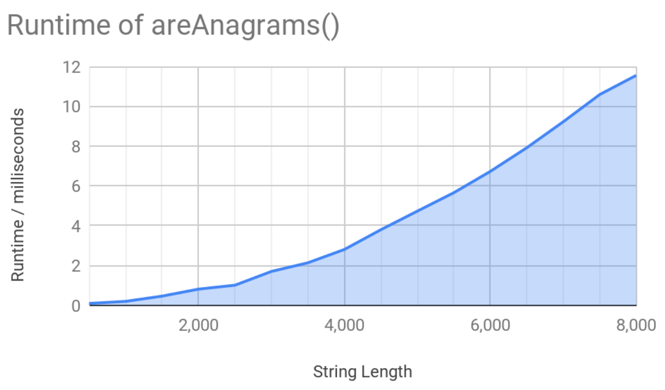
Assignment 4: Anagram Checker

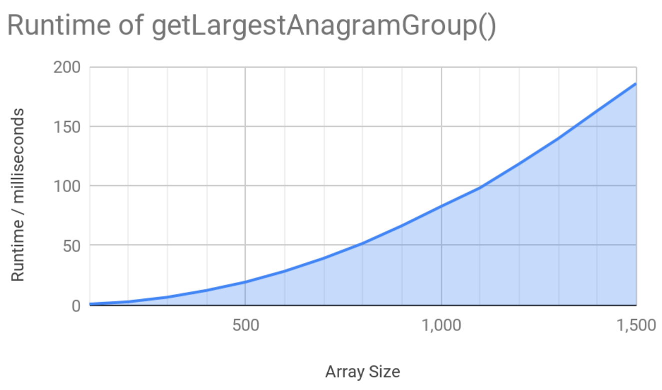
Analysis Document by Qianlang Chen (u1172983)

As in assignment 2 and 3, I had Brandon Walton as my programming partner in this one again. Brandon seemed experienced with sorting algorithms and programming in general; he also knew many tricks in coding and did not make mistakes that many coding beginners usually make. Because of that, we worked so efficiently that we finished our *AnagramChecker* class in about 2 hours. I will plan to work with Brandom again because coding with him is enjoyable and efficient.

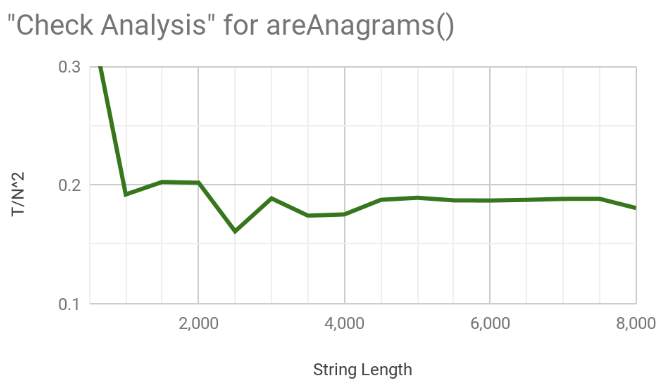
We encountered and implemented two insertion sorting methods in this assignment: one sorts the characters in a String alphabetically *[String sort(String)]*, and the other sorts the elements with any type in an array with a given comparator *[void insertionSort(T[], Comparator)]*. The method which sorts Strings has to have a return value since Java **does not** allow Strings to be directly modified once they are created, so the method creates and returns a new String containing the alphabetically-sorted characters. The *insertionSort()* method, on the other hand, **can** modify the elements in the array because Java allows so, so the method moves the elements in the array around to finish the sort without having to return a brand new array.

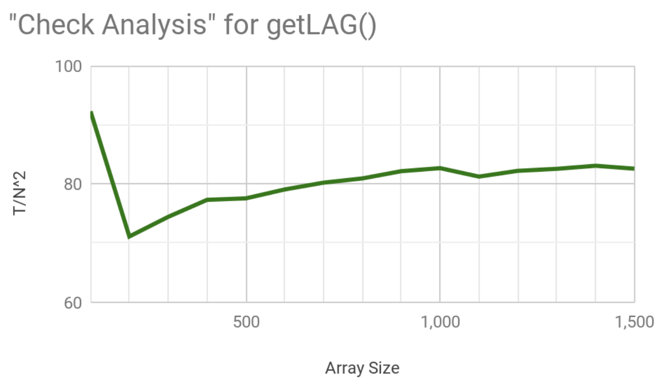
To analyze the performance of our *AnagramChecker* class, I ran and recorded the runtimes of two of the most important methods of the class: the *areAnagrams()* and *getLargestAnagramGroup()* methods. For *areAnagrams()* method, I chose to use the **lengths of the Strings** passed into the method to represent different data sizes, and for *getLAG()* method, I generated different **sizes of arrays** containing random Strings of the same length to pass into the method. I then plotted and graphed my results to figure the behavior of the runtimes of the methods versus the data size. I also used the "Check Analysis" technique to verify my conclusions about the behaviors of the methods. Here are my results:





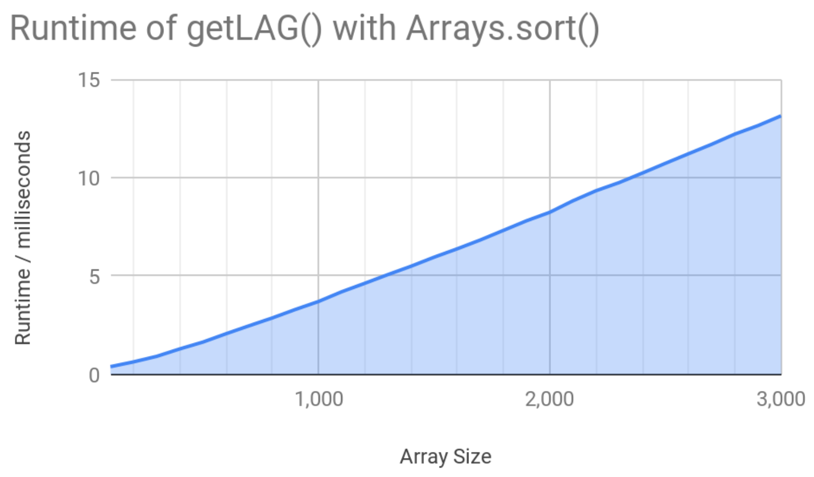
Both of the methods show a **quadratic** behavior in the graphs. The reason is, in our *areAnagrams()* method, we compare the sorted versions of the two String arguments to tell if they are anagrams. Sorting requires using the insertion sort we have implemented in the class, which has a quadratic runtime behavior. For similar reasons, our *getLAG()* method uses insertion sort on the array, giving it a quadratic behavior as well. Even though the *getLAG()* method also calls *andAnagrams()* method, but the time it costs to run is still controlled by the size of the array, so its quadratic relationship to the data size is direct to the size of the array, instead of the sizes of the Strings in the array. Finally, using the technique of "Check Analysis", I took each runtime divided by the Big-Oh behavior and graphed the results:



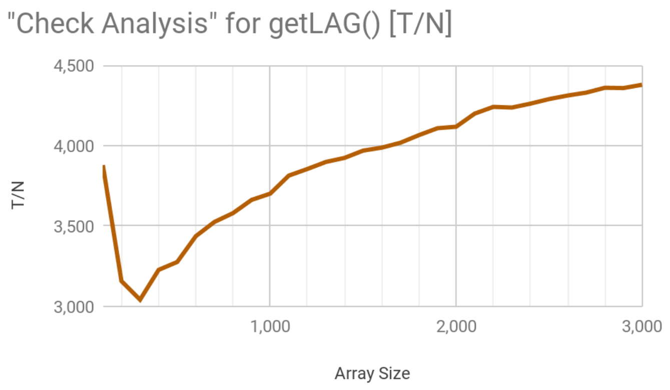


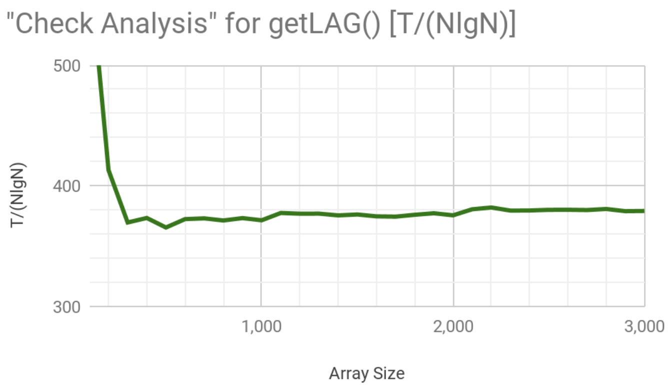
The results shown in the graphs seem to **converge** to a constant, which means my conclusions of our methods' behaviors are correct.

Java provided a native sorting method for arrays which uses a Mergesort. Mergesort divides the array into halves, sorts them, and repeats this process until the array cannot be divided into smaller pieces. Its sorting mechanism has a linear behavior because it takes advantage of the fact that the smaller pieces of the array are also sorted, therefore, the Mergesort must have an overall Big-Oh behavior of NlgN. However, again, since our *getLAG()* method also calls *areAnagrams()* method which uses insertion sort on Strings, it will still have an overall behavior of quadratic due to its dominance as the data size gets bigger. If we use the Mergesort method on both of our two methods, then we will get an NlgN behavior for our *getLAG()* method. To verify this, I temporarily modified the code so that it used Java's native sorting method instead and graphed the runtimes:



The graph shows that the behavior could be either linear or NlgN. I then used the "Check Analysis" method to process the result and to figure out which one it is:





From the graphs, I see that the behavior of our *getLAG()* method using Java's native *Arrays.sort()* method is **NlgN,** because its T/(NlgN) graph seems to converge to a constant more than the T/N graph. According to my reasoning above, this result makes sense.