

# **CPSC 319**

## **Assignment 2**

Aleksander Berezowski

Tutorial Section: T03

TA Name: Shopon

Email: [aleksander.berezowsk@ucalgary.ca](mailto:aleksander.berezowsk@ucalgary.ca)

## Contents

Figures.....	2
Tables .....	2
Equations .....	2
Data collected .....	3
Question 1.....	4
Question 2.....	6
Theoretical Analysis .....	6
Node.java .....	6
LinkedList.java.....	7
Assign2.java.....	11
Conclusion.....	19
Experimental Analysis.....	20
Conclusion.....	20
Bibliography .....	21

## Figures

Figure 1 - Program Output.....	3
Figure 2 - Run Time Graph .....	3

## Tables

Table 1 - Program Run Time.....	3
---------------------------------	---

## Equations

Equation 1 - Trendline Equation from Figure 2.....	3
--	---

## 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

```

Run: main
"C:\Program Files\Java\jdk-11.0.14\bin\java.exe" -Djava.class.path=.\Assign2.jar -Djava.library.path=.\lib\Assign2.jar -jar Assign2.jar
Results for: input
14963401
Results for: example_1
5945500
Results for: example_2
1646099
Results for: example_3
5005888499
Results for: large
698659327400
Results for: medium
164966160300
Results for: small
7692470001
Process finished with exit code 0

```

Figure 1 - Program Output

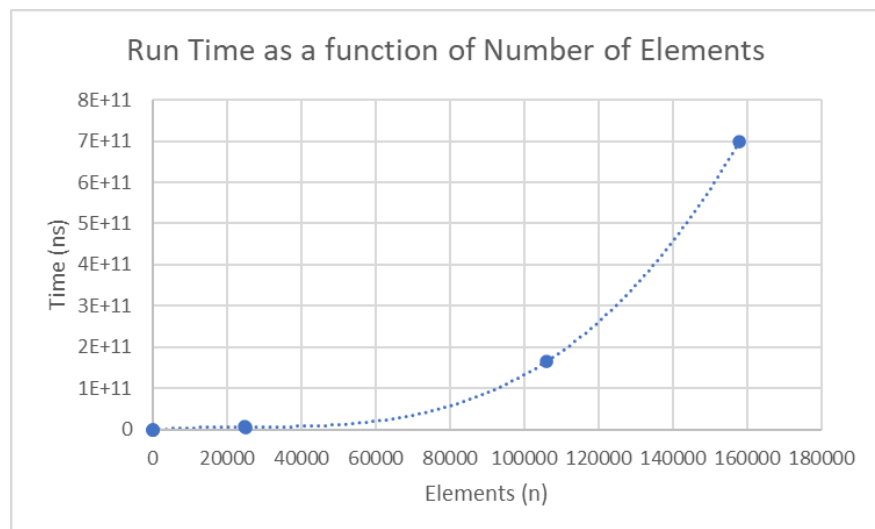


Figure 2 - Run Time Graph

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

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	<code>char[] b = str2.toCharArray();</code>	2
Line 7		
Line 8	<code>return Arrays.equals(sortCharArray(a), sortCharArray(b));</code>	1 + 2(4+3k+10k <sup>2</sup> )
Line 9	<code>}</code>	

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

Line 1	public class Node {
Line 2	private String data;
Line 3	private Node next;
Line 4	
Line 5	public Node(String data){
Line 6	this.setData(data);
Line 7	this.setNext(null);
Line 8	}
Line 9	
Line 10	public void setData(String data){ this.data = data;}
Line 11	
Line 12	public void setNext(Node next){ this.next = next;}
Line 13	
Line 14	public String getData(){ return this.data;}
Line 15	
Line 16	public Node getNext(){ return this.next;}
Line 17	}

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

Line 1	public class LinkedList {
Line 2	private Node head;
Line 3	
Line 4	public void setHead(Node newHead){this.head = newHead;}
Line 5	
Line 6	public Node getHead(){ return this.head;}
Line 7	
Line 8	public LinkedList(String data){insert(data);}
Line 9	
Line 10	public void insert(String data) {
Line 11	Node new_node = new Node(data);
Line 12	new_node.setNext(null);
Line 13	
Line 14	if (this.head == null) { this.head = new_node;
Line 15	}
Line 16	else {
Line 17	Node last = this.head;
Line 18	while (last.getNext() != null) {
Line 19	last = last.getNext();
Line 20	}
Line 21	
Line 22	last.setNext(new_node);
Line 23	}
Line 24	}
Line 25	
Line 26	public void push(String new_data) {
Line 27	Node new_Node = new Node(new_data);
Line 28	
Line 29	new_Node.setNext(head);
Line 30	
Line 31	head = new_Node;
Line 32	}
Line 33	
Line 34	public void printList() {
Line 35	Node currNode = this.head;
Line 36	
Line 37	System.out.print("\nLinkedList: ");
Line 38	
Line 39	while (currNode != null) {
Line 40	System.out.print(currNode.getData() + " ");

```

Line 41         currNode = currNode.getNext();
Line 42     }
Line 43 }
Line 44
Line 45
Line 46 public String toString() {
Line 47     Node currNode = this.head;
Line 48     StringBuilder returnString = new StringBuilder(3000);
Line 49     String appendString = "";
Line 50
Line 51     while (currNode != null) {
Line 52         appendString = currNode.getData() + " ";
Line 53         returnString.append(appendString);
Line 54
Line 55         currNode = currNode.getNext();
Line 56     }
Line 57     returnString.setLength(returnString.length() - 1);
Line 58     return returnString.toString();
Line 59 }
Line 60
Line 61 public static LinkedList deleteByKey(LinkedList list, String key) {
Line 62     Node currNode = list.head, prev = null;
Line 63
Line 64     if (currNode != null && currNode.getData() == key) {
Line 65         list.head = currNode.getNext(); // Changed head
Line 66
Line 67         System.out.println(key + " found and deleted");
Line 68
Line 69         return list;
Line 70     }
Line 71
Line 72     while (currNode != null && currNode.getData() != key) {
Line 73         prev = currNode;
Line 74         currNode = currNode.getNext();
Line 75     }
Line 76
Line 77     if (currNode != null) {
Line 78         prev.setNext(currNode.getNext());
Line 79         System.out.println(key + " found and deleted");
Line 80     }
Line 81     if (currNode == null) {
Line 82         // Display the message
Line 83         System.out.println(key + " not found");
Line 84     }
Line 85     return list;
Line 86 }
Line 87
Line 88 public static LinkedList deleteAtPosition(LinkedList list, int index) {

```



```

Line 89     Node currNode = list.head, prev = null;
Line 90
Line 91     if (index == 0 && currNode != null) {
Line 92         list.head = currNode.getNext(); // Changed head
Line 93
Line 94         System.out.println(
Line 95             index + " position element deleted");
Line 96
Line 97         return list;
Line 98     }
Line 99
Line 100     while (currNode != null) {
Line 101
Line 102         if (counter == index) {
Line 103             prev.setNext(currNode.getNext());
Line 104
Line 105             System.out.println(
Line 106                 index + " position element deleted");
Line 107             break;
Line 108         }
Line 109         else {
Line 110             prev = currNode;
Line 111             currNode = currNode.getNext();
Line 112             counter++;
Line 113         }
Line 114     }
Line 115
Line 116     if (currNode == null) {
Line 117         // Display the message
Line 118         System.out.println(
Line 119             index + " position element not found");
Line 120     }
Line 121
Line 122     return list;
Line 123 }
Line 124
Line 125 public void swapNodes(String x, String y) {
Line 126     if (x == y)
Line 127         return;
Line 128
Line 129     Node prevX = null, currX = head;
Line 130     while (currX != null && !Objects.equals(currX.getData(), x)) {
Line 131         prevX = currX;
Line 132         currX = currX.getNext();
Line 133     }
Line 134
Line 135     Node prevY = null, currY = head;
Line 136     while (currY != null && !Objects.equals(currY.getData(), y)) {

```

Line 137	prevY = currY;
Line 138	currY = currY.getNext();
Line 139	}
Line 140	
Line 141	if (currX == null    currY == null)
Line 142	return;
Line 143	
Line 144	if (prevX != null)
Line 145	prevX.setNext(currY);
Line 146	else // make y the new head
Line 147	head = currY;
Line 148	
Line 149	if (prevY != null)
Line 150	prevY.setNext(currX);
Line 151	else // make x the new head
Line 152	head = currX;
Line 153	
Line 154	Node temp = currX.getNext();
Line 155	currX.setNext(currY.getNext());
Line 156	currY.setNext(temp);
Line 157	}
Line 158	
Line 159	public Node insertionSort(){
Line 160	Node dummy = new Node(null);
Line 161	Node curr = this.head;
Line 162	
Line 163	while (curr != null) {
Line 164	Node prev = dummy;
Line 165	
Line 166	while (prev.getNext() != null &&
Line 167	prev.getNext().getData().compareTo(curr.getData()) < 0) {
Line 168	prev = prev.getNext();
Line 169	}
Line 170	
Line 171	Node next = curr.getNext();
Line 172	curr.setNext(prev.getNext());
Line 173	prev.setNext(curr);
Line 174	
Line 175	curr = next;
Line 176	}
Line 177	
Line 178	return dummy.getNext();
Line 179	}
Line 180	}

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 = "";

Line 7	private boolean debuggingMode = true;
Line 8	private String[] inputWordList;
Line 9	private LinkedList[] linkedLists;
Line 10	private long startTime;
Line 11	
Line 12	private void printHeader(String methodName) {
Line 13	if (debuggingMode) {
Line 14	System.out.println("\n\t - - - - " + methodName + " - - - -");
Line 15	}
Line 16	}
Line 17	
Line 18	private void readFile() {
Line 19	printHeader("readFile");
Line 20	List<String> listOfStrings = new ArrayList<String>();
Line 21	BufferedReader bf;
Line 22	
Line 23	try {
Line 24	bf = new BufferedReader(new FileReader("src/"+inputFileName));
Line 25	} catch (FileNotFoundException e) {
Line 26	throw new RuntimeException(inputFileName + " not found, " + e.getMessage());
Line 27	}
Line 28	
Line 29	try {
Line 30	String line = bf.readLine();
Line 31	
Line 32	while (line != null) {
Line 33	listOfStrings.add(line);
Line 34	line = bf.readLine();
Line 35	}
Line 36	} catch (Exception e){
Line 37	throw new RuntimeException("Error reading file, exception caught " +
Line 38	e.getMessage());
Line 39	}
Line 40	
Line 41	try {
Line 42	bf.close();
Line 43	} catch (IOException e) {
Line 44	throw new RuntimeException("Error closing file, exception caught " +
Line 45	e.getMessage());
Line 46	}
Line 47	
Line 48	inputWordList = listOfStrings.toArray(new String[0]);
Line 49	
Line 50	if(debuggingMode) {
Line 51	System.out.println("\nPrint input file's unsorted contents:");
Line 52	int lineNumber = 1;
Line 53	for (String str : inputWordList) {
Line 54	System.out.println("Line " + String.valueOf(lineNumber) + ": " + str);

```

Line 55         lineNumber++;
Line 56     }
Line 57 }
Line 58 }
Line 59
Line 60 private boolean checkAnagram(String str1, String str2) {
Line 61     printHeader("checkAnagram");
Line 62
Line 63     if (str1.length() != str2.length())
Line 64         return false;
Line 65
Line 66     char[] a = str1.toCharArray();
Line 67     char[] b = str2.toCharArray();
Line 68
Line 69     return Arrays.equals(sortCharArray(a), sortCharArray(b));
Line 70 }
Line 71
Line 72 private char[] sortCharArray(char[] arr){
Line 73     char temp;
Line 74
Line 75     int i = 0;
Line 76     while (i < arr.length) {
Line 77         int j = i + 1;
Line 78         while (j < arr.length) {
Line 79             if (arr[j] < arr[i]) {
Line 80
Line 81                 temp = arr[i];
Line 82                 arr[i] = arr[j];
Line 83                 arr[j] = temp;
Line 84             }
Line 85             j += 1;
Line 86         }
Line 87         i += 1;
Line 88     }
Line 89
Line 90     return arr;
Line 91
Line 92 }
Line 93
Line 94 private void quickSortLinkedListArray(){
Line 95     printHeader("quickSortLinkedListArray");
Line 96     quickSortLinkedListArrayAlg(linkedLists, 0, linkedLists.length - 1);
Line 97
Line 98     if(debuggingMode) {
Line 99         System.out.println("\nPrint input file's sorted contents:");
Line 100         int lineNumber = 1;
Line 101         for (LinkedList list : linkedLists) {
Line 102

```

```

Line 103         System.out.println("Line " + String.valueOf(lineNumber) + ": " +
Line 104 list.getHead().getData());
Line 105         lineNumber++;
Line 106     }
Line 107 }
Line 108 }
Line 109
Line 110 private void quickSortLinkedListArrayAlg(LinkedList[] arr, int l, int h) {
Line 111     printHeader("quickSortLinkedListArrayAlg");
Line 112
Line 113     int[] stack = new int[h - l + 1];
Line 114     int top = -1;
Line 115     stack[++top] = l;
Line 116     stack[++top] = h;
Line 117
Line 118     while (top >= 0) {
Line 119         h = stack[top--];
Line 120         l = stack[top--];
Line 121         int p = partitionLinkedListArray(arr, l, h);
Line 122
Line 123         if (p - 1 > l) {
Line 124             stack[++top] = l;
Line 125             stack[++top] = p - 1;
Line 126         }
Line 127
Line 128         if (p + 1 < h) {
Line 129             stack[++top] = p + 1;
Line 130             stack[++top] = h;
Line 131         }
Line 132     }
Line 133 }
Line 134
Line 135 private int partitionLinkedListArray(LinkedList[] arr, int low, int high) {
Line 136     printHeader("partitionLinkedListArray");
Line 137
Line 138     LinkedList pivot = arr[high];
Line 139
Line 140     int i = (low - 1);
Line 141     for (int j = low; j <= high - 1; j++) {
Line 142         if (arr[j].getHead().getData().compareTo(pivot.getHead().getData()) < 0) {
Line 143             i++;
Line 144
Line 145             Node temp = arr[i].getHead();
Line 146             arr[i].setHead(arr[j].getHead());
Line 147             arr[j].setHead(temp);
Line 148         }
Line 149     }
Line 150 }

```

```

Line 151     Node temp = arr[i + 1].getHead();
Line 152     arr[i + 1].setHead(arr[high].getHead());
Line 153     arr[high].setHead(temp);
Line 154
Line 155     return i + 1;
Line 156 }
Line 157
Line 158 private void createLinkedListArray(){
Line 159     printHeader("createLinkedListArray");
Line 160
Line 161     linkedLists = new LinkedList[inputWordList.length];
Line 162
Line 163     linkedLists[0] = new LinkedList(inputWordList[0]);
Line 164
Line 165     if(debuggingMode){
Line 166         System.out.println("List after first position populated");
Line 167         printLinkedLists();
Line 168     }
Line 169
Line 170     for(int j = 1; j < inputWordList.length; j++){
Line 171         for(int i = 0; i < linkedLists.length; i++) {
Line 172             if (linkedLists[i] != null) {
Line 173                 if (checkAnagram(inputWordList[j], linkedLists[i].getHead().getData())) {
Line 174                     linkedLists[i].insert(inputWordList[j]);
Line 175                     if(debuggingMode){
Line 176                         System.out.println(inputWordList[j] + " is an anagram of "
Line 177                             + linkedLists[i].getHead().getData() + ", inserting into list" );
Line 178                     }
Line 179                     break;
Line 180                 } else if (debuggingMode){
Line 181                     System.out.println(inputWordList[j] + " is not an anagram of "
Line 182                         + linkedLists[i].getHead().getData());
Line 183                 }
Line 184             } else {
Line 185                 linkedLists[i] = new LinkedList(inputWordList[j]);
Line 186                 if(debuggingMode){
Line 187                     System.out.println("Didn't find list with " + inputWordList[j] + ", making new
Line 188 list");
Line 189                 }
Line 190                 break;
Line 191             }
Line 192         }
Line 193     }
Line 194
Line 195     if(debuggingMode){
Line 196         System.out.println("List before insertion sort");
Line 197         printLinkedLists();
Line 198     }

```

```

Line 199
Line 200     for (LinkedList linkedList : linkedLists) {
Line 201         if (linkedList != null) {
Line 202             linkedList.setHead(linkedList.insertionSort());
Line 203         }
Line 204     }
Line 205
Line 206     if(debuggingMode){
Line 207         System.out.println("List after insertion sort");
Line 208         printLinkedLists();
Line 209     }
Line 210
Line 211
Line 212 }
Line 213
Line 214 private void printLinkedLists() {
Line 215     for (LinkedList s : linkedLists) {
Line 216         if (s != null) {
Line 217             if (s.getHead() != null) {
Line 218                 s.printList();
Line 219             } else {
Line 220                 System.out.println("null");
Line 221             }
Line 222         } else {
Line 223             System.out.println("nullptr");
Line 224         }
Line 225     }
Line 226 }
Line 227
Line 228 private void shrinkLinkedListArray(){
Line 229     printHeader("shrinkLinkedListArray");
Line 230
Line 231     if(debuggingMode){
Line 232         System.out.println("List with nulls");
Line 233         printLinkedLists();
Line 234     }
Line 235
Line 236     ArrayList<LinkedList> list = new ArrayList<LinkedList>();
Line 237     for (LinkedList s : linkedLists)
Line 238         if (s != null)
Line 239             list.add(s);
Line 240     linkedLists = list.toArray(new LinkedList[list.size()]);
Line 241
Line 242
Line 243     if(debuggingMode){
Line 244         System.out.println("List without nulls");
Line 245         printLinkedLists();
Line 246     }

```



```

Line 247     }
Line 248
Line 249     private void printFile(){
Line 250         printHeader("printFile");
Line 251
Line 252         try {
Line 253             PrintWriter myFile = new PrintWriter(outputFileName + ".txt", "UTF-8");
Line 254             for (LinkedList list : linkedLists) {
Line 255                 if(debuggingMode){
Line 256                     System.out.println("Printing to file: " + list.toString());
Line 257                 }
Line 258                 myFile.println(list.toString());
Line 259             }
Line 260             myFile.close();
Line 261         } catch (IOException e) {
Line 262             System.out.println("An error occurred.");
Line 263             e.printStackTrace();
Line 264         }
Line 265     }
Line 266
Line 267     public void startOfProgram(){
Line 268         startTime = System.nanoTime();
Line 269     }
Line 270
Line 271     public void endOfProgram(){
Line 272         System.out.println(System.nanoTime() - startTime);
Line 273     }
Line 274
Line 275     public Assign2(String[] args) {
Line 276         startOfProgram();
Line 277
Line 278         if(args.length < 2){
Line 279             throw new IllegalArgumentException("Wrong number of input arguments: too few
Line 280 arguments");
Line 281         } else if(args.length > 3){
Line 282             throw new IllegalArgumentException("Wrong number of input arguments: too
Line 283 many arguments");
Line 284         }
Line 285
Line 286         if(args.length == 3){
Line 287             try{
Line 288                 Integer.parseInt(args[2]);
Line 289             } catch (NumberFormatException e){
Line 290                 throw new IllegalArgumentException("Debugging mode argument is not a
Line 291 number");
Line 292             }
Line 293
Line 294             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](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/](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](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/>