For CM1210's Spring coursework, we were tasked with creating and testing multiple sorting algorithms in order to calculate time complexities and swap numbers occurred, and also to systematically fix a circular queue to ensure complete functionality is provided. I employed Object oriented programming (OOP), classes, constructors, methods, etc. to complete this task.

Firstly, I created a separate method in the Main class named ReadFiles(filename), which took a parameter of filename, which removed all variants of punctuation in a sentence, whilst returning all separated lines as a list of strings.

The InsertionSort() class involves taking an element in the allWordsStoppedRemovedList, and inserting the value into the correct index, depending on the value of the element; this is repeated multiple times until the list is sorted. I employed a for loop, which looped through the elements in stopwords, and a nested while loop, which compared the values of elements in input, until it was alphabetically sorted accordingly. The time complexity in the first for loop is O(n), also known as linear complexity, meaning the running time of the algorithm will increase linearly with the number of inputs. The nested while loop however, has a time complexity of O(n^2), also known as quadratic complexity, meaning that the running time is proportional to the square of the number of items in the list; overall, being quite inefficient as a sorting method, however, it is functionally the only method of creating an insertion sort, as it needs to go through the list multiple times. This resulted in the following time calculations for the insertion sort:

```
The time it took to sort 100 words in the insertion sort (Nanoseconds): 3988900
The time it took to sort 200 words in the insertion sort (Nanoseconds): 1992500
The time it took to sort 442 words in the insertion sort (Nanoseconds): 9972800
The total number of swaps in the insertion sort: 39498
```

The MergeSort() class involves recursively splitting the list in half, until the items are broken-down into singular values. These values are then repeatedly sorted with the values in the previously broken-down lists, until the list is fully back-together and sorted. This process is illustrated in my method MergeSort(), where this process occurs recursively until all the values are singled out and sorted (using a for loop in my merge method), before being built back up again into the fully sorted list. The recursive nature of the merge sort gives it a unique time complexity of O(n log n), also known as loginear complexity, meaning that the performance of the algorithm goes up linearly, whilst the n goes up exponentially, being a fairly fast time complexity overall. This resulted in the following time calculations for the merge sort:

```
The time it took to sort 100 words in the merge sort (Nanoseconds): 1994800
The time it took to sort 200 words in the merge sort (Nanoseconds): 2991700
The time it took to sort 442 words in the merge sort (Nanoseconds): 3990200
The total number of swaps in the merge sort: 442
```

Figure 1 shows all time complexities, and if we are comparing the efficiency of the insertion and merge sort, we can see in the diagram that the time complexity of the merge sort, O(n log n), is more efficient than the insertion sort's, O(n^2), efficiency, which is clearly shown in my results calculated above for these time expectancies; therefore, I would recommend using a merge sort instead of an

insertion sort as the performance is better overall, especially for longer lists, where the merge sort is clearly superior.

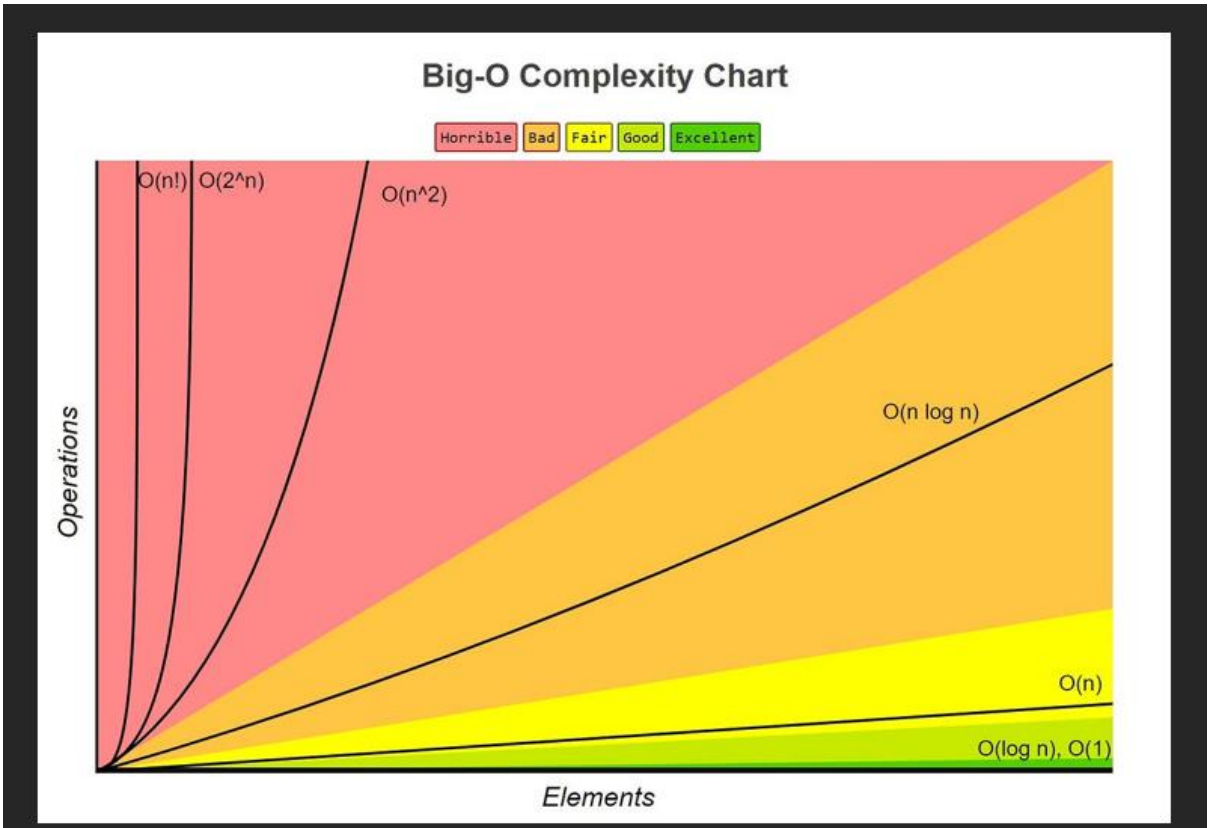

Figure 1

I created multiple constructors in order to parse data from the insertionsort(), mergesort() and sortWordsBasic() classes back into the main class where the results could be initialised. However, they would firstly all have to make a constructor for the SortResult class, which took the 3 essential return values from these classes and converted them into a String list containing the sorted list, an integer containing the number of swaps, and an integer containing the time efficiency of the algorithm (in nanoseconds). These values were returned to their respective classes, before being returned back to the main class for examining and executing. In conclusion, I used an efficient OOP approach to alphabetically sort a list of words using two different methods, whilst also finding their natural efficiencies and number of swaps occurred.

We were then tasked with fixing the current circular queue given to us in the task, which involved retroactively creating the enqueue and dequeue section of the code. I read all relevant lecture notes on the topic to figure out the complexities of a circular queue. The enqueue method I developed will increase the rear pointer (or using modulus, will return the pointer back to the beginning of the queue, if the space is not filled), and will place an Object element in that position; otherwise, if full, a do while loop will create a new list and plant all previous elements inside, with pointers incrementing respectively. The dequeue method will check to ensure the list is not empty (if so, it will return null),

however, if its not, then increment the front pointer (go to beginning if empty) and set the value to null before returning it as a previously established variable. These are the results of these methods:

```
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
```

Overall, this taught me a lot about time complexities, sorts and OOP approaches I can develop to improve my coding theory/development.