Assignment 11 Analysis Document:

1. The first problem we needed to solve in this code was reading the input file and figuring out how to read each line of the document. First, we created a main method in our RandomPhraseGenerator. Inside the main method, we called the RandomPhraseGenerator constructor and passed in args[0], the parameter from the main method. Args[0] contains the filename that the user passed in. The constructor takes in that filename, and creates a new file with it. From there, we take that file and create a Scanner to scan through the file. This scanner is called fileReader, and is stored in an instance variable of the RandomPhraseGenerator object. This file and scanner initialization is surrounded by a try/catch block, so that if we encounter a FileNotFoundException our code catches it and prints out the stack trace.

   After we open the file and create a scanner to read it, we call the generateMap() method. This method is only called once, when we are creating a new RandomPhraseGenerator object. This method uses the scanner to scan through every line in the file and store all non-terminal definition lines (indicated by <>) as a key in the hashMap instance variable, and it stores all subsequent lines in the associated value, an ArrayList<String> that stores all possible production rules for that non-terminal definition. The production rules may contain non-terminals, but these phrases are still contained in the ArrayList<String>, as we have a separate method that deals with production rules containing non-terminals. To ensure the hashMap doesn't include any comments contained in the file, we used a while loop that runs until the current line is a non-terminal. We called a separate method called isNonTerminal() to check the line. This helper method returns true if the first character of the line is "<" and it does not contain any whitespaces until after it reads the character ">". If it is a properly formatted non-terminal definition, then it calls map.put() to put the current line as a key in the map, and creates an empty ArrayList<String> as the value for that key in the map. It then returns true or false to the generateMap() method. Once we reach a non-terminal line, we get the ArrayList<String> associated for that non-terminal in the map, and for every single line after that non-terminal (and before the next curly brace "}") we added it to the ArrayList. After that, the loop repeats for the next non-terminal, and the next, etc. until we reach the end of the file.

   Once the constructor has generated the map, the RandomPhraseGenerator construction is complete. After we have called the constructor in the main method, we then have a for loop that runs from 0 to the value of args[1], which is the number of phrases the user wants generated. In each iteration of the for-loop, the generator calls generatePhrase() and then prints out the value contained in the instance variable called phrase.

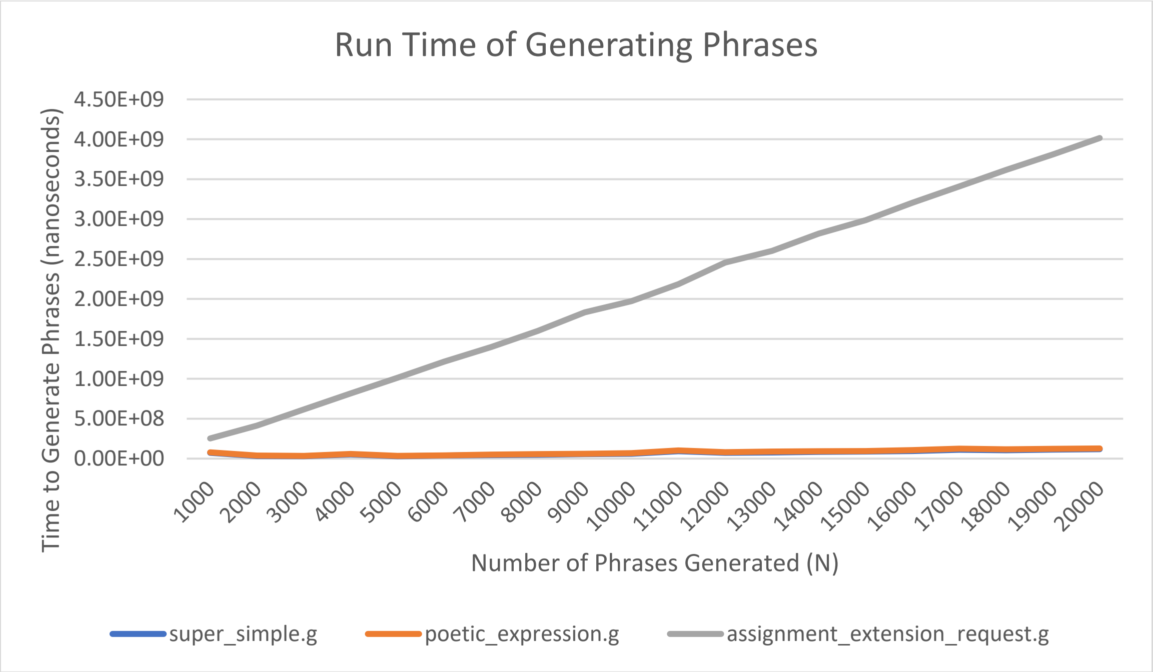
   generatePhrase() is the method that actually generates the phrase. It begins by initializing the StringBuilder instance variable called phrase that contains the generated phrase. We used StringBuilder instead of String so that we could actually alter the String while creating it. Additionally, we create a random object that will take care of generating a random integer, to be used later in the method. We select the overall phrase structure by getting the ArrayList<String> associated with the key "<start>" in the map, and then getting a random value contained in the list by calling get(random.nextInt(0, arrayList.size()). We then split that value by two characters, < and >, and store that in a String[] array. This is to isolate the non-terminals so we can expand them until they contain only terminals. We use a for-loop to loop through the entire String[] array (called phraseStructure). For each value, we check if it is a non-terminal by checking the beginning character of the very next value in the array. If the first character in the next value in the array is >, then we know that our current value is a non-terminal. We also check to make sure i isn't the last value in the array, because if i is the last value, then we know it can't be a non-terminal either, because it doesn't have the > in the next value. If the value is a terminal (i.e. doesn't meet the previous condition) then we delete the <> characters out of the string, and append it to our phrase instance variable if it actuallycontains a value (i.e. isn't just a blank space, we do this to ensure we don't add extra whitespaces). If it is a non-terminal, we get the ArrayList<String> from the map that contains the production rules for that non-terminal, select a random production rule from that ArrayList<String>, and send that production rule to the helper method called generateSubPhrase().

This helper method takes in a production rule from a non-terminal and generates a subphrase from that, calling itself recursively until there are no more non-terminals in the subphrase. To do so, we split the subphrase by <> to isolate the non-terminals. We follow a similar structure to the generatePhrase() method above. We loop through each value in the split subphrase array, and each value is checked (like above) to see if it is a non-terminal. If it is a terminal, then we delete the <> characters and add it to the 'completedPhrase' String that we created in this method. If it is a non-terminal, we select one of it's production rules and call generateSubPhrase() with that new production rule as the subphrase. At the end of the for-loop, we return the completedPhrase from index 0 to the second to last index, to ensure we remove any extra whitespace at the end. The base case for this method is when we reach a subphrase that is only a terminal. When all iterations of the recursively called generateSubPhrase() finish, we return the 'completedPhrase' value and send it back to generatePhrase(), where we continue the process for the rest of the values contained in the phraseStructure array. Once that completes, the phrase is printed out in the main method.

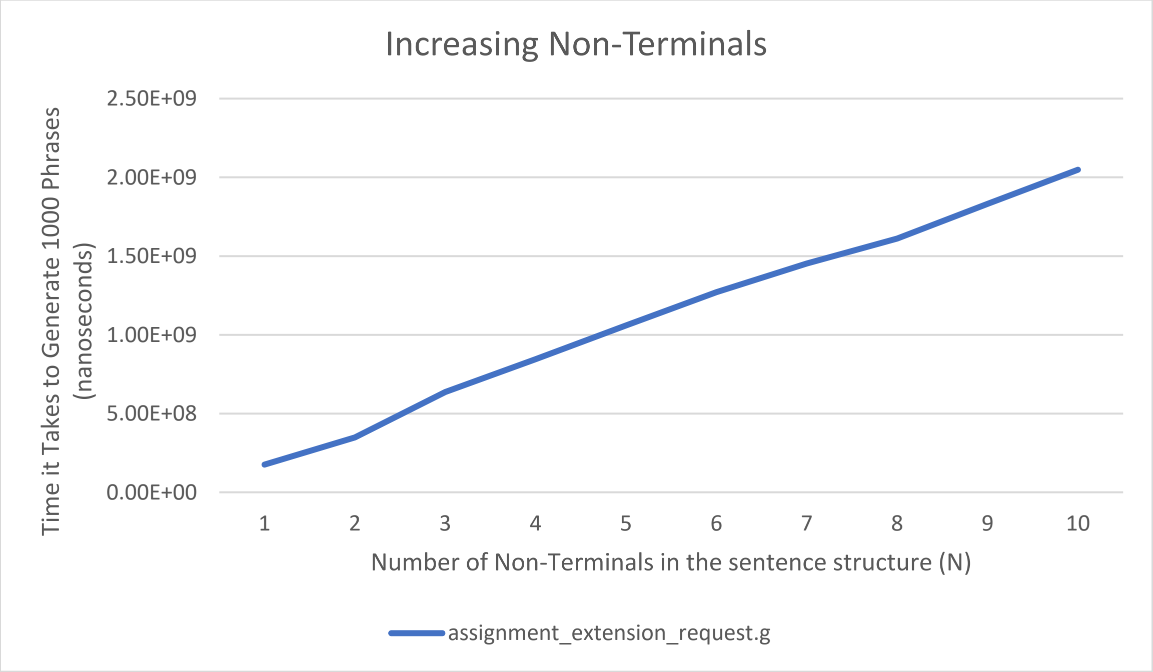
We used a hashMap to store the non-terminals and their production rules because that way, we only had to loop through the file once, and from there we could access all production rules for a non-terminal in constant time, and make a selection quickly. It cut down on our cost a lot. We also used an ArrayList<String> instead of a different data structure (like a LinkedList) because we can access any random value in the ArrayList in constant time. We used a StringBuilder to store our phrase because that way, we could get rid of the <> characters after we generated the phrase, instead of deleting them from the beginning and having to deal with complex methods of deciding if a phrase was a non-terminal or not. The Scanner was an efficient method of looking through every line in the file because it was straightforward, and we knew we wouldn't miss any lines because we could loop through each line using scanner.nextLine(). Each data structure was selected to make the simple accessing/adjusting processes as simple as possible, and as efficient as possible.

The only class we created was the RandomPhraseGenerator class. We did not need any extra classes that we created. We used Java's random class, the HashMap class, the Array and ArrayList class, the StringBuilder and String classes, and the Scanner and File classes. We also used the FileNotFoundException class, but only in the event that the filename passed in was invalid.

2. Run Time for generating increasing number of phrases: The first experiment I ran was to test how long it took to generate an increasing number of phrases, starting from 1000 and going to 20000, increasing by 1000 each time. I generated phrases using the structure provided in the assignment_extension_request.g as it has a medium amount of terminal vs. non-terminal production phrases, and it is a medium-sized file. I also did experiments using super_simple.g and poetic_expression.g, as these two files represent the smaller file, and then a middle sized file respectively.



Run time for increasing number of non-terminals in the grammar structure sentence: For the next experiment, I decided to test how long it took to generate 1000 phrases for an increasing number of non-terminals in the grammar structure sentence. To test this, I used the assignment_extension request, and for each number 1-10, I added another <plea> to the structure. This way, I can ensure that the production rules are the same across the experiments so it is fair, but I am still testing an increasing number of non-terminals in the structure. So, for example, for N=1, the sentence structure looks like "I need an extension because <plea>", for N=2, the sentence structure looks like "I need an extension because <plea>, and <plea>", and so on until I reach 10.



3. Experiment 1: I expect this experiment to have a complexity of O(N), or in other words, for it to be linear. In this experiment, N is the number of times we are generating a phrase. I expected this experiment to be linear because our implementation uses a for loop that runs N times, and calls generatePhrase() each iteration of the loop. This clearly has a complexity of O(N), and because I expect that generatePhrase() is relatively consistent in its own time, it makes sense that when we increase N, the time will increase accordingly. The time it takes for generatePhrase() to run is not impacted by N in this case, because we don't change the structure of the input file, and I kept the file consistent for each test. Thus, the only change in complexity would come from the for-loop itself, and obviously, that would be a linear complexity.
   Experiment 2: I also expected this complexity to be linear, as well. In this case, N is the number of non-terminals contained in the sentence structure rule (the sentence after <start>). To ensure I was only testing the increase of time vs. the increase of non-terminals, I made sure that there was only one option for the sentence structure, and I kept the non-terminal that I was increasing consistent. This way, it wasn't due to a difference in the number of production rules for the added non-terminal, a difference in sentence structure, etc. I expected this experiment to have a complexity of O(N) because despite the potential variations between the generated phrases (different production rules get selected, etc), the time it takes to generate a phrase for <plea> will be relatively consistent. It will be relatively consistent each time because I ran each phrase generation five times, and took the average of those tests to account for potential variation in production rule selection, etc. Thus, the only thing that N impacts is the number of times we call <plea>, and if the time it takes to generate a phrase for <plea> stays consistent (as explained above), then we are simply adding that time N times for each phrase

generation. Thus, it makes sense that when N increases by one, we would increase by 1 unit of time (unit being the time it takes on average to generate <plea>), and thus that would result in a linear complexity, or a big-o complexity of O(N).

4. I don’t think that our phrase generation is as efficient as it possibly can be, because we are still beginner programmers and there may be some data structures or implementations that were more efficient. For example, maybe it would have been more efficient to only add non-terminals that we know will be used to the map, to account for the possibility that there’s a non-terminal contained in the file that may never get used. Maybe there was a more efficient way to expand the phrase until there were no more non-terminals contained, etc. Additionally, it could be that there are some small places that could have been optimized, such as storing some values in a variable instead of calling for them each time, or maybe getting rid of a variable that isn’t used very often to optimize memory needed. All of these optimizations come with programming experience, so although we wrote it as efficiently as we could, I’m sure there are still ways that advanced programmers could have optimized it. I do think that it is pretty efficient in general, though.
5. I think our program is designed and written well. We made sure to include plenty of comments to guide anyone looking over the code. We also made sure to include descriptive Javadoc comments to explain the purpose for each method, and we did our best to name each variable so that it would describe the data it was containing. If I were to do it over again, I would rename a few variables to sound less similar, so that it would be more clear what their purpose is. Additionally, I would add a few more helper methods to reduce large chunks of code to a more concise method call. Finally, I would try to get rid of our generateSubPhrase() method entirely, and find a way to have the function of that method included in our generatePhrase() method. I believe there was a way we could have used generatePhrase() recursively to expand the phrase until there were no non-terminals, and thus we would not have needed a generateSubPhrase() method. However, I think that our program is pretty efficient and well written overall, so I wouldn’t change much about the core structure of our code. I think the data structures we used were efficient, and I think our implementation was straightforward and didn’t have many unnecessary functions.