For my TrieNode class, I designed so that each TrieNode contains a HashMap called children that takes in a Character as its key and another TrieNode as its value. The TrieNode class also contains a Boolean called isWord to signal at the TrieNode if a complete word has been reached. Also, there’s a String variable called prefix which builds as you iterate through the Trie. So, once you have reached at the of the word, prefix would contain a complete word and isWord would be set to true. Looking at the constructor of the TrieNode class, we initialize the root TrieNode and use the insert(String word) method to add every word from the dictionary argument into the Trie (this is to simulate that the dictionary is already inputted and the Trie has been created). Then, the insert(String word) takes in a String which is a word (from the dictionary) and loops till the word’s length using a looping variable (TrieNode starting from the root). If the root’s HashMap doesn’t contain the certain character of the word, it puts the character in as the key and creates a new TrieNode and inputs that as the value. Then for that new TrieNode, prefix is updated to contain the letters that been inputted so far as part of that word. When it is done linking the characters in the Trie (looping through the word’s characters), the last TrieNode’s isWord is set to true to signal that a valid word has been reached. In the search(String searchString) method, it first sees if the searchString’s characters (in that linking order) are in the Trie. If they are not, an empty list is returned. Otherwise, the current TrieNode (which is the TrieNode that is the value of last key or letter of the searchString) gets inputted into the recursive function findAllChildrenWords(List<String> result, TrieNode current). This function then checks if the current TrieNode is a word; if it is, add it to the list. Then it recurses for each key set in the HashMap of that TrieNode to see the various words that come up that follow the searchString. This is the best way to implement the autocomplete because it beats string comparison. Instead of just going through each word in the dictionary, the TrieNode eliminates one part of the tree if the word(s) wouldn’t exist there. This can be reflected in the time complexities. The time complexity for the String Comparison is O(n \* k), where n is the length of the dictionary and k is the length of the searchString. Time complexity of the Trie Search is O(c\*k), where c is the number of word solutions and k is the length of the searchString. The runtimes can also be viewed in the data file I have included (which also includes a graph). I’m not the best when it comes to space complexities, but I think for both Trie Search and String Comparison it is O(n). After going through the recursive method calls for the Trie Search, the memory allocation would just get popped off the memory stack, so I think it is O(n) at the end.