My initial instinct when seeing the autocomplete problem was to iterate through every word in the dictionary and then for each word, checking if the word started with the input String to verify that the word can be a valid autocomplete. This is obviously not a very efficient solution because it requires the algorithm to iterate through an entire list of words(potentially millions of words), then iterating the letters of each word. As described in wednesday's lecture, the big O time complexity would be considered O(n * c), where n is the number of words in the dictionary and c is the amount of letters to iterate through in each word(length of input). This would be a linear trend, however, the slope of the trend can be increased greatly, proportional to the amount of words in the dictionary and the length of the input, and if you have around a million words to iterate through, this will be greatly inefficient.

The solution presented by wednesday's lecture and the solution I decided to implement was the Trie data structure - recursive backtracking solution. This solution works by first inserting every word in a dictionary into a trie structure - a tree structure which is similar to a binary tree, but instead of each node having 2 child nodes, each trie node holds an array of child trie nodes. The insertion of a word works like this: we take our word and iterate through each character. As we iterate through each character, we add the current character to the trie structure. The trie structure consists of a series of nodes, each holding a set of "children" chars. The parent-child relationship represents the letter-to-next-letter relationship within a word. Each successive letter in a word is a child of the previous letter. With "hello" the algorithm takes the first letter "h" and adds it to the root(head) node's children array.

In order to simplify letter search a later part in the algorithm, I designed the children array to work kind of like a map: I designated each letter in the alphabet to a specific index in the array through the use of ascii conversion. In the trie node class I have an arraylist instance variable(separate from the children array) which holds the indexes of the available letters in the children array. Whenever I add a new letter to my children array I assign it to its corresponding index and add that index to my arraylist of indexes. With "h", the letter h gets assigned as a trienode in index 7 in the array, and the arraylist of indexes now holds 7.

Now, moving on to letter "e" in "hello", we assign e by following the same process: we go deeper into the trie structure by assigning e as a child of the previous letter trie node h. We continue this process until the word is complete, and the whole dictionary's set of words have been added to the trie structure. Since each word and its letters had to be iterated through to produce this trie structure the time complexity is O(n) where n is the total number of letters added to the trie. The complexity trend is

linear but proportional to the amount of words in the dictionary, so it would still cost some time for this action to happen. It is all worth it, however, because the actual autocomplete word generation complexity can now be improved. Even though the Trie construction cost some time, the act of construction does not happen whenever a user searches a word, so we can kind of ignore it since the user will not experience it.

Now, to the actual auto-complete word generation algorithm. The algorithm works by taking an input and iterating through each letter. As we iterate through each letter, we also iterate through the word's letters in the trie structure. We want to iterate until we iterate through the whole word. For example, if our input was "sea", we would simply iterate through the letters, s -> e -> a within our trie structure.

This action takes linear time O(n), where n is the amount of letters in the word. Once we've gotten to this point in the trie structure, we must use recursive backtracking to seek out all possible words/paths in the trie from this point. Using our "sea" input example, our recursion would branch into multiple paths, one going to "attle" and another going to "tac".

The code works by recursively taking a current trie node and iterating through each child/succeeding letter and collecting those letters into a result String. Once we reach an ending point of a word(which will be indicated by a boolean value instance variable within the letter trienode), we add that result String to our final list. Here, the complexity is O(n), where n is the amount of letters that can autocomplete the input.

While the complexity is still linear, like the simple brute force-iterate-through-a-million-words algorithm, the amount of items, n, is ***greatly*** reduced, because the algorithm only iterates through the words that can autocomplete our input, and isn't interested in the thousands, or millions of other words that can't autocomplete the input.