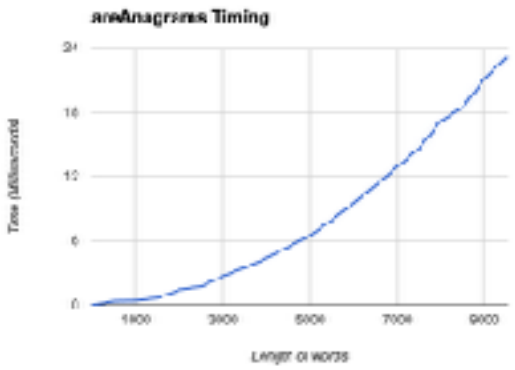
1.) Analyze the run-time performance of the areAnagrams method.

- 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: You may want to randomly generate some strings for this or randomly choose a subset of the English Language file.)



I predict that the Big-O notation for areAnagrams will be N^2 (N being the length of the words). Because the sorting method used has the same complexity and we ignore the constants. We randomly generated some strings of length N and averaged the time for the comparison.

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

No, as the graph shows the relationship is exponential.

2.) Analyze the run-time performance of the getLargestAnagramGroup method using your insertion sort algorithm. (Use the same list of guiding questions as above.) 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 a random word generator (write a function to randomly build a string of characters). If you use the random word generator, use modest word lengths, such as 5-15 characters.

To time this method we made a list of N length and tested varying N lengths. We found that the relationship between the time and N was definitely not linear. We increased N linearly but the time became more and more spaced out so the trend will definitely be exponential. The data points for getLargestAnagramGroup go as follows:

Number of Components: 10 Time: 0.09734396000000001
Number of Components: 50 Time: 0.42446126
Number of Components: 100 Time: 1.0184355999999999
Number of Components: 500 Time: 14.33530339
Number of Components: 1000 Time: 55.17807436
Number of Components: 5000 Time: 1318.55598122

As you can see the gap between the time of 1000 components and 5000 components is very big.

3.) Analyze the run-time performance of the insertionSort method using an array of strings and an array of integers. Does the speed of computing match the speed of the getLargestAnagramGroup? Explain why/why not.

The data points below show the sorting speeds for varying sizes of the integer and String arrays. As you can see that String arrays take more time to sort than the integer arrays do. The speed is way faster than the getLargestAnagramGroup because the sort only sorts the one time. getLargestAnagram has to sort the Strings and the list and go through and see which group is the largest.

Insertion sort timing for Strings:

Number of Components: 1000 Time: 0.09381971
Insertion sort timing for Integers:
Number of Components: 1000 Time: 0.11156018

Insertion sort timing for Strings:
Number of Components: 5000 Time: 1.66923912
Insertion sort timing for Integers:
Number of Components: 5000 Time: 1.08902559

Insertion sort timing for Strings:
Number of Components: 10000 Time: 5.25615788
Insertion sort timing for Integers:
Number of Components: 10000 Time: 2.64081995

Insertion sort timing for Strings:
Number of Components: 40000 Time: 95.14575295
Insertion sort timing for Integers:
Number of Components: 40000 Time: 31.07517387

4.) What is the run-time performance of the getLargestAnagramGroup method if we use Java's ArrayList sort method instead of our own (http://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html)? How does it compare to using insertion sort? (Use the same list of guiding questions as above.)

The insertion sort that we wrote is slower than the Java's built in sort which is merge sort/ quick sort(Which is expected). This makes sense because the complexity of the insertion sort is N^2(N being the number of components) whereas the merge sort is N*log(N), which will be much faster.