|  |  |
| --- | --- |
| **CPSC 319**  **Assignment 2**   |  | | --- | | Aleksander Berezowski  Tutorial Section: T03  TA Name: Shopon  Email: aleksander.berezowsk@ucalgary.ca | |

Contents

[Figures 2](#_Toc95579201)

[Tables 2](#_Toc95579202)

[Equations 2](#_Toc95579203)

[Data collected 3](#_Toc95579204)

[Question 1 4](#_Toc95579205)

[Question 2 6](#_Toc95579206)

[Theoretical Analysis 6](#_Toc95579207)

[Node.java 6](#_Toc95579208)

[LinkedList.java 7](#_Toc95579209)

[Assign2.java 11](#_Toc95579210)

[Conclusion 19](#_Toc95579211)

[Experimental Analysis 20](#_Toc95579212)

[Conclusion 20](#_Toc95579213)

[Bibliography 21](#_Toc95579214)

# Figures

[Figure 1 - Program Output 3](#_Toc95576120)

[Figure 2 - Run Time Graph 3](#_Toc95576121)

# Tables

[Table 1 - Program Run Time 3](#_Toc95576156)

# Equations

[Equation 1 - Trendline Equation from Figure 2 3](#_Toc95576597)

# 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.



Table 1 - Program Run Time

|  |  |
| --- | --- |
| Graphical user interface, text, application  Description automatically generated  Figure 1 - Program Output | Chart, line chart  Description automatically generated  Figure 2 - Run Time Graph |

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  Line 2  Line 3  Line 4  Line 5  Line 6  Line 7  Line 8  Line 9  Line 10  Line 11  Line 12  Line 13  Line 14  Line 15  Line 16  Line 17  Line 18  Line 19  Line 20 | 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;  } | 1  2  k  2  k  3  2  2  2  1  1  1 |

Mathematically:

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(k2)*, which confirms the mathematical proof above.

|  |  |  |
| --- | --- | --- |
| Line 1  Line 2  Line 3  Line 4  Line 5  Line 6  Line 7  Line 8  Line 9 | private boolean checkAnagram(String str1, String str2) {  if (str1.length() != str2.length())  return false;  char[] a = str1.toCharArray();  char[] b = str2.toCharArray();  return Arrays.equals(sortCharArray(a), sortCharArray(b));  } | 3  1  2  2  1 + 2(4+3k+10k2) |

Mathematically:

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(k2)*, 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(k2). 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  Line 2  Line 3  Line 4  Line 5  Line 6  Line 7  Line 8  Line 9  Line 10  Line 11  Line 12  Line 13  Line 14  Line 15  Line 16  Line 17 | 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

|  |  |
| --- | --- |
| Line 1  Line 2  Line 3  Line 4  Line 5  Line 6  Line 7  Line 8  Line 9  Line 10  Line 11  Line 12  Line 13  Line 14  Line 15  Line 16  Line 17  Line 18  Line 19  Line 20  Line 21  Line 22  Line 23  Line 24  Line 25  Line 26  Line 27  Line 28  Line 29  Line 30  Line 31  Line 32  Line 33  Line 34  Line 35  Line 36  Line 37  Line 38  Line 39  Line 40  Line 41  Line 42  Line 43  Line 44  Line 45  Line 46  Line 47  Line 48  Line 49  Line 50  Line 51  Line 52  Line 53  Line 54  Line 55  Line 56  Line 57  Line 58  Line 59  Line 60  Line 61  Line 62  Line 63  Line 64  Line 65  Line 66  Line 67  Line 68  Line 69  Line 70  Line 71  Line 72  Line 73  Line 74  Line 75  Line 76  Line 77  Line 78  Line 79  Line 80  Line 81  Line 82  Line 83  Line 84  Line 85  Line 86  Line 87  Line 88  Line 89  Line 90  Line 91  Line 92  Line 93  Line 94  Line 95  Line 96  Line 97  Line 98  Line 99  Line 100  Line 101  Line 102  Line 103  Line 104  Line 105  Line 106  Line 107  Line 108  Line 109  Line 110  Line 111  Line 112  Line 113  Line 114  Line 115  Line 116  Line 117  Line 118  Line 119  Line 120  Line 121  Line 122  Line 123  Line 124  Line 125  Line 126  Line 127  Line 128  Line 129  Line 130  Line 131  Line 132  Line 133  Line 134  Line 135  Line 136  Line 137  Line 138  Line 139  Line 140  Line 141  Line 142  Line 143  Line 144  Line 145  Line 146  Line 147  Line 148  Line 149  Line 150  Line 151  Line 152  Line 153  Line 154  Line 155  Line 156  Line 157  Line 158  Line 159  Line 160  Line 161  Line 162  Line 163  Line 164  Line 165  Line 166  Line 167  Line 168  Line 169  Line 170  Line 171  Line 172  Line 173  Line 174  Line 175  Line 176  Line 177  Line 178  Line 179  Line 180 | 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(n2)*

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(n2)* big-O notation.

### Assign2.java

|  |  |
| --- | --- |
| Line 1  Line 2  Line 3  Line 4  Line 5  Line 6  Line 7  Line 8  Line 9  Line 10  Line 11  Line 12  Line 13  Line 14  Line 15  Line 16  Line 17  Line 18  Line 19  Line 20  Line 21  Line 22  Line 23  Line 24  Line 25  Line 26  Line 27  Line 28  Line 29  Line 30  Line 31  Line 32  Line 33  Line 34  Line 35  Line 36  Line 37  Line 38  Line 39  Line 40  Line 41  Line 42  Line 43  Line 44  Line 45  Line 46  Line 47  Line 48  Line 49  Line 50  Line 51  Line 52  Line 53  Line 54  Line 55  Line 56  Line 57  Line 58  Line 59  Line 60  Line 61  Line 62  Line 63  Line 64  Line 65  Line 66  Line 67  Line 68  Line 69  Line 70  Line 71  Line 72  Line 73  Line 74  Line 75  Line 76  Line 77  Line 78  Line 79  Line 80  Line 81  Line 82  Line 83  Line 84  Line 85  Line 86  Line 87  Line 88  Line 89  Line 90  Line 91  Line 92  Line 93  Line 94  Line 95  Line 96  Line 97  Line 98  Line 99  Line 100  Line 101  Line 102  Line 103  Line 104  Line 105  Line 106  Line 107  Line 108  Line 109  Line 110  Line 111  Line 112  Line 113  Line 114  Line 115  Line 116  Line 117  Line 118  Line 119  Line 120  Line 121  Line 122  Line 123  Line 124  Line 125  Line 126  Line 127  Line 128  Line 129  Line 130  Line 131  Line 132  Line 133  Line 134  Line 135  Line 136  Line 137  Line 138  Line 139  Line 140  Line 141  Line 142  Line 143  Line 144  Line 145  Line 146  Line 147  Line 148  Line 149  Line 150  Line 151  Line 152  Line 153  Line 154  Line 155  Line 156  Line 157  Line 158  Line 159  Line 160  Line 161  Line 162  Line 163  Line 164  Line 165  Line 166  Line 167  Line 168  Line 169  Line 170  Line 171  Line 172  Line 173  Line 174  Line 175  Line 176  Line 177  Line 178  Line 179  Line 180  Line 181  Line 182  Line 183  Line 184  Line 185  Line 186  Line 187  Line 188  Line 189  Line 190  Line 191  Line 192  Line 193  Line 194  Line 195  Line 196  Line 197  Line 198  Line 199  Line 200  Line 201  Line 202  Line 203  Line 204  Line 205  Line 206  Line 207  Line 208  Line 209  Line 210  Line 211  Line 212  Line 213  Line 214  Line 215  Line 216  Line 217  Line 218  Line 219  Line 220  Line 221  Line 222  Line 223  Line 224  Line 225  Line 226  Line 227  Line 228  Line 229  Line 230  Line 231  Line 232  Line 233  Line 234  Line 235  Line 236  Line 237  Line 238  Line 239  Line 240  Line 241  Line 242  Line 243  Line 244  Line 245  Line 246  Line 247  Line 248  Line 249  Line 250  Line 251  Line 252  Line 253  Line 254  Line 255  Line 256  Line 257  Line 258  Line 259  Line 260  Line 261  Line 262  Line 263  Line 264  Line 265  Line 266  Line 267  Line 268  Line 269  Line 270  Line 271  Line 272  Line 273  Line 274  Line 275  Line 276  Line 277  Line 278  Line 279  Line 280  Line 281  Line 282  Line 283  Line 284  Line 285  Line 286  Line 287  Line 288  Line 289  Line 290  Line 291  Line 292  Line 293  Line 294  Line 295  Line 296  Line 297  Line 298  Line 299  Line 300  Line 301  Line 302  Line 303  Line 304  Line 305  Line 306  Line 307  Line 308  Line 309  Line 310  Line 311  Line 312  Line 313  Line 314  Line 315  Line 316  Line 317  Line 318  Line 319 | import java.io.\*;  import java.util.\*;  public class Assign2 {  private String inputFileName = "";  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){  debuggingMode = false;  } else if (Integer.parseInt(args[2]) != 1){  throw new IllegalArgumentException("Debugging mode argument is not 0 or 1");  }  } else {  debuggingMode = false;  }  inputFileName = args[0] + ".txt";  outputFileName = args[1];  readFile();  createLinkedListArray();  shrinkLinkedListArray();  quickSortLinkedListArray();  printFile();  endOfProgram();  }  public static void main(String[] args){  new Assign2(args);  }  } |

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(l2)*

As shown in Question 1, the worst-case complexity of this algorithm when checking if two words are anagrams of each other is O(l2).

quickSortLinkedListArray: *O(n2)*

This method uses *quickSortLinkedListArrayAlg* and *partitionLinkedListArray* as helper methods, and has a worst-case of *O(n2)* as proven in Assignment 1.

createLinkedListArray: *O(n3 l2)*

This method has 2 *for* loops on line 169 and line 170, each of *O(n)* complexity with a combined big-O of *O(n2)*. Then on line 172 there is a function call for *checkAnagram* which has been previously shown to be *O(l2)*; multiplying *O(n2)* by *O(l2)* gives a time complexity of *O(n2 l2)*. 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(n3 l2)*.

printLinkedLists: *O(n2)*

This method has a big-O notation of *O(n2)* 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(n2)*.

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(n2)*

This method has the *toString* method nested inside a *for* loop. Both are *O(n)* time complexity, therefore nesting them results in a *O(n2)* 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(n3 l2)*

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(n3 l2)*.

main: *O(n3 l2)*

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(n3 l2)*, 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(n3 l2)*, 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 *n3*.

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(n3)*, 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(l2)*.

## 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(n3 l2)*. However, the experimental analysis can only confirm the big-O run time to be at least *O(n3)*. 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(n3)*, although it is possible to assume the actual time complexity is *O(n3 l2)*. 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/