CM1210: Data Structures and Algorithms in Java

Question 1:

This method takes an ArrayList of *input* words and *stopwords* and returns an ArrayList with *stopwords* removed.

My method can be called by constructing a new sortAssignment object and calling the method *deleteStopwords* and passing in two ArrayLists as arguments.

```
//Constructor
public sortAssignment(){}

//Removes stopwords from list input
public static ArrayList deleteStopwords(ArrayList input, ArrayList stopwords){
    ArrayList newList = new ArrayList<String>(input);
    for (int i = 0; i < newList.size(); i++){                //Iterates through
        String index = newList.get(i).toString();
        if (stopwords.contains(index)){
            //System.out.println("Deleting" + index);
            newList.remove(index);                           //Removes any word
        }
    }
    return newList;
}
```

Figure 1: Constructor and deleteStopwords

When called this method duplicates *input* into a new ArrayList and iterates through the duplicate list, removing any value that is contained in *stopwords*. This method is very simple and the only way I could think of to improve efficiency is to not have a temporary list as this may impact performance with larger lists.

```
//Method to make it easier to convert text files into ArrayLists
public static ArrayList wordReader(String file) throws FileNotFoundException{
    Scanner rdr = new Scanner(new FileReader(file));
    ArrayList<String> inputList = new ArrayList<>();
    rdr.useDelimiter("[^A-Za-z']+");
    while (rdr.hasNext()) {
        String word = rdr.next();
        inputList.add(word);
    }
    rdr.close();
    return inputList;
}
```

Figure 2: wordReader

I also created the *wordReader* method to convert text files to ArrayLists as we were given the *input* and *stopwords* list as text files. It splits the text file into words, removing unnecessary punctuation.

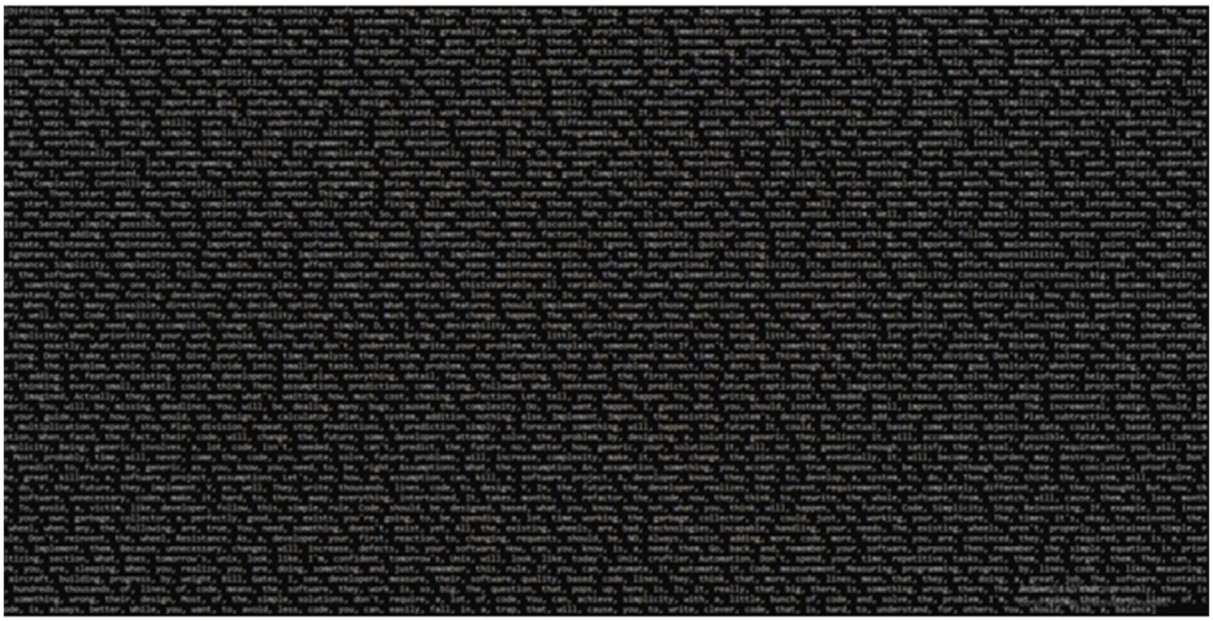

Figure 3: deleteStopwords output

Question 2:

I used pseudocode given in the lecture for my insertion sort. The method receives an ArrayList as an argument and returns a sorted list.

```
public static ArrayList insertionSort(ArrayList listOfWords){
    ArrayList newList = new ArrayList<String>(listOfWords);
    for (int i = 1; i < newList.size(); i++){
        String item = newList.get(i).toString().toLowerCase();
        int j = (i-1);
        while (j >=0 && newList.get(j).toString().toLowerCase().compareTo(item) > 0){
            newList.set(j+1, newList.get(j).toString().toLowerCase());
            //System.out.println("Swap");
            swaps++;
            j = j-1;
        }
        newList.set(j+1, item);
        //System.out.println("Swap");
        swaps++;
    }
    return newList;
}
```

Figure 5: insertionSort

The method creates a duplicate list, it then iterates through this list, comparing the previous item in the list to the current iterated item. While the previous item's ASCII value is bigger (further along in the alphabet), the previous item is set to the current item and the previous item index moves down by one for each cycle, otherwise it is kept in the same place. Another part worth mentioning is I need to pass each ArrayList item through the *toString*() and *toLowerCase*() methods to compare the ASCII values, as ArrayLists are their own type and need to be converted to a string to have ASCII values, also ASCII characters in uppercase have higher values so cannot be compared. The *compareTo*() method returns a positive or negative integer representing the difference between the ASCII values of the argument string and the main string.

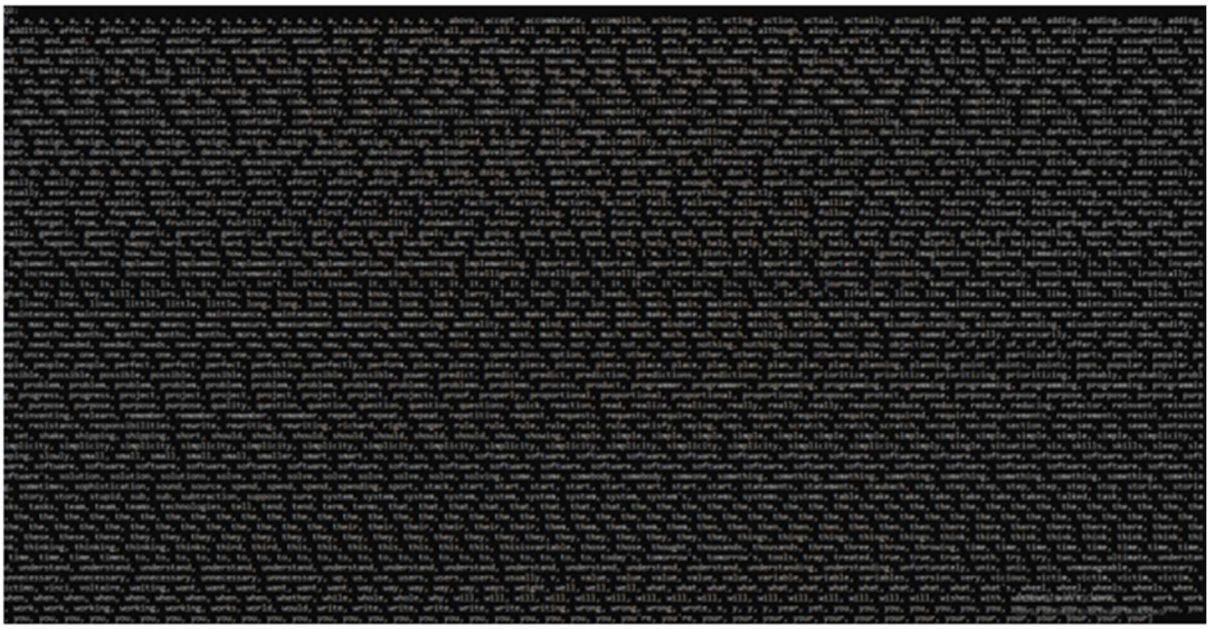

Figure 4: insertionSort output

Question 3:

```
public static ArrayList mergeSort(ArrayList s){
    int p = 0;
    int r = s.size()-1;
    if (p < r){
        int q = (p+r)/2;
        ArrayList s1 = new ArrayList<>();
        for (int i = p; i <= q; i++){
            s1.add(s.get(i));
        }
        ArrayList s2 = new ArrayList<>();
        for (int i = q+1; i <= r; i++){
            s2.add(s.get(i));
        }
        mergeSort(s1);
        mergeSort(s2);
        merge(s1, s2, s);
    }
    return s;
}
```

This method contains two parts, *mergeSort* can be called using an unordered ArrayList as an argument, this then calls *merge* with the unordered ArrayList and two other ArrayLists as arguments.

The *mergeSort* method is designed to split an ArrayList and call the other sorting method when this is done. If the size of the ArrayList is more than 1 it is split into two ArrayLists: *s1* and *s2*. The method is then called recursively for *s1* and *s2* until the length of *s1* and *s2* is 1. These are then sorted using *merge*.

The *merge* method takes *s1, s2* and the unordered ArrayList as arguments. While there are items in *s1* and *s2*, items with the lowest ASCII values get put first in a temporary array and are removed from *s1* or *s2*. If there are no more items in *s1* or *s2* then items from the remaining ArrayList are appended to the temporary array. Finally, the argument ArrayList is replaced by the sorted temporary array.

Figure 7: merge

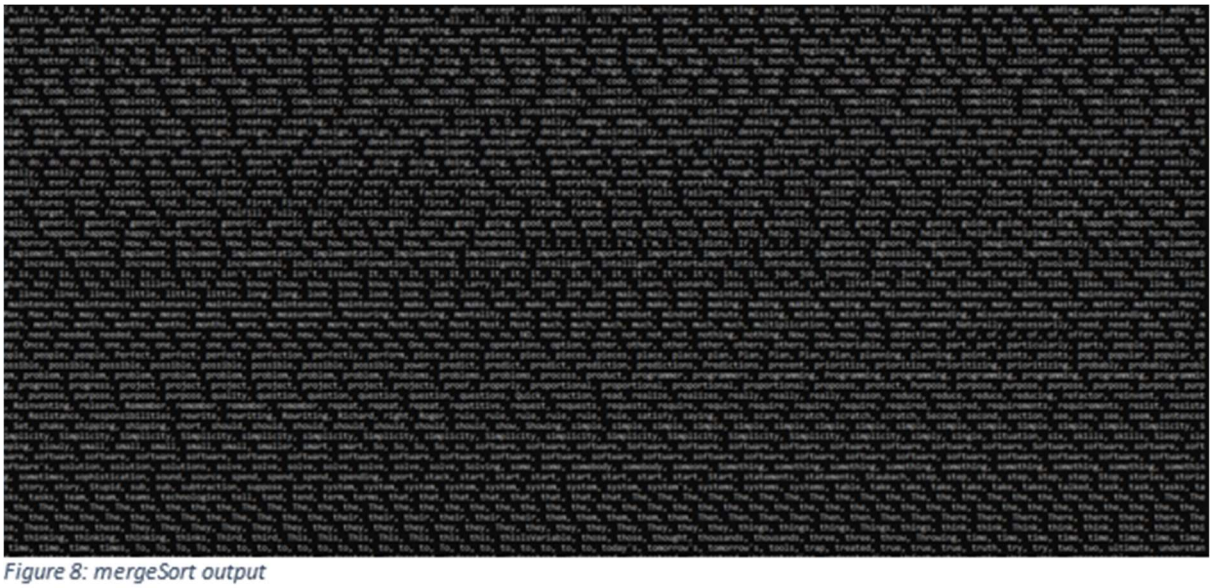

Figure 8: mergeSort output

Question 4:

To judge the performance of the merge and insertion sort algorithms, I measured the time and number of moves/swaps that each algorithm would take to sort certain lengths of unordered words.

```
public static void sortPerformance()throws FileNotFoundException{
    ArrayList inputList = sortAssignment.wordReader("input.txt");
    ArrayList stopwordList = sortAssignment.wordReader("stopwords.txt");
    ArrayList masterList = sortAssignment.deleteStopwords(inputList,stopwordList);
    ArrayList oneHundredList = new ArrayList<>(masterList.subList(0, 99));
    ArrayList twoHundredList = new ArrayList<>(masterList.subList(0, 199));
    ArrayList fiveHundredList = new ArrayList<>(masterList.subList(0, 499));
```

Figure 9: sortPerformance

I created a separate method which initialized a list of words using the *deleteStopwords* and split them into ArrayLists of 100, 200 and 500 words to test performance across a variety of lengths.

I measured the time taken in nanoseconds to finish executing the sorting methods by taking a timestamp of the system time at the start and end of execution and finding the difference between them.

```
//Measures time taken to sort words for insertion sort
public static long sortTimesInsertionSort(ArrayList listOfWords){
    long startTime = System.nanoTime();
    insertionSort(listOfWords);
    long endTime = System.nanoTime();
    return endTime-startTime;
}

//Measures time taken to sort words for merge sort
public static long sortTimesMergeSort(ArrayList listOfWords){
    long startTime = System.nanoTime();
    mergeSort(listOfWords);
    long endTime = System.nanoTime();
    return endTime-startTime;
}
```

Figure 10: sortTimes

To measure moves/swaps I added an int variable and incremented it every time a move was performed. For *insertionSort* it would increment whenever an item was modified in the new list. For *mergeSort* it incremented whenever an item is moved to the temporary ArrayList as *mergeSort* does not have swaps.

```
Q4:
Insertion sort time, 100 words: 193900 nanoseconds
Insertion sort moves/swaps, 100 words: 2021

Merge sort time, 100 words: 356500 nanoseconds
Merge sort moves/swaps, 100 words: 664

Insertion sort time, 200 words: 1192200 nanoseconds
Insertion sort moves/swaps, 200 words: 9240

Merge sort time, 200 words: 486000 nanoseconds
Merge sort moves/swaps, 200 words: 1535

Insertion sort time, 500 words: 6047400 nanoseconds
Insertion sort moves/swaps, 500 words: 63812

Merge sort time, 500 words: 1134700 nanoseconds
Merge sort moves/swaps, 500 words: 4478
```

Figure 11: sortPerformance output

Question 5:

```
/** insert theElement at the rear of the queue */
public void enqueue(Object theElement)
{
if ((rear == front-1) || (front == 0 && rear == queue.length-1)){
   Object [] newQueue = new Object[queue.length*2];
   System.arraycopy(queue, front, newQueue, 0, queue.length-front);
   System.arraycopy(queue, 0, newQueue, queue.length-front, (rear+1) % queue.length);
   rear = queue.length-1;
   front = 0;
   queue = newQueue;
}
rear = (rear+1) % queue.length;
queue[rear] = theElement;
}
```

Figure 12: enqueue

I created two methods: *enqueue* and *dequeue*. The *enqueue* adds an item to the circular queue. If called it first tests to see if the queue is full by checking if the rear is equal to the front-1, or in the case that the front is 0 the rear should be the last index. If full, a new queue is created with double the capacity. Then the front of the original queue up to the 0 index is copied into the new queue, starting at its 0 index. Then the 0 index until the rear of the original queue is copied into the new queue, starting at the next empty index. The front and rear values are then reset for the new queue.

The *dequeue* is meant to remove and return an element from the front of the queue. If the queue is empty, it returns null, otherwise it moves the front forward by one, returning the element now at the front index. It then removes the element at the front index.

```
/** remove an element from the front of the queue
  * @return removed element */
public Object dequeue()
{

   if (isEmpty()){
      return null;
   }
   else{
      Object element;
      front = (front+1) % queue.length;
      element = queue[front];
      queue[front] = null;
      return element;
   }
}
```

Figure 13: dequeue

```
Rear element   : element12
Front element  : element3
Removed element: element3

Rear element   : element12
Front element  : element4
Removed element: element4

Rear element   : element12
Front element  : element5
Removed element: element5

Rear element   : element12
Front element  : element6
Removed element: element6

Rear element   : element12
Front element  : element7
Removed element: element7

Rear element   : element12
Front element  : element8
Removed element: element8

Rear element   : element12
Front element  : element9
Removed element: element9

Rear element   : element12
Front element  : element10
Removed element: element10

Rear element   : element12
Front element  : element11
Removed element: element11

Rear element   : element12
Front element  : element12
Removed element: element12

empty queue
```

Figure 14: Circular Array output