In Part C, you must create a pdf format document called written-tasks.pdf, which explains the impact of only allowing each letter to appear once in each word (regardless of how many times the letter appears on the board). Your answer must state an upper-bound on the time complexity reflecting the impact of this change, with each term used explained clearly.

In order to avoid trivial answers, you must assume the board could be extended arbitrarily to higher dimensions (e.g. 5x5 and beyond) and that the alphabet used could increase in size (e.g. the maximum length of a word is not 26 letters).

Notation:

W	total number of words in the dictionary
L	average length of words in the dictionary
M	dimension of the boggle board

Time complexity and space complexity of part 2a:

In part 2a, I used functions newPrefixTree and addWordToTree to construct the prefix tree for the given dictionary, the cost for contructing the prefix Tree is

$$\Theta (W*L)$$

I get this because for each word in the dictionary, we need to do a look up in the tree for each letter in each word, if the prefix is not pre-existed, then we create a new node for the new letter. On average, we do L looks up for each word. In other words, we scan every character in the dictionary. Therefore, constructing the tree has time complexity of Θ (W*L). The space complexity is hard to determine as it can be hard to determine how many nodes needed to be created for the dictionary, but if we know the total number of nodes in the trie (N), then space complexity for storing the prefix trie is O((257)*N) as every node can store at most 256 pointer and an integer that indicates whether the node is at end of a word.

After constructing the tree, I then used function findWordsInBoggle to construct possible words using letters in the given boggle board while checking against with the constructed trie tree. This function calls another function newFindWordForChar for M^2 times to scan every letter in the board.

The time complexity of constructing possible words in a given boggleboard while checking against with the constructed trie tree would be $O((M^2)*8*7^L)$

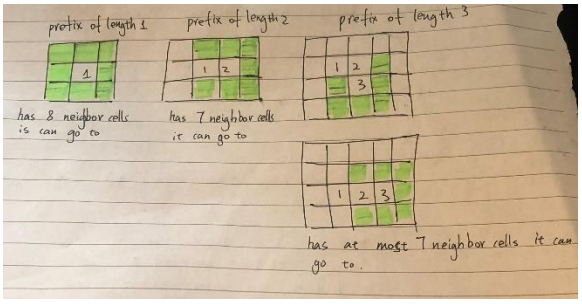
This is because each cell in the boggle board has potential to be a starting letter in a word, so we scan each cells to let it be the initial prefix of a possible word.(which is where the M^2 comes from in the time complexity). To find the upper found, we assume that the prefix path will keep expanding in the board, as long as the prefix exist in the prefix trie. This process of expanding the prefix path is by first scan the board to find a possible starting cell, the letter in this cell will become our current prefix, then since each cell would have at most 8 neighbour cells, we will check through every neighbour cell to see whether it can continue our current prefix in the trie. To find the upper bound, we assume the new prefix constructed with every neighbour cell exist in the prefix trie, and this

applied with every starting cell in the board which result in ($8*M^2$) complexity. Then we keep expanding the prefix trie, the directions that our current prefix(with length 2) can go is at most 7, and the next direction our prefix (with length 3) can go is also at most 7 …. We keep expanding the prefix until we reach the end of the word. Also, for every prefix constructed, we do a search in the dictionary prefix trie to check whether it is a word and this take O(W*L) time complexity. Therefore, time complexity in searching the board while check against with the dictionary in the worst case is $O(8*M^2*7^L*W*L)$

Where M^2 is the cost of scanning every character in the board as we consider that every character can be a starting letter of a word.

$8*7^L$ is the cost of depth first search and forming prefix in the board

W*L is the cost of searching the constructed prefix in the dictionary prefix trie



Finally, before outputting the words I found, I need to convert the trie that I used to store the found words to an array of strings which takes O(W*L) as we need to traverse every node in the trie, and the maximum number of nodes is W*L (total number of letter in the dictionary). I done this by using the function storeAllWordsInTree.

Part 2d time complexity

Notation:

L	Average length of words in the dictionary
M	Dimension of the boggle board
W	total number of words in the dictionary
Q	average frequency of characters in the boggle board

In this part, I check whether words in the dictionary exist in the boggle board, with a new rule that only word that is unique can be checked whether it exist in the board or not. Therefore, output words will not contain any not-unique words. The definition of unique is that the word must be constructed by unique letters, for example, bee is not a unique word, as e appears twice in the word.

This should improve the time complexity from part a, as instead of constructing a prefix trie for the dictionary, and checking the dictionary prefix trie while doing depth first search, in part d, we don't construct the dictionary prefix trie at all, but scan every word in the input dictionary, and check whether the word exist in the board, if it does, then store it.

In storing the output words, same as in part a, I will use a prefix trie to store output words, and latter convert it to an array of words.

The expected upper bound of the time complexity of searching unique words in the boggle board is O(W*L)

The N in the time complexity represents that we need to check whether a word in the dictionary is unique, the checking process is by setting up an int array that can store 256 pointers each represent an ascii Index of a character, and for every character of the word, if its ascii Index represents its position in the array is taken up, then it is not a unique word. Hence, in worst case, we check every character for every word in the dictionary which is W*L.

Then for every unique word in the dictionary, we scan every character of the word, starting with the first character in the word, we then search in the board for this first character, where could be multiple instances of this character, we search through every instance, but return as soon as we had found the word. Let say we found the first character in the position board[i][j], then we search every neighbour cells of board[i][j], but different from part1a, instead of form a new prefix with every neighbour and check whether the new prefix exist in the dictionary prefix trie. In this part, I will check whether the neighbour cell contains the next character of the word, if not, then return to the next neighbour cell.

Therefore, if there are zero instances of a starting character exist in the board, then we do at most M^2 scan as every character in the board does not match with the starting character. And if the word contains n characters, and there are q instances of the starting character exist in the board, for every instance of the q instances result in a prefix that contains (n-1) character matches with the word, this means the word does not exist in the word. Hence, when scanning the q instances of starting characters, we scan their neighbour cells along the path but will never find and return the word. This makes the time complexity of searching in worst case become $O(W*L + W*M^2*Q*7^{n-1})$.

Where W*L is the cost of checking whether a word in unique

And $W*M^2$ is the cost of searching for a word in the board

$Q*7^{n-1}$ is the cost of searching for a path of a word in the board which is depending on the frequency of the starting character of the word in the board, and the maximum cells that will be visited from the starting characters in the board is 7^{n-1}