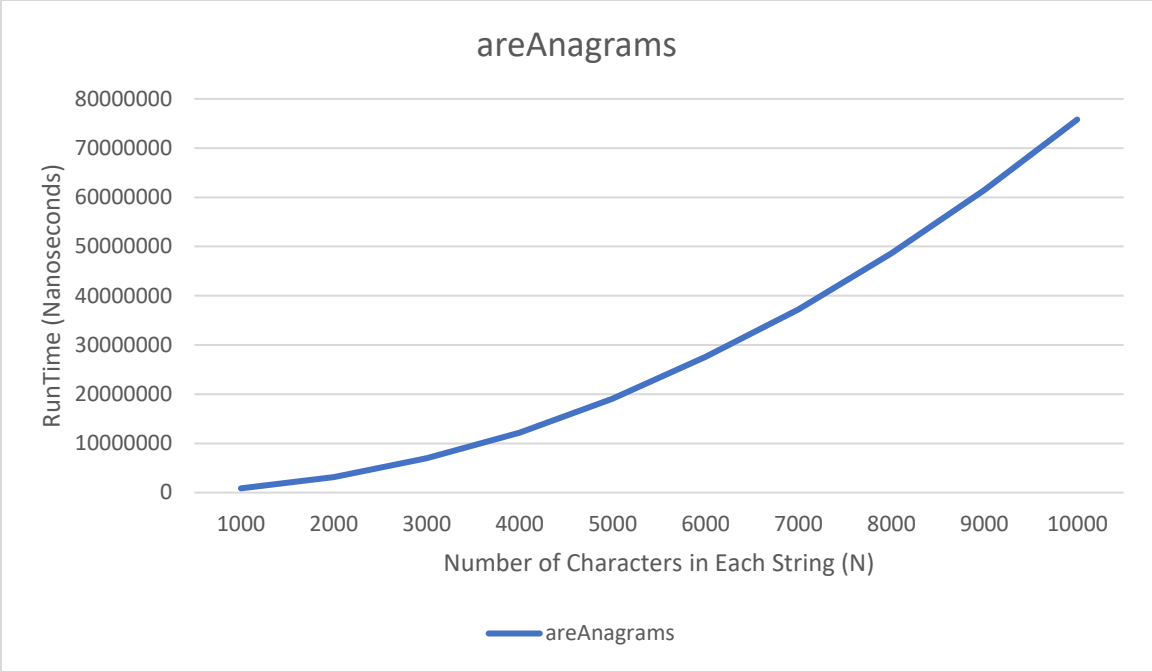
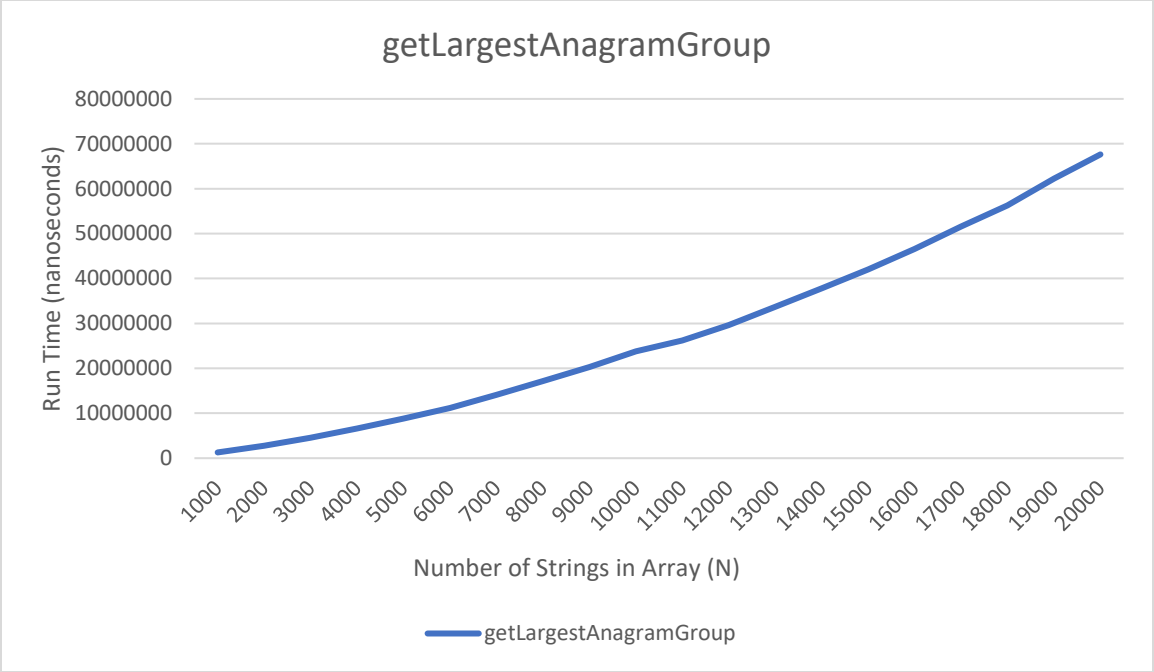
Analysis Document: AnagramChecker

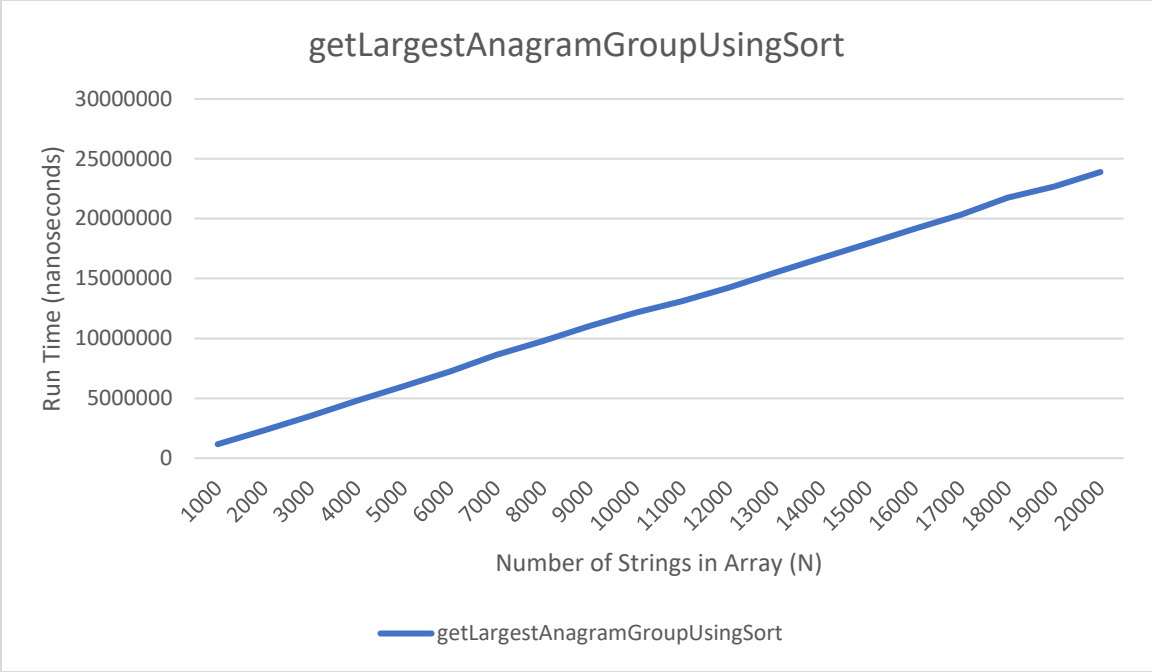
1. My partner and I did well on applying the pair programming techniques for this assignment. We switched after about every thirty minutes or so, and we also made sure that each of us got to work on a well-rounded representative sample of the assignment (i.e. we didn't just code methods that were similar, we alternated to ensure we did various methods). I think we did pretty well in developing this program, but we did run into a few issues due to misunderstanding the instructions for the sort() method. We coded it to ignore case, but we were supposed to code it to include case sensitivity and the other methods would ignore case. We had to adjust our code for that, but otherwise we were effectively. I believe we spent about 10 hours on coding and testing this program. I plan to work with my partner again, I think we work together well and we both are good about taking on both roles of pair programming as well as listening and communicating effectively.
2. The signature of the sort method has a return type of String because we are not altering the string itself, instead, we are creating a new string that is the lexographically sorted version of the string parameter. Thus, we have to return the new string, as the parameter string wasn't altered. We need to use the un-altered string in the other methods, so we cannot alter the variable that holds the string, otherwise we would use the sorted string in methods like the insertionSort, getLargestAnagramGroup, etc. and those methods rely on having both the unsorted version and the sorted version. The insertionSort method, however, is altering the parameter array. We want to sort the array itself so that we can use it, and because we don't need the unsorted array again for any other method, it is not necessary to create a new array to hold the sorted array. Thus, we don't need to return anything because the referenced array itself is being altered, so returning it would be redundant as we already have access to that sorted array. Additionally, insertion sorts are implemented "in-place", or in other words, within the original array itself. Thus, there is no new array to return, and we don't need to return the unsorted array because it is being altered in the method.

3. 

N for the areAnagrams method is the number of characters in the two strings that we are comparing. So if string1 has 1000 characters, and string2 has 1000 characters, then N = 1000. We never change the number of strings that we pass into the areAnagrams method, so the only aspect that can alter the problem size is the lengths of strings used. The Big-O behavior for the areAnagrams method is N^2, because in the areAnagrams method we call the sort() method, which uses an insertion sort to sort the strings. The Big-O complexity of an insertion sort is N^2. Once we return the sorted strings to the areAnagrams method, we run over it N times to compare the two strings. So this makes the total run equal to N^2 + N, but for Big-O complexity we only care about the limit as the problem size grows larger, so the Big-O complexity is N^2. The growth rate of the plotted run times does match this Big-O behavior, both in the shape of the graph as well as the ratio between the measured times and the actual Big-O times. When I found the ratio between the measured times and actual times, I got roughly the same proportion of 0.8 across all of my running times. This confirms that the Big-O complexity is N^2, because the proportion would not have been constant if it wasn't N^2.

4. 

N for the getLargestAnagramGroup method is the number of strings contained in the array that we are checking. For example, if there are 1000 strings in the parameter array, then N is 1000. The Big-O complexity for this method should be N^2, because we call the insertionSort() method inside the getLargestAnagramGroup() which has a complexity of N^2. Then, we loop over the length of that list to compare each value, so that would end up being N, and our total runtime is N^2+N, but because we only really care about the limit, the Big-O complexity should be O(N^2). The graph looks a little like a quadratic graph in that it curves upward a bit, but not nearly as much as it should for a proper N^2 graph. Additionally, the proportions across the measured runtimes and the expected Big-O complexity do not stay constant. In fact, they decrease quite significantly from N = 1000 to N = 20000. It did not take as long to time as it should have, which leads me to believe that the error lies in my timing code, not my getLargestAnagrams() code. This belief is also due to the extensive testing done on my getLargestAnagramGroup() code, and each test shows that this code is working as it should. I also checked the proportion across the measured runtimes using N instead of N^2, N^3 instead of N^2, and both were incorrect as well. This makes me wonder if it is an issue caused by external factors such as my computer's processing speed, issues with background processes impacting the timing, etc. I cannot pin down exactly where this error is, but even though the proportions are not constant, N^2 still most closely resembles the actual running time of this method.

5. 

N for the getLargestAnagramGroup using Java's sort() method is the number of strings contained in the array that we are checking. As above, if there are 1000 strings in the parameter array, then N is 1000. Theoretically, the Big-O complexity for this should be O(NlogN) as that is the Big-O complexity of Java's sort() method. However, clearly the actual Big-O complexity ended up being N. The ratio that I calculated across the measured runtimes and N were roughly constant, with all ratios hovering around 1100. This could be due to an error in implementing the sort() method in the altered getLargestAnagramGroup() method. It could also be issues with my processing system, background processes impacting run times, etc. though none of those would keep the proportion constant. Therefore, it must be an error in either the timing of my code, or the changed getLargestAnagramGroup() method itself. Especially due to the error in the previous test, I believe it is an error with the timing of my code, not with my code itself as I have extensively tested the method, and the only difference was in the method I used to sort the parameter array, and thus should still pass all tests. It is possible that I didn't include everything that needed to be timed, and that could be what caused the error in both question four and this question. Complexity errors aside, it is clear that the insertionSort() is far less efficient than using Java's sort method. After a bit of research, I learned that Java's sort method uses a quicksort method, and the average complexity for a quicksort is O(NlogN) whereas the average for an insertion sort is N^2. Thus, it is much cheaper to use sort instead of an insertion sort.