The first question of this code includes a method that takes an input text and an input stop-words to take away input text's words from the words that included in stop-words. The program first uses a scanner class to open both files than reads both files and adds words in both files into separate array list's. It than uses a loop to find words that match and then takes away these words from the input text's array list and prints it out. The code is useful since it can be used in a NLP application where before AI takes the key words useless parts can be removed (e.g. the, is). An issue and an improvable part could be the contraction words and how the program deals with them.

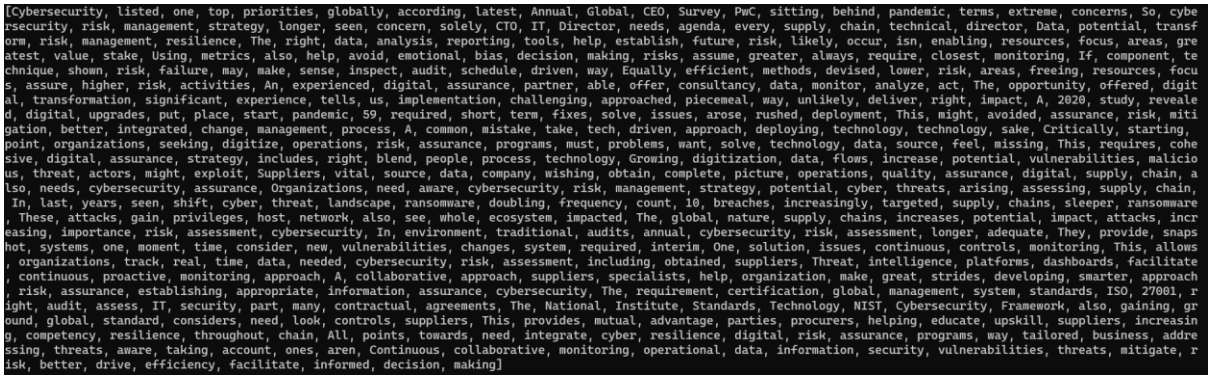
The second question of this code includes a method written for insertion sort for array lists. The code takes an array list that contains strings and sorts them via alphabetical order by going from left to right and sorting two elements in each comparison. The code can be useful for sorting small datasets or text's to than maintaining the order of equal elements. An improvement could be that another algorithm can be used for faster sorting since the time complexity of insertion sort is O(n^2). Even though it is memory efficient.

The third question of this code includes another method for sorting which is merge sort. The code takes an array list that contains strings and shorts it alphabetically using a divide and conquer approach by dividing the dataset into two parts each time until they are small enough and then work backwards to sort each item using recursion. This code is more useful for big datasets since it is faster in time complexity where time complexity of merge sort is O(n log n). But this algorithm is also not very memory efficient since it has to store the divided parts of the dataset somewhere.

The fourth question asks me to run some tests on these two algorithms from question 2 and 3 to see which one is better. In the first test I was supposed to measure the time for running these algorithms for 100, 200 and 500 words. Merge sort is seen to be faster in these tests compared to insertion sort. In the second test I was required to change the algorithms and add them a counter that counts the number of swaps. Merge sort was still better than the insertion sort where number of swaps are compared. The only two parts where merge sort is worse than the insertion sort is the memory part since it stores array lists in order to achieve divide and conquer and the coding part since it includes more code to be achieved/completed.

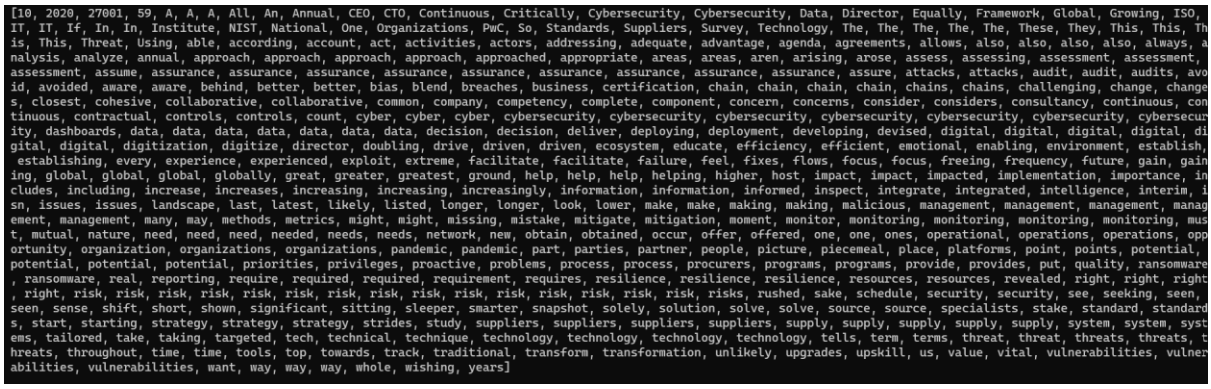
The fifth question asks me to finish two java methods in MyArrayQueue.java file where the file includes Queue implementation using arrays. The first method was enqueue which is used to add an element to a queue object. The second method was dequeue which is used for deleting an element of a queue object and returning the deleted value. Some advantages of array queue are first in first out (FIFO), building real time applications with it. Some disadvantages are memory since it doubles each time even if a double size might not be required it takes unnecessary space and it is not suited for priority queues.

Question 1:



Question 2:

Question 3:



Question 4:

```
Runtime(in nanaoseconds) for 100 values in insertion sort: 45968500
Number of Sorts for 100 values in insertion sort: 2244
Runtime(in nanoseconds) for 100 values in merge sort: 4696500
Number of Sorts for 100 values in merge sort: 1058
Runtime(in nanaoseconds) for 200 values in insertion sort: 3770300
Number of Sorts for 200 values in insertion sort: 9646
Runtime(in nanoseconds) for 200 values in merge sort: 3243000
Number of Sorts for 200 values in merge sort: 4901
Runtime(in nanaoseconds) for 500 values in insertion sort: 9213700
Number of Sorts for 500 values in insertion sort: 60440
Runtime(in nanoseconds) for 500 values in merge sort: 2822000
Number of Sorts for 500 values in merge sort: 29739
```

Question 5:

```
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
```