1. Who is your programming partner? Which of you submitted the source code of your program?

My programming partner is: Philippe David (u0989696). I submitted the source code for our program.

2. What did you learn from your partner? What did your partner learn from you?

I learned how to use the auto formatter in the Eclipse IDE. I showed Philippe the test driven development development methodology, where we write the Javadocs and the unit tests first before implementing the code.

3. Evaluate your programming partner. Do you plan to work with this person again?

I plan to work with this person again. Philippe was deeply knowledgeable about the Eclipse IDE and Java. We were able to share a lot of clever tricks that made life much easier (e.g. the auto formatter, automatically filling in method signatures, etc.). Having a second pair of eyes also helped us catch bugs.

4. Analyze the run-time performance of the areAnagrams method.

- What is the Big-O behavior and why? Be sure to define N.

I expect the Big-O behavior to be $O(N^2)$ because in the `areAnagrams` method is dominated by the sort method, which uses insertion sort. Insertion sort, as was discussed in class, scales $O(N^2)$ because of the two nested loops. N in this case, is the number of letters in the two words to be compared.

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

See Fig. 1.

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

Yes.

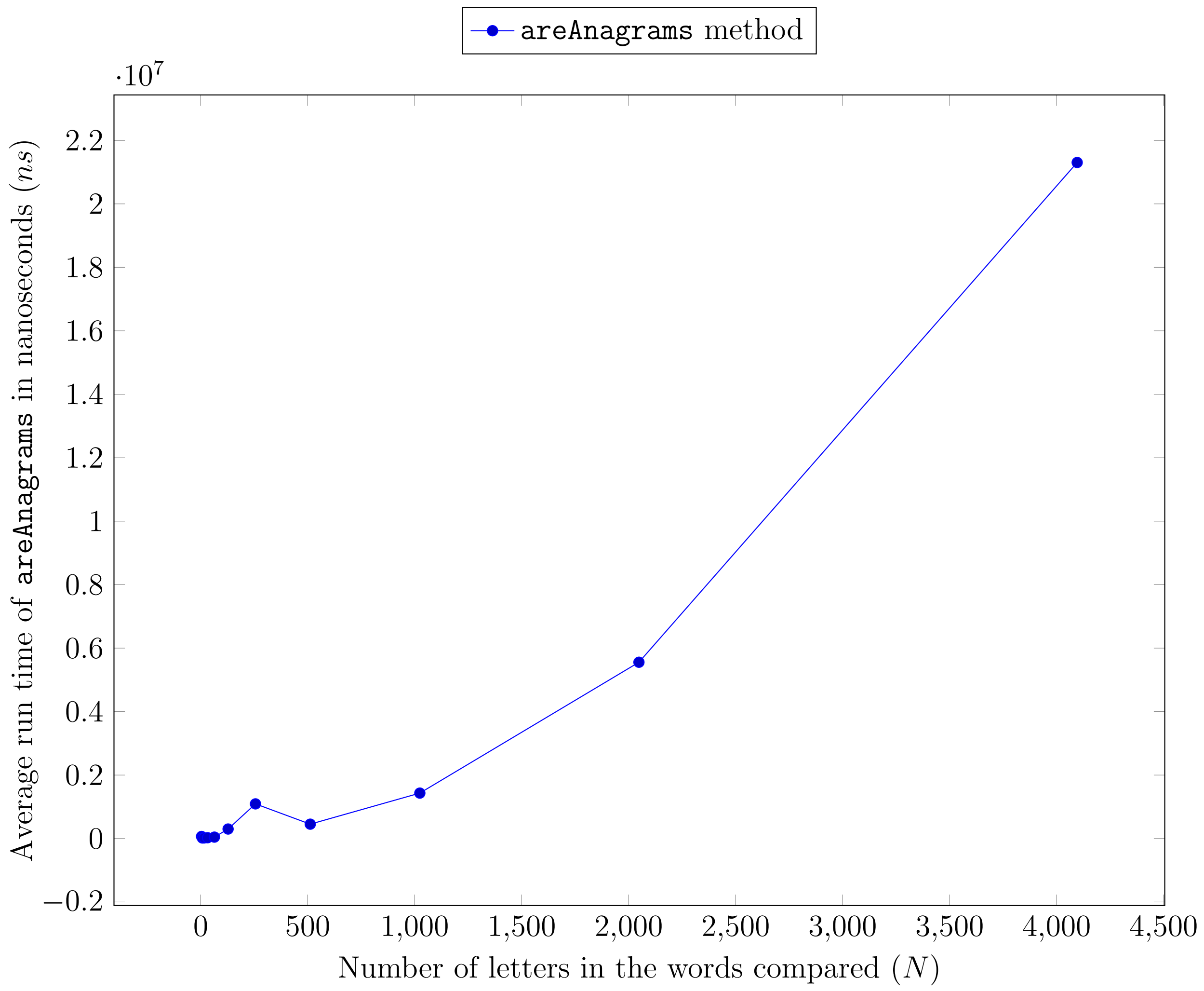

Figure 1: Average Run time of `areAnagrams`.

5.Analyze the run-time performance of the getLargestAnagramGroup method using your insertion sort algorithm. (Use the same list of guiding questions as in #4.) 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, provided in AnagramTester.java. If you use the random word generator, use a modest word length, such as 5-15 characters.

I expect the Big-O behavior to be $O(N^2)$. In this case, N is defined as: the number of words from which the largest group of anagrams is to be found, not the length of words. The length of words is kept constant. I expect the behavior to be $O(N^2)$ because I decided to sort the list of words whose letters have been sorted before searching for the largest group (sorting the words puts the groups of anagrams next to one another). As mentioned earlier, insertion sort scales $O(N^2)$.

Since the length of words is kept constant, sorting the letters in each word scales linearly $O(N)$, which when summed with sorting the words would had asymptotically diverged, thus leaving only $O(N^2)$ as the dominant term. The same applies when searching for the largest group of anagrams, which used linear search.

Had both the number of words and the length of words increased with N, $O(N^3)$ may have been observed because insertion sort $O(N^2)$ for each word in the list $O(N)$. The growth rate observed in the plot matches my expectations, see Fig. 2.

6. What is the run-time performance of the getLargestAnagramGroup method if we use Java's sort method instead (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 in #4.)

The run-time performance of the `getLargestAnagramGroup` method using Java's sort method is $O(N\log(N))$, see Fig. 3. This scales considerably better when compared to insertion sort, see Fig. 4. I expect Java's sort method to scale $O(N\log(N))$ because it uses merge sort according to Java's documentation, which runs recursively using a divide and conquer design pattern. The divide is defined recursively where the problem is repeatedly halved, which as discussed in the last assignment would scale $O(\log(N))$. The conquer is a merge operation, and thus scales $O(N)$. Putting the two together, I get $O(N\log(N))$. This expectation matches what I observe in the figure, where there is an initial bump at the beginning where $O(\log(N))$ influences the curve before it flattens out and the curve's growth rate becomes dominated by the linear term.

7. How many hours did you spend on this assignment?

I spent 8 hours working on this assignment.

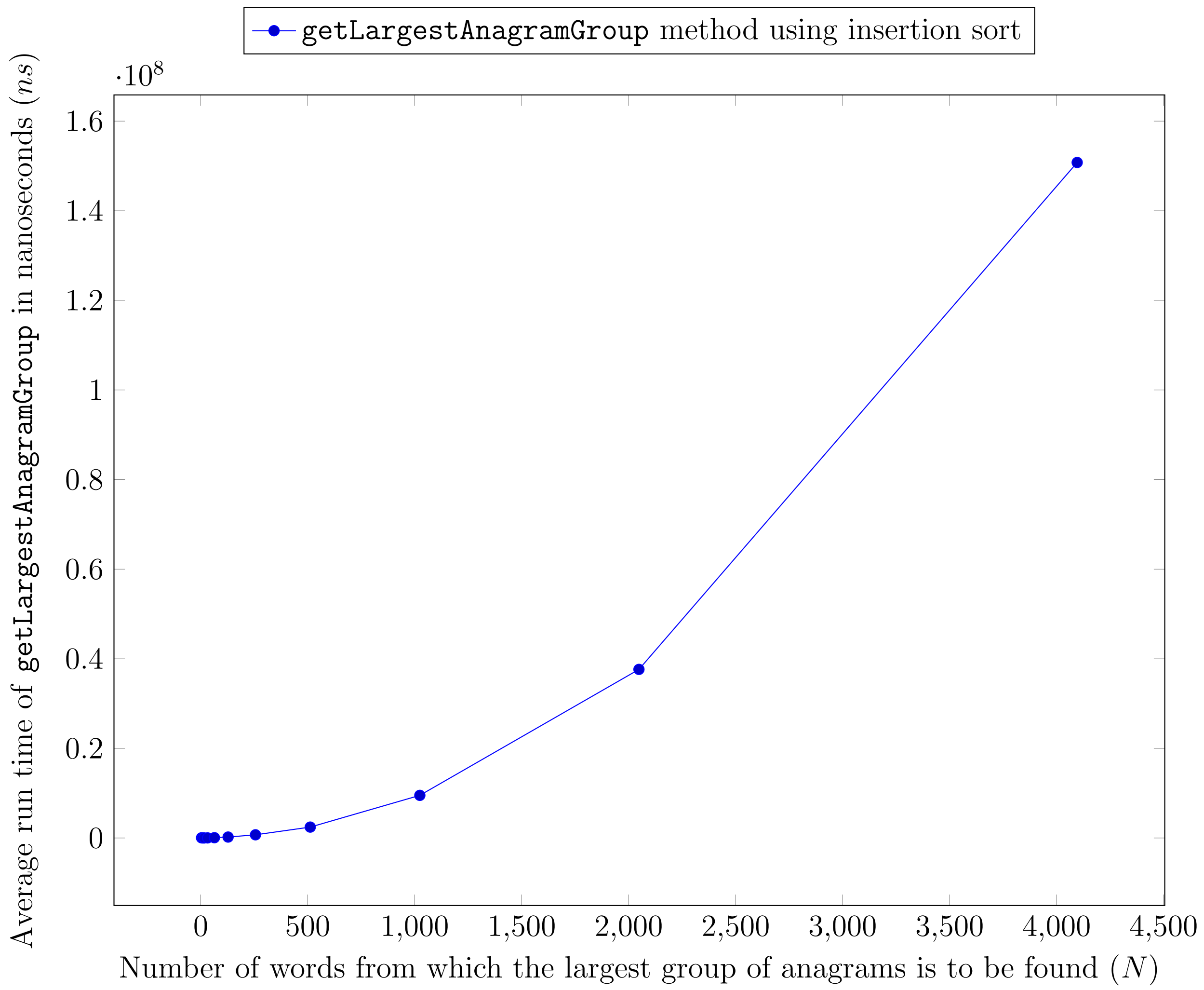

Figure 2: Average Run time of `getLargestAnagramGroup` using insertion sort.

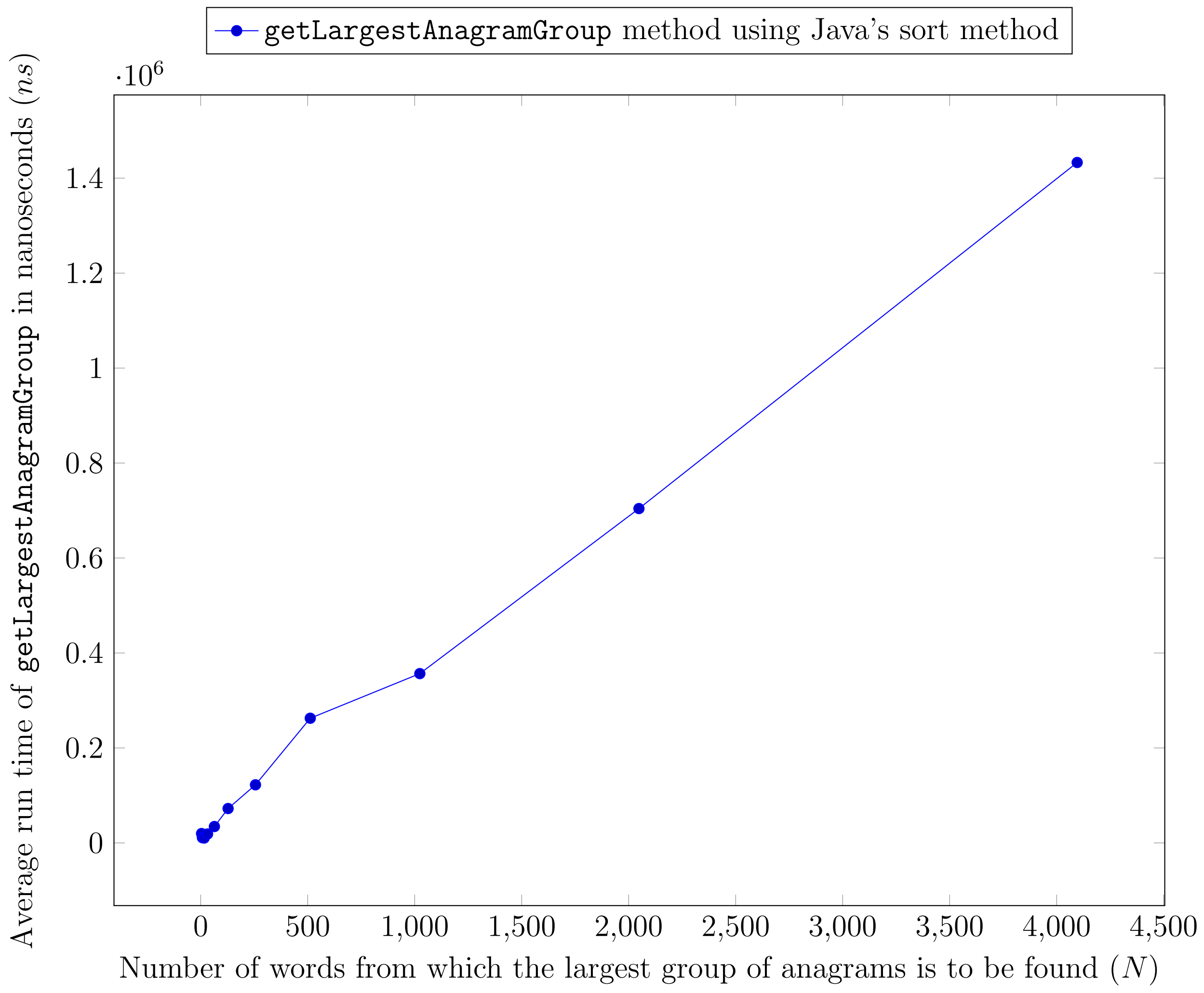

Figure 3: Average run time of `getLargestAnagramGroup` using Java's sort method.

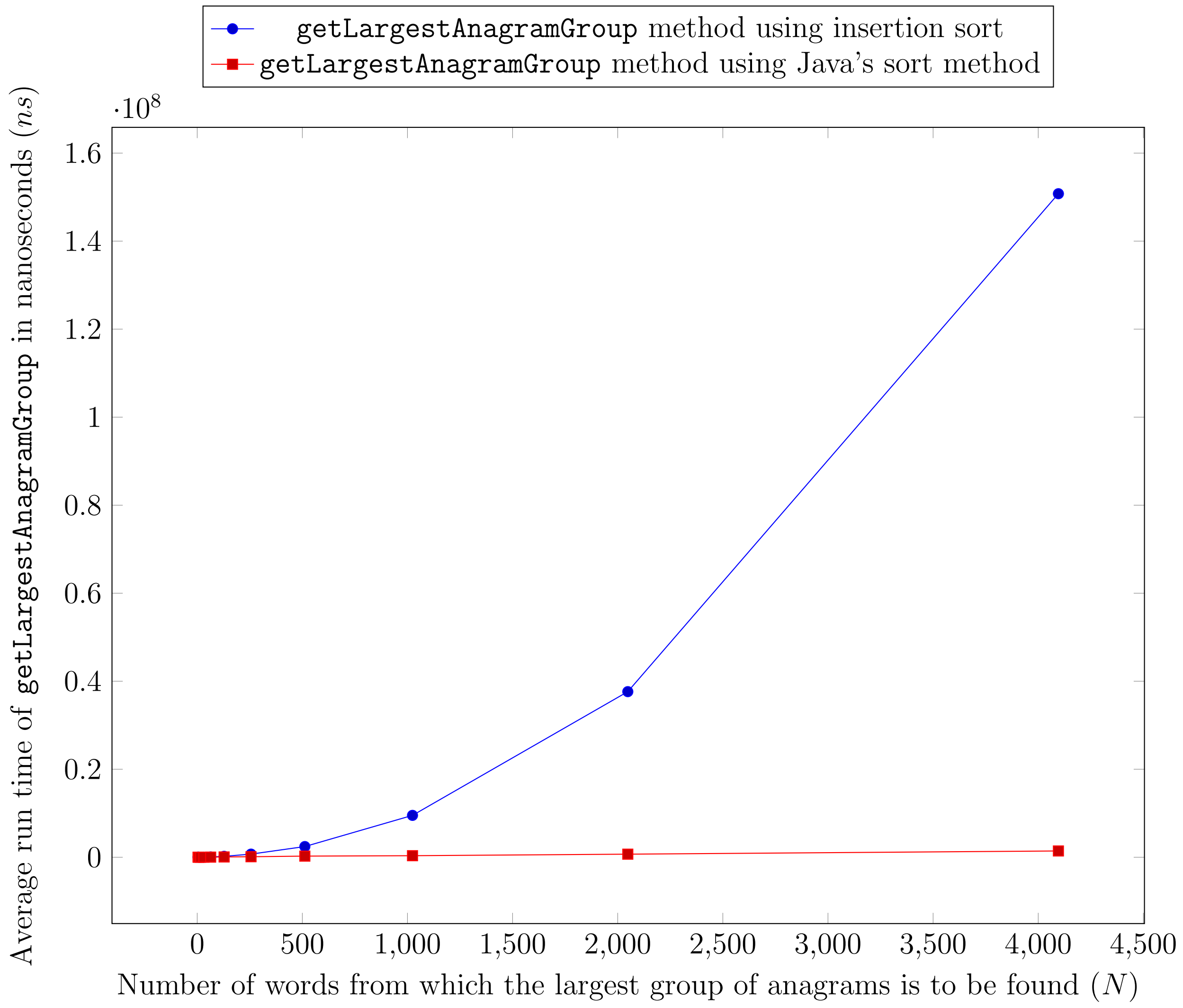

Figure 4: Average run time of `getLargestAnagramGroup` using insertion sort and Java's sort method.