Object Oriented Java Programming Report

Name: Tam Kevin Lok Sun
Student Number: 1742270

Module: CM1210
Word Count: 1539

Introduction

This report comprises of two parts, Algorithms and Data Structures. For algorithms, descriptions of both Bubble and Merge sort algorithms will be provided, alongside implementation of both sorting methods in my java program. Results of runtime, number of moves and number of swaps when the two sorting methods are ran will also be shown. For data structures, a description of how I have implemented the two MyLinkedList methods can be expected. In addition, a user guide will be provided for both programs at the end of the report, alongside a brief conclusion.

Algorithms

Description of Bubble Sort

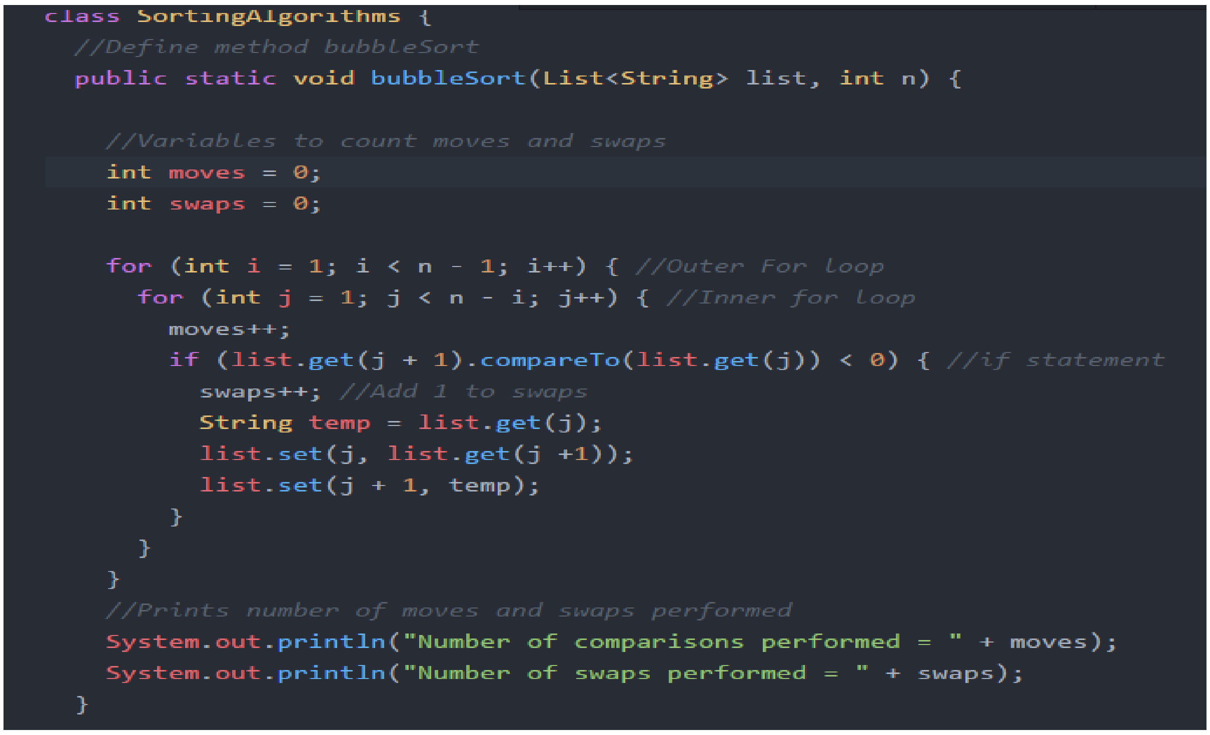
Bubble sort is a simple comparison-based algorithm where adjacent elements are compared and swapped if they are not in order. For each pass, the algorithm compares element 1 with element 2 and swaps it if necessary and does the same to element 2 and 3 on the next pass. Therefore, the array would be sorted after n – 1 passes, where n is the number of elements in the array. This sorting method is notorious for being one of the least efficient ways of sorting data because it doesn't ignore elements that are already sorted, having an average and worst case performance of O(n^2). Thus, this sorting method is unsuitable for large or even medium sized arrays and is rarely used.

Description of Merge Sort

Merge sort is an efficient comparison-based algorithm which utilizes a divide and conquer approach to sorting data. The algorithm takes a sequence, s and splits it, creating 2 sequences, S1 and S2, each containing half of the elements in S. S1 and S2 is then sorted by recursively calling a sort algorithm, which is then combined by putting elements from the now sorted S1 and S2 sequences into S, resulting in a sorted sequence. With a worst case performance of O(n log n), it is highly efficient, thus suitable for very large sequences. Merge sort is also a stable sorting method, meaning it preserves the relative order of keys in the input data set.

Implementation into program: SortingAlgorithms

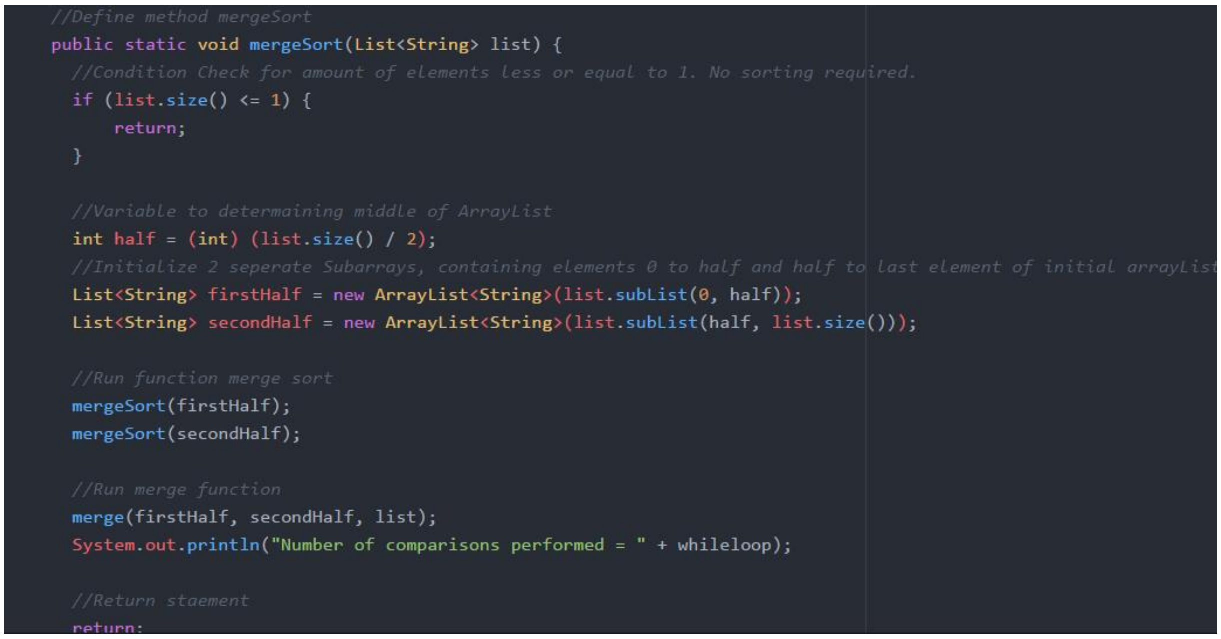
Bubble Sort



```java
class SortingAlgorithms {
  //Define method bubbleSort
  public static void bubbleSort(List<String> list, int n) {

    //Variables to count moves and swaps
    int moves = 0;
    int swaps = 0;

    for (int i = 1; i < n - 1; i++) { //Outer For Loop
      for (int j = 1; j < n - i; j++) { //Inner for loop
        moves++;
        if (list.get(j + 1).compareTo(list.get(j)) < 0) { //if statement
          swaps++; //Add 1 to swaps
          String temp = list.get(j);
          list.set(j, list.get(j +1));
          list.set(j + 1, temp);
        }
      }
    }
    //Prints number of moves and swaps performed
    System.out.println("Number of comparisons performed = " + moves);
    System.out.println("Number of swaps performed = " + swaps);
  }
```

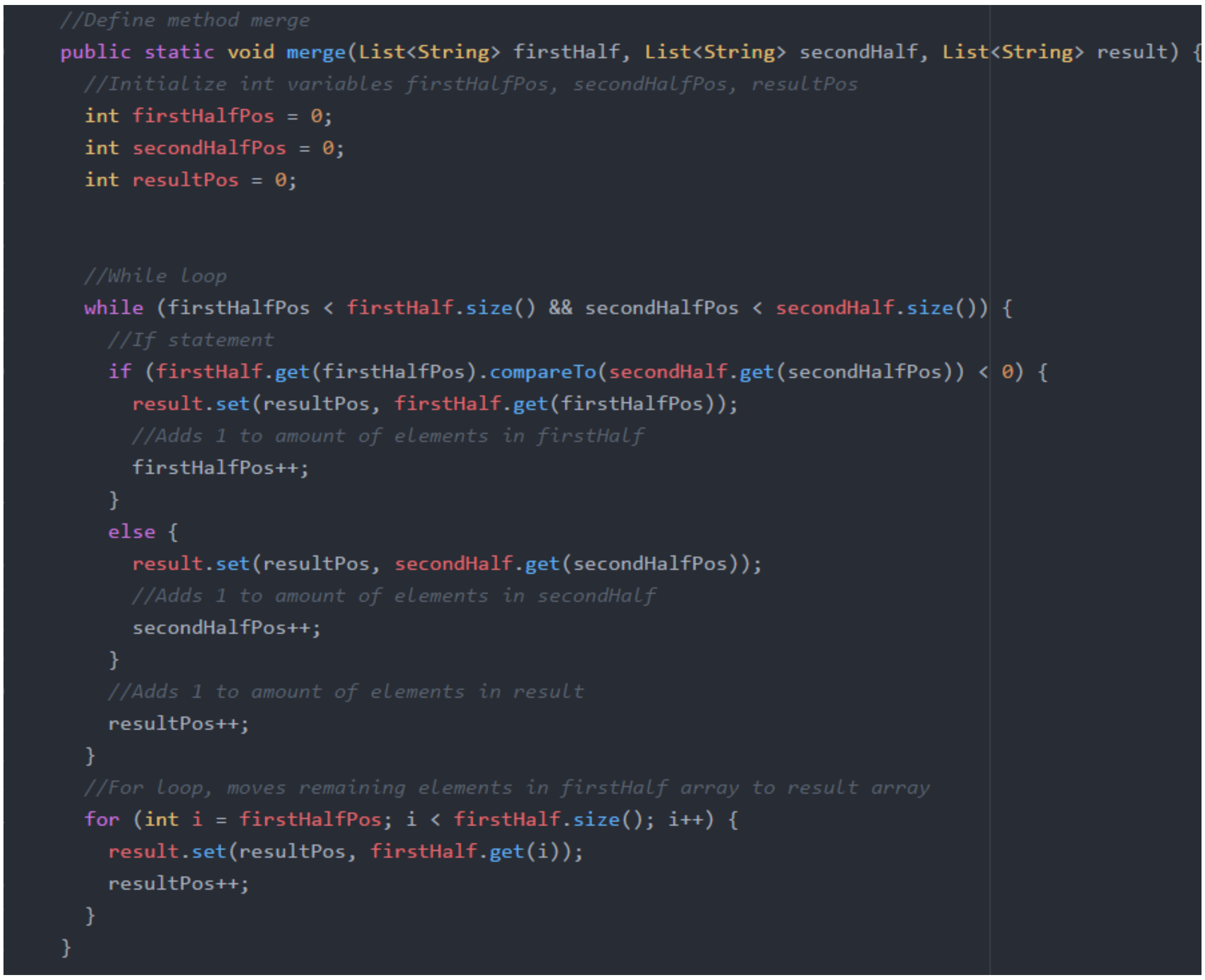
The class is defined as SortingAlgorithms and Bubble Sort is implemented via a method called bubbleSort, which takes the ArrayList list and an integer n, which is passed as the length of the array when the main function is called, discussed later. Integer variables are defined to count moves and swaps. An outer for loop nests an inner for loop, which encases an if statement which compares the 2 adjacent elements and swaps them if the condition is met. This would count as 1 swap and thus the number of swaps is added by 1. The end prints out the number of comparisons and swaps performed during the sorting process.

Merge Sort



```java
  //Define method mergeSort
  public static void mergeSort(List<String> list) {
    //Condition Check for amount of elements less or equal to 1. No sorting required.
    if (list.size() <= 1) {
        return;
    }

    //Variable to determaining middle of ArrayList
    int half = (int) (list.size() / 2);
    //Initialize 2 seperate Subarrays, containing elements 0 to half and half to last element of initial arrayList
    List<String> firstHalf = new ArrayList<String>(list.subList(0, half));
    List<String> secondHalf = new ArrayList<String>(list.subList(half, list.size()));

    //Run function merge sort
    mergeSort(firstHalf);
    mergeSort(secondHalf);

    //Run merge function
    merge(firstHalf, secondHalf, list);
    System.out.println("Number of comparisons performed = " + whileloop);

    //Return staement
    return;
```

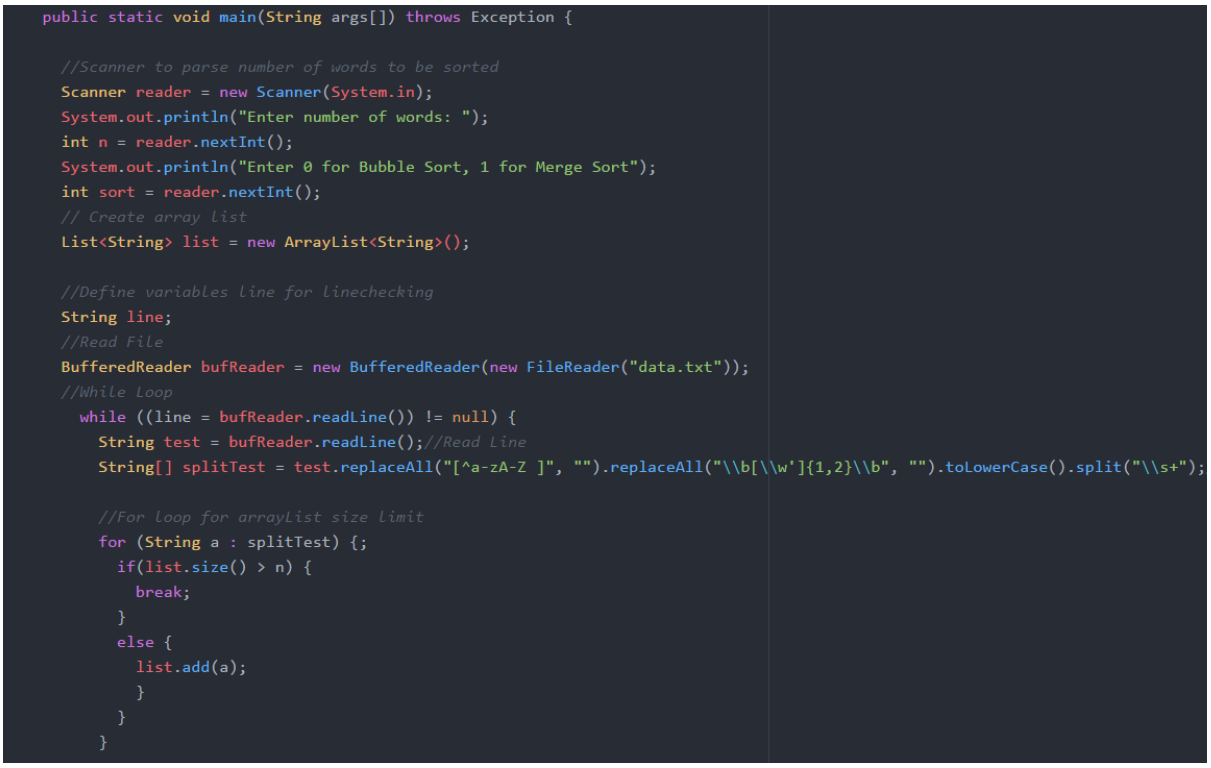
Merge sorting is implemented in two parts. First, the method mergeSort is defined, which takes an ArrayList. An if condition checks whether the array requires sorting, which if it contains less or equal to 1 element, the function returns and nothing would be done. Else, an integer called half is declared, which would signify the mid point of the array. This variable is used to create two sub-arrays, firstHalf and secondHalf, which houses elements from index 0 – half and half – the size of array, essentially splitting the original array. This function is then called recursively on both arrays until both is sorted, at which point the merge function, discussed next, would be called to combine the two arrays into one. When complete, function returns.



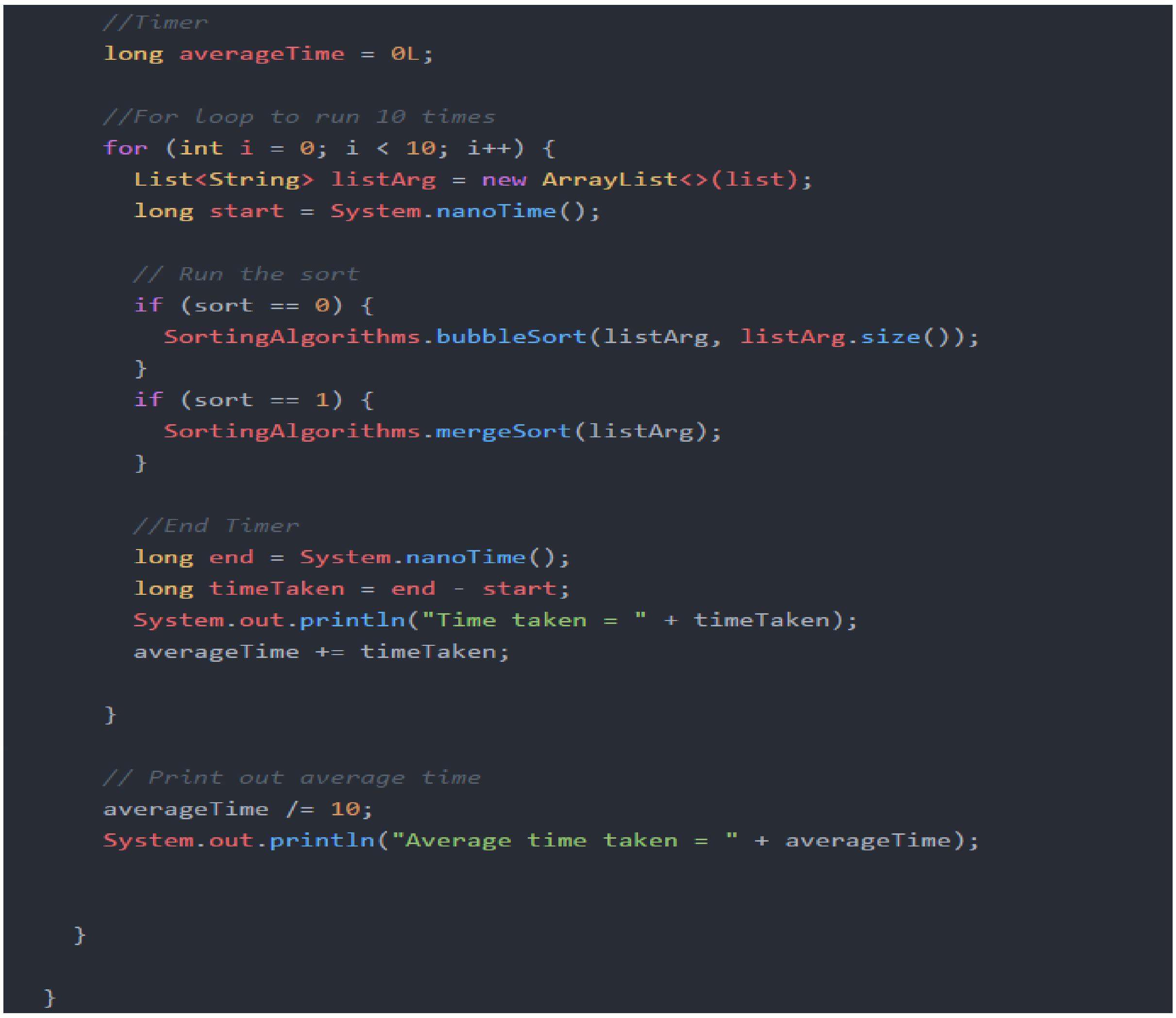
```
//Define method merge
public static void merge(List<String> firstHalf, List<String> secondHalf, List<String> result) {
  //Initialize int variables firstHalfPos, secondHalfPos, resultPos
  int firstHalfPos = 0;
  int secondHalfPos = 0;
  int resultPos = 0;


  //While loop
  while (firstHalfPos < firstHalf.size() && secondHalfPos < secondHalf.size()) {
    //If statement
    if (firstHalf.get(firstHalfPos).compareTo(secondHalf.get(secondHalfPos)) < 0) {
      result.set(resultPos, firstHalf.get(firstHalfPos));
      //Adds 1 to amount of elements in firstHalf
      firstHalfPos++;
    }
    else {
      result.set(resultPos, secondHalf.get(secondHalfPos));
      //Adds 1 to amount of elements in secondHalf
      secondHalfPos++;
    }
    //Adds 1 to amount of elements in result
    resultPos++;
  }
  //For loop, moves remaining elements in firstHalf array to result array
  for (int i = firstHalfPos; i < firstHalf.size(); i++) {
    result.set(resultPos, firstHalf.get(i));
    resultPos++;
  }
}
```

The second part of Merge Sort implementation is the merging. A method, merge is defined, taking arrays firstHalf, secondHalf and result. 3 variables are initialized to determine position of elements, each corresponding to their respectively named arrays. A while loop executes providing that both firstHalf and secondHalf arrays are not empty. An if statement executes if the next element in firstHalf is less or equal than secondHalf and moves the next element of firstHalf into the result array and vice versa if it is false. firstHalfPos and secondHalfPos are added respectively to signify an occupied position in their respective arrays. Finally, a for loop moves all remaining elements from firstHalf to results and merging is complete.

Main Function



```java
  public static void main(String args[]) throws Exception {

    //Scanner to parse number of words to be sorted
    Scanner reader = new Scanner(System.in);
    System.out.println("Enter number of words: ");
    int n = reader.nextInt();
    System.out.println("Enter 0 for Bubble Sort, 1 for Merge Sort");
    int sort = reader.nextInt();
    // Create array list
    List<String> list = new ArrayList<String>();

    //Define variables line for linechecking
    String line;
    //Read File
    BufferedReader bufReader = new BufferedReader(new FileReader("data.txt"));
    //While Loop
      while ((line = bufReader.readLine()) != null) {
        String test = bufReader.readLine();//Read Line
        String[] splitTest = test.replaceAll("[^a-zA-Z ]", "").replaceAll("\\b[\\w']{1,2}\\b", "").toLowerCase().split("\\s+");

        //For loop for arrayList size limit
        for (String a : splitTest) {;
          if(list.size() > n) {
            break;
          }
          else {
            list.add(a);
            }
          }
        }
```



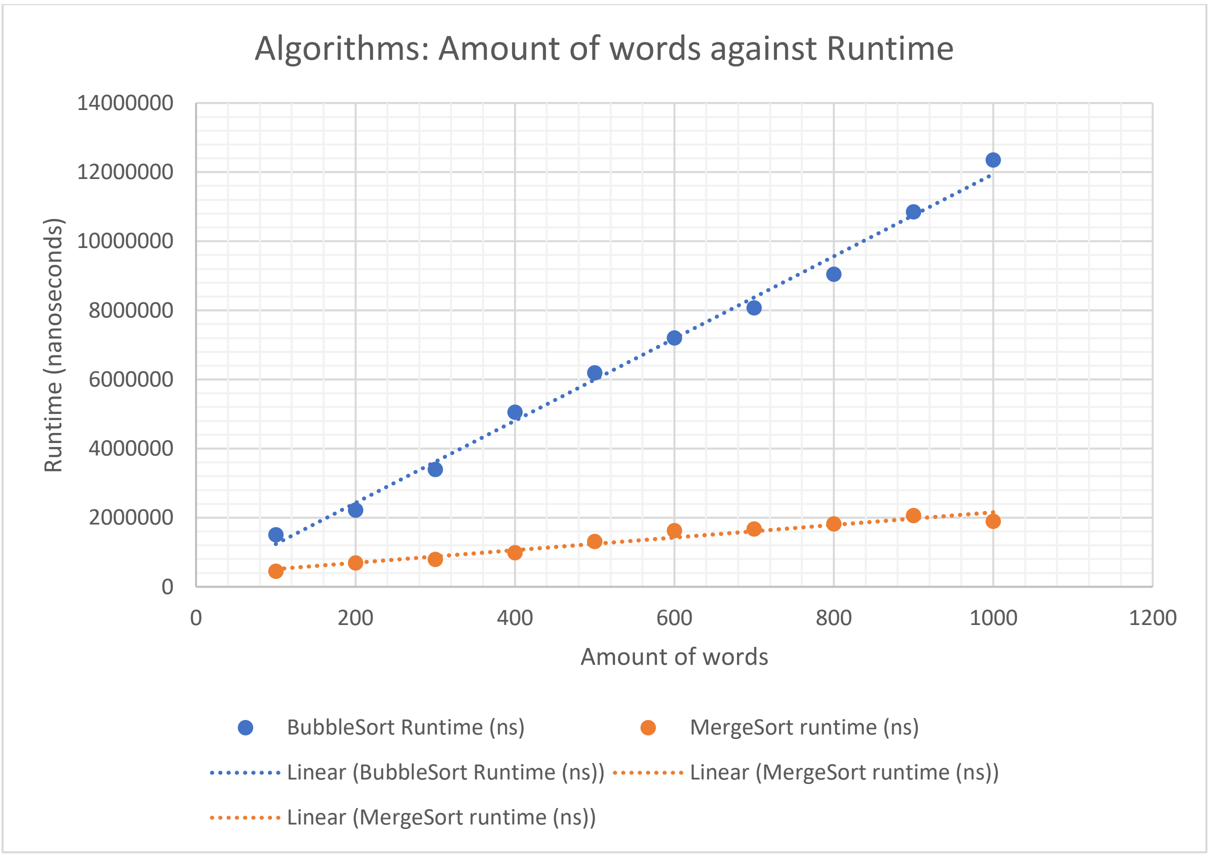
```java
      //Timer
      long averageTime = 0L;

      //For loop to run 10 times
      for (int i = 0; i < 10; i++) {
        List<String> listArg = new ArrayList<>(list);
        long start = System.nanoTime();

        // Run the sort
        if (sort == 0) {
          SortingAlgorithms.bubbleSort(listArg, listArg.size());
        }
        if (sort == 1) {
          SortingAlgorithms.mergeSort(listArg);
        }

        //End Timer
        long end = System.nanoTime();
        long timeTaken = end - start;
        System.out.println("Time taken = " + timeTaken);
        averageTime += timeTaken;

      }

      // Print out average time
      averageTime /= 10;
      System.out.println("Average time taken = " + averageTime);


    }

  }
```

The main method runs the program. The main method is defined, at which a scanner is initialized to take the number of words and method of sorting desired. Variable integer n and sort is initialized for these purposes. A new ArrayList is then created and a BufferedReader nests a FileReader which reads the file. A while loop is executed which reads the file line by line providing it is not empty, puts it into a string called test and is then placed into an array, declared splitTest, which utilizes regex to remove all non-alphabetical symbols, removes any terms which contain less than 3 characters, turns them all into lower case and splits it. A for loop is then used to limit the amount of words added to the ArrayList, denoted by the user input variable n, declared at the beginning of the program.

A for loop is then used to execute the sorting algorithms 10 times in order to get an average result, which a new ArrayList called listArg is defined, having elements passed from ArrayList list. A long variable is created called start to signify the start of execution, where depending on the number entered in the beginning, would sort the array using bubble sort, or merge sort. When sorting is complete, long variable end is declared to signify the end of execution. Time taken is calculated by subtracting end with start and result is added to averageTime. averageTime is then divided by 10 to get the average time taken, at which point is printed in the console.

Results of runtime, number of moves and swaps of algorithms

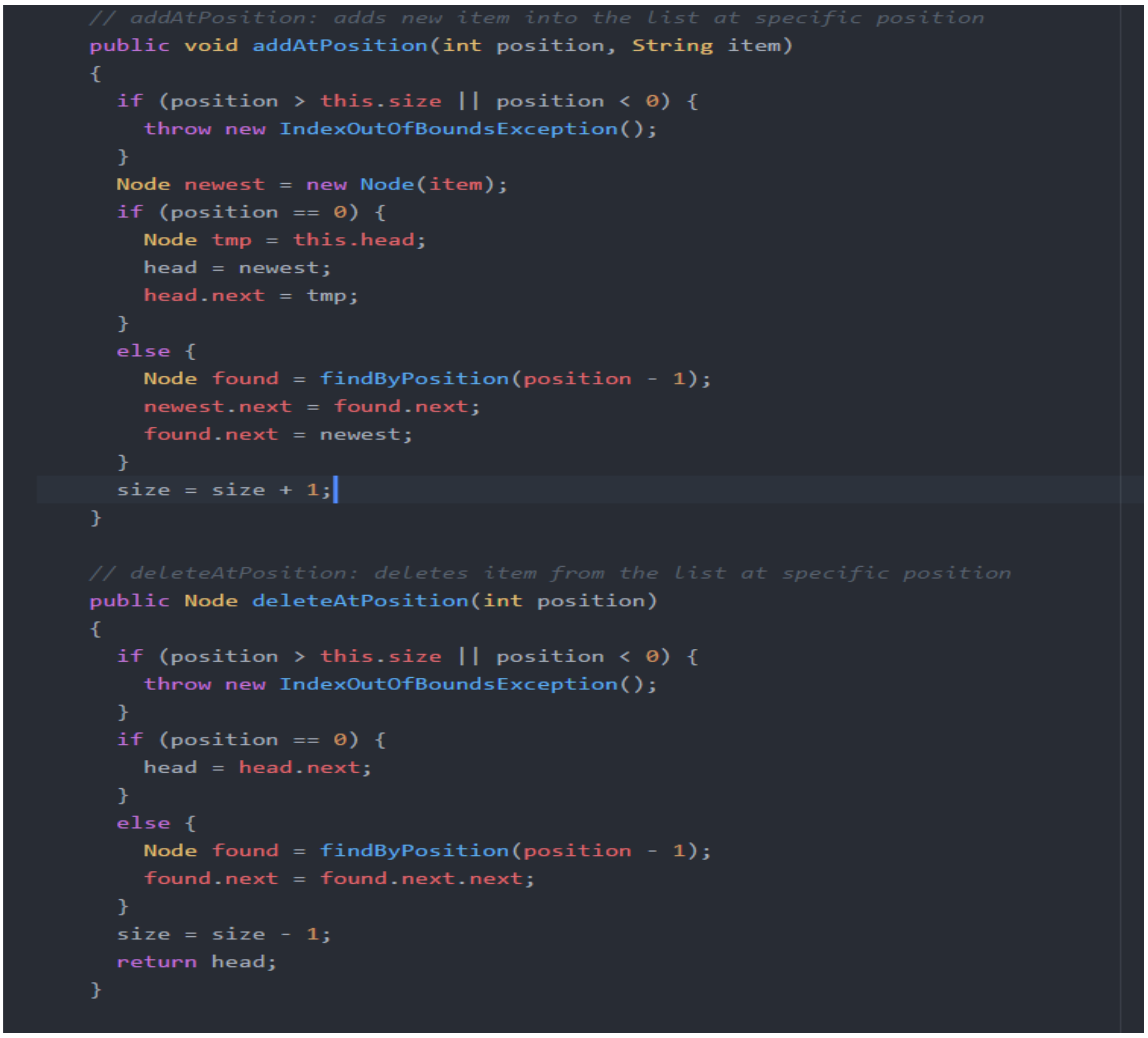
Words	BubbleSort Runtime (ns)	MergeSort runtime (ns)	Comparisons BubbleSort	Swaps BubbleSort
100	1508740	453491	4950	2306
200	2221747	697323	19900	8694
300	3399821	792651	44850	20063
400	5053509	993816	79800	38577
500	6197489	1313382	124750	64104
600	7207187	1626586	179700	93347
700	8071976	1671189	244650	125726
800	9044934	1826606	319600	162935
900	10851235	2066646	404550	201916
1000	12351639	1901589	499500	254003



As seen from the runtimes of both algorithms, results are as expected. Bubble sort, having an average case performance of O(n^2), shows a significantly longer runtime than Merge Sort, which only has an worst case performance of O(n log n). Also noted is the linear increase of runtimes for both algorithms as number of words increased.

Data Structures

Implementation of MyLinkedList method



```java
    // addAtPosition: adds new item into the list at specific position
    public void addAtPosition(int position, String item)
    {
      if (position > this.size || position < 0) {
        throw new IndexOutOfBoundsException();
      }
      Node newest = new Node(item);
      if (position == 0) {
        Node tmp = this.head;
        head = newest;
        head.next = tmp;
      }
      else {
        Node found = findByPosition(position - 1);
        newest.next = found.next;
        found.next = newest;
      }
      size = size + 1;
    }

    // deleteAtPosition: deletes item from the list at specific position
    public Node deleteAtPosition(int position)
    {
      if (position > this.size || position < 0) {
        throw new IndexOutOfBoundsException();
      }
      if (position == 0) {
        head = head.next;
      }
      else {
        Node found = findByPosition(position - 1);
        found.next = found.next.next;
      }
      size = size - 1;
      return head;
    }
```

Implementation of MyLinkedList class is achieved by completing the two methods, addAtPosition and deleteAtPosition. addAtPosition takes an integer position and a string called item. Error checking is provided by an if statement. Else, it creates a new node, references head to tmp, points head to newest and point head.next to tmp if adding into the first position. Else, it addes element at position and links newest element to the next element. The size of the array is then increased by one.

The process is also similar in deleteAtPosition. An if statement again checks for index. If node is in the first position, it deletes the node by linking head to the next node. For any other nodes, the position of the previous node to be deleted is linked to the next reference to the found.next.next node. The node head is then returned to show which node was deleted.

User Guide for Sorting Algorithms

Running the algorithms is simple. First, execute the program, at which point you would be prompted with entering the number of words. Enter any number at this point and choose which sorting algorithm to utilize by typing 0 for Bubble Sort and 1 for Merge Sort. The program would then execute and end when the sorting is complete.

Conclusion

In conclusion, the report reinforces the importance of searching performance and shows how Merge Sort is a much more efficient method of sorting, even for such a small sample size of 1000 words. In addition, the importance of integration of all methods into a single program is also highlighted, as I did not create a separate method called parseData(), instead combining it with the two sorting algorithms to make the program as compact as possible.