has time complexity of *O(n)* due to the *while* loop on line 32. Therefore, this method has an exponential time complexity.

checkAnagram: *O(l^2)*

> As shown in Question 1, the worst-case complexity of this algorithm when checking if two words are anagrams of each other is O(l^2).

quickSortLinkedListArray: *O(n^2)*

> This method uses *quickSortLinkedListArrayAlg* and *partitionLinkedListArray* as helper methods, and has a worst-case of *O(n^2)* as proven in Assignment 1.

createLinkedListArray: $O(n^3\ l^2)$

This method has 2 *for* loops on line 169 and line 170, each of $O(n)$ complexity with a combined big-O of $O(n^2)$. Then on line 172 there is a function call for *checkAnagram* which has been previously shown to be $O(l^2)$; multiplying $O(n^2)$ by $O(l^2)$ gives a time complexity of $O(n^2\ l^2)$. Finally, on line 173 there is a function call for *LinkedList*'s *insert* method, which was previously shown to be $O(n)$. Multiplying everything together, the time complexity for *createLinkedListArray* is $O(n^3\ l^2)$.

printLinkedLists: $O(n^2)$

This method has a big-O notation of $O(n^2)$ due to the *for* loop that iterates over the entire list of *LinkedList*'s, and then it calling the method *printList* that was shown to be $O(n)$. Multiplying these together gives a total time complexity of $O(n^2)$.

shrinkLinkedListArray: $O(n)$

This method has a linear time complexity due to the *for* loop that iterates over the entire list, leading to a big-O notation of $O(n)$.

printFile: $O(n^2)$

This method has the *toString* method nested inside a *for* loop. Both are $O(n)$ time complexity, therefore nesting them results in a $O(n^2)$ big-O notation.

startOfProgram: $O(1)$

This method is extremely simple and only has 1 constant command, therefore giving a big-O notation of $O(1)$.

endOfProgram: $O(1)$

This method is extremely simple and only has 2 constant commands, therefore giving a big-O notation of $O(1)$.

Assign2: $O(n^3\ l^2)$

Assign2 calls *readFile*, *createLinkedListArray*, *shrinkLinkedListArray*, *quickSortLinkedListArray*, and *printFile* in series, therefore the big-O notation will be the most complex of all these functions due to the big-O summation rule. The most complex method is *createLinkedListArray*, therefore Assign2 will inherit its time complexity of $O(n^3\ l^2)$.

main: $O(n^3\ l^2)$

All the main does is call Assign2, therefore the big-O complexity is the same as the Assign2 function.

Conclusion

In conclusion, the big-O notation for the time complexity of this program is $O(n^3\ l^2)$, with n being the amount of words and l being the amount of letters in a word.

Experimental Analysis

The theoretical analysis came to a conclusion of $O(n^3\ l^2)$, which shows a dependency on both number of words, *n*, and the length of each word, *l*. Therefore, experimental analysis will have to be done on the number of words as well as the word length.

For experimental analysis in relation to the number of words, the data from Table 1, illustrated in Figure 2, clearly shows an exponential relationship, which supports the conclusion from the theoretical analysis. Equation 1 shows the equation for the trendline, which was automatically calculated in Excel; This equation has a highest power of 3, which further supports the theoretical analysis' conclusion of n^3.

The issue with this experiment is no files were provided with the same number of words, but each word having substantially more or less letters. This means the data collection doesn't take word length into account; the data collection only reflects the amount of words. Therefore, the dependency on word length cannot be proven or disproven due to the data set provided.

In conclusion, the experimental analysis supports a big-O notation with a cubic relationship to the amount of words in a list, as in $O(n^3)$, but due to how this experiment was run no conclusion can be drawn about the theoretical squared relationship with the word length, shown as $O(l^2)$.

Conclusion

When letting n be the number of words in the input word list and l be the maximum length of any word, the theoretical big-O running time of this program is $O(n^3\ l^2)$. However, the experimental analysis can only confirm the big-O run time to be at least $O(n^3)$. While this experiment does not disprove the length of a word being a factor in run time, it does not conclusively prove it either. This means the run time is proven to be $O(n^3)$, although it is possible to assume the actual time complexity is $O(n^3\ l^2)$. However, to definitively prove this, more tests would need to be run with different input files that focused on word length over word quantity.

Bibliography

- “Java Create and Write To Files,” w3schools. [Online]. Available: https://www.w3schools.com/java/java_files_create.asp
- “Read file into an array in Java,” GeeksForGeeks. [Online]. Available: https://www.geeksforgeeks.org/read-file-into-an-array-in-java/
- “Checking if two strings are permutations of each other,” Stack Overflow. [Online]. Available: https://stackoverflow.com/questions/2131997/checking-if-two-strings-are-permutations-of-each-other
- “Sort a string in Java,” GeeksForGeeks. [Online]. Available: https://www.geeksforgeeks.org/sort-a-string-in-java-2-different-ways/
- “Quick Sort,” GeeksForGeeks. [Online]. Available: https://www.geeksforgeeks.org/quick-sort/
- “Java Quicksort Giving Stackoverflow Error,” Reddit. [Online]. Available: https://www.reddit.com/r/learnprogramming/comments/1yzs5z/java_quicksort_giving_stackoverflowerror_for/
- “Iterative Quick Sort,” GeeksForGeeks. [Online]. Available: https://www.geeksforgeeks.org/iterative-quick-sort/
- “Remove null value from string array in Java,” Stack Overflow. [Online]. Available: https://stackoverflow.com/questions/4150233/remove-null-value-from-string-array-in-java
- “Java Create and Write To Files,” W3Schools. [Online]. Available: https://www.w3schools.com/java/java_files_create.asp
- “Quick sort in 4 minutes,” YouTube. [Online]. Available: https://www.youtube.com/watch?v=Hoixgm4-P4M&t=49s
- “147. Insertion Sort List,” GeeksForGeeks. [Online]. Available: https://leetcode.com/problems/insertion-sort-list/solution/
- “https://www.geeksforgeeks.org/implementing-a-linked-list-in-java-using-class/,” GeeksForGeeks. [Online]. Available: https://www.geeksforgeeks.org/implementing-a-linked-list-in-java-using-class/