1. Analyze the run-time performance of the areAnagrams method.
-What is the Big-O behavior and why? Be sure to define N.

-Plot the running time for various problem sizes (up to you to choose problem sizes that sufficiently analyze the problem). (NOTE: Use the method you wrote for generating a random string of a certain length.)

-Does the growth rate of the plotted running times match the Big-O behavior you predicted?

The behavior of .areAnagrams() should be O(n^2). Because our underlying algorithm is a pair of insertion sorts every element in our array (string) must be viewed and then potentially compared to every other element in that same array (string). Our best case scenario will be better than that, but as our array (string) size gets arbitrarily large our algorithm will still progress towards O(n^2).

I compared two words of equal length (n) and recorded the difference in time areAnagrams took at different word lengths. See figure 1 for the results.

The recorded growth rate indeed appears to be very close to O(n^2) and our line compares closely to the slope of n^2.

2. Analyze the run-time performance of the getLargestAnagramGroup method using your insertion sort algorithm. (Use the same list of guiding questions as in #1.) Note that in this case, N is the number of words, not the length of words. Finding the largest group of anagrams involves sorting the entire list of words based on some criteria (not the natural ordering). To get varying input size, consider using the very large list of words linked on the assignment page, save it as a file, and take out words as necessary to get different problem sizes, or use your random word generator. If you use the random word generator, use a modest word length, such as 5-15 characters.

Our behavior of .getLargestAnagramGroup() should also be O(n^2), this time the math is messier though. At the algorithm's heart we are still utilizing an insertion sort to sort through every string in our array of strings. For the same reasons as stated above this will resolve to O(n^2), where n is the number of strings in the array.

We are still calling .areAnagrams() as part of our comparator for sorting, but because the word size is so small (I chose a length of 10 letters) as our array of strings gets arbitrarily large the cost of running .areAnagrams() becomes negligible. We also have to look through our array after it's sorted which adds a cost of n, but again as our array of strings grows according to O (n^2) that cost becomes negligible as well.

I sorted multiple arrays of strings with different sizes and recorded the time required for each sort. See figure 2 for the results.

Again, the recorded growth rate seems to match up with our expectations - it closely compares to the slope of n^2.

3. What is the run-time performance of the getLargestAnagramGroup method if we use Java's sort method instead? How does it compare to using insertion sort? (Use the same list of guiding questions as in #1.)

The javadoc on Arrays.sort() informs me that the java implementation uses a mergesort at the basic level, but with a number of optimizations that greatly increase performance (approaching O(n) when the array is close to sorted. Since our array is entirely randomly generated I would expect Arrays.sort() to be close to O(nlog(n)) as it is with a general mergesort.

I used the java Arrays.sort() method to sort multiple arrays of strings with different sizes and recorded the time required for each sort. See figure 3 for the results.

Surprisingly, Arrays.sort() beat my expectations. Even with our array of strings more or less created at random, Arrays.sort() approximates O(n) incredibly closely. So close in fact that it's difficult to differentiate the lines on the graph

Figure 1 -

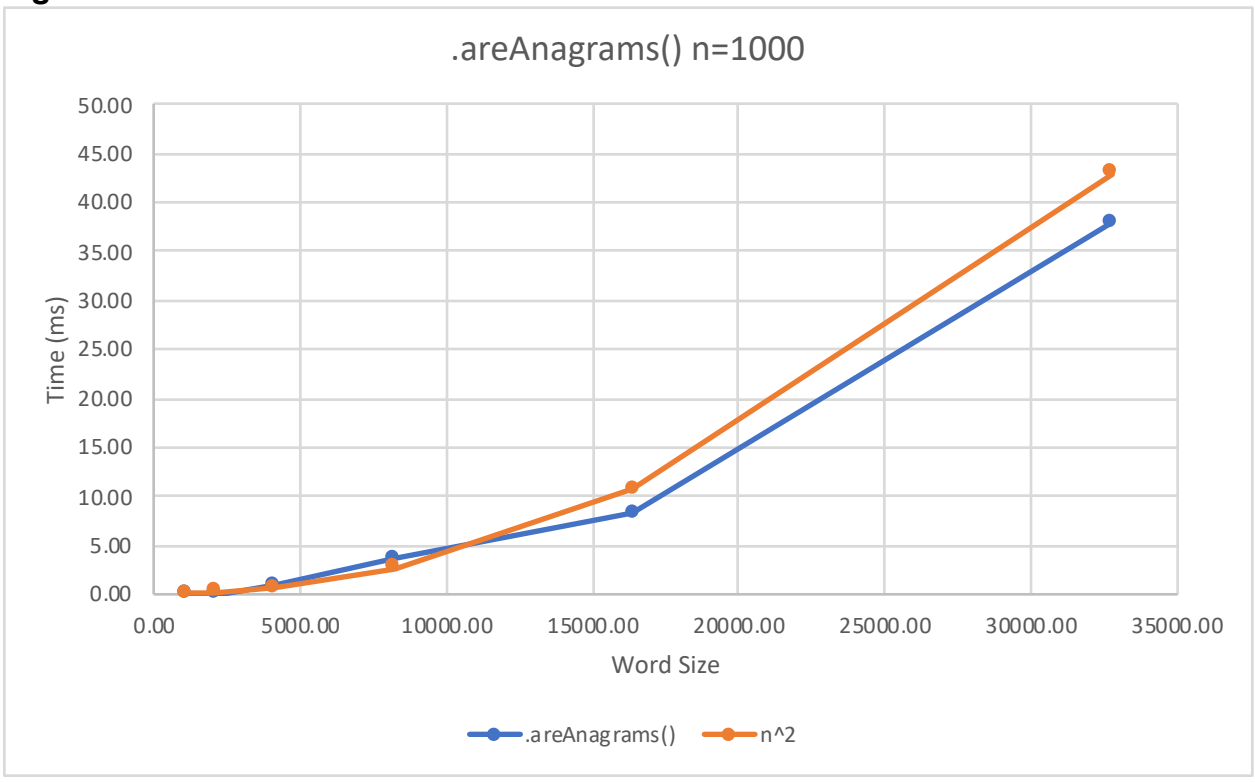

Figure 2 -

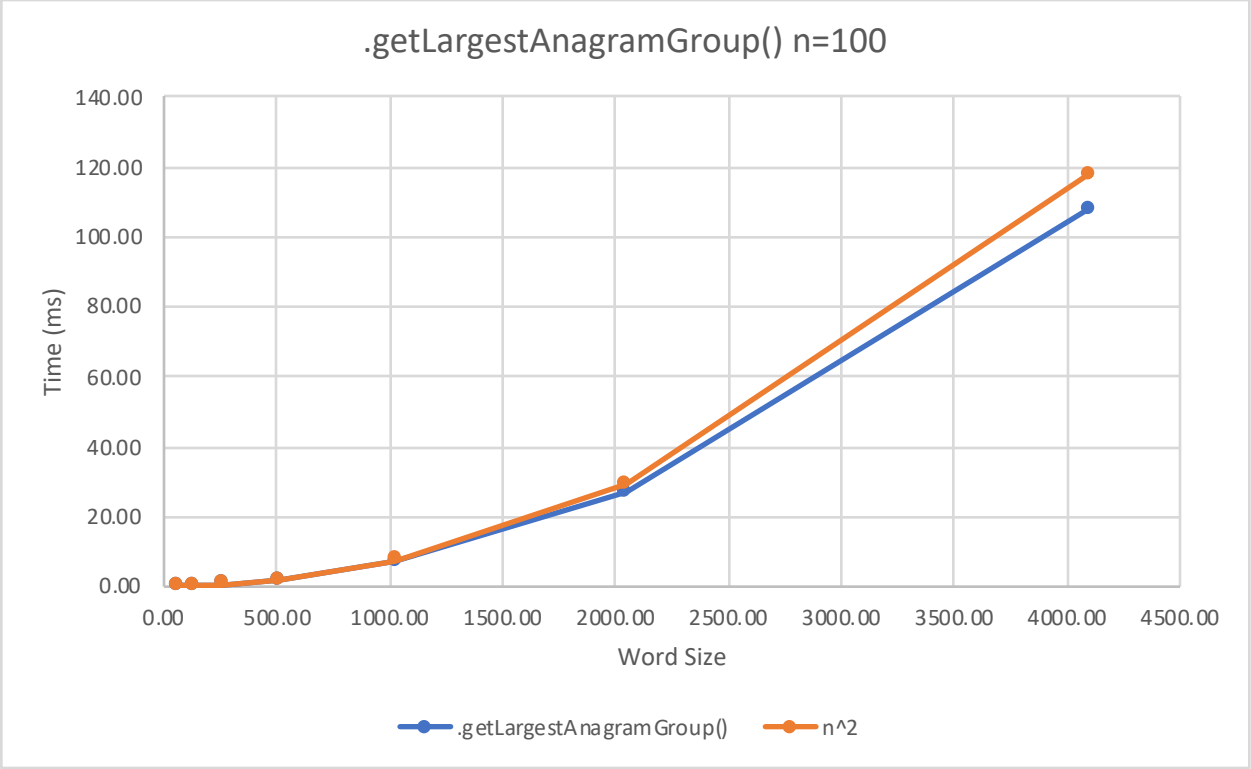

Figure 3 -