212. Word Search II

Finding words in a 2D board can be efficiently solved using a combination of Trie (also known as a prefix tree) and Backtracking techniques. This approach leverages the advantages of data structures and algorithms to search for words on the board while minimizing search space.

Problem Understanding:

Given a 2D grid (board) of characters, we are tasked with finding all the words from a list that can be formed by sequentially adjacent letters on the board. The adjacent letters can only be horizontally or vertically neighboring, and each cell can only be used once while forming a word. This constraint of using each cell once ensures that we must carefully track the cells we visit during the word formation process.

Trie for Efficient Word Search:

A Trie is a specialized tree-like data structure that is particularly effective for storing strings, where each node represents a single character of the string. The Trie is well-suited for problems involving prefix matching. In this context, we can insert all the words from the given list into the Trie, which allows for efficient prefix lookups while traversing the board.

By storing the words in a Trie, we can quickly check whether a prefix exists in any of the words. This reduces unnecessary search space by terminating early if no valid word can be formed from a given prefix. Each node of the Trie either contains references to its child nodes or stores the complete word at the end of its path. This makes it easier to track whether we have formed a valid word as we navigate through the board.

Backtracking for Word Exploration:

To search for words on the board, we use a Backtracking approach. This method allows us to explore all possible paths starting from each cell in the grid, while checking whether the characters we encounter can form a word present in the Trie. The idea of backtracking is to explore one possible solution at a time and backtrack as soon as we realize that the current path does not lead to a valid solution.

Starting from any cell, we recursively attempt to move to its neighboring cells (horizontally or vertically). At each step, we check whether the current prefix formed by the characters matches any prefix in the Trie. If we find a match, we continue exploring; otherwise, we backtrack and try another path. To ensure that each cell is used only once in a word, we temporarily mark the cell as visited during the backtracking process.

Finding Complete Words:

During the backtracking process, as soon as we form a valid word that exists in the Trie, we add it to our result set. We also remove the word from the Trie once it has been found to prevent duplicate entries. This helps in optimizing the search, as we don't need to search for the same word again.

Pruning the Trie:

An important optimization step involves pruning the Trie as we explore the board. Once a word has been found, we can remove the corresponding path in the Trie if it no longer leads to any other words. This pruning helps in reducing the size of the Trie and eliminates unnecessary lookups during the backtracking process.

Challenges and Edge Cases:

There are several edge cases to consider. The board might be too small to fit any of the words, or the words might contain characters that do not exist on the board at all. Additionally, if the board contains many repeated characters, the algorithm needs to handle this efficiently by carefully exploring all possibilities while avoiding redundant checks.

Time and Space Complexity:

The overall time complexity involves two main operations: building the Trie and performing the backtracking search. Constructing the Trie requires processing all the words in the list, making the time complexity for this step O(W L), where W is the number of words and L is the average length of each word.

Backtracking involves traversing the board starting from each cell. In the worst case, for each cell, we explore up to 4 possible directions, and this exploration can continue for as long as the length of the longest word. Therefore, the time complexity for backtracking is O(M N 4^L), where M and N are the dimensions of the board, and L is the maximum length of any word.

The space complexity is mainly determined by the size of the Trie, which is O(W L), and the recursion stack depth for backtracking, which can go up to O(L).

Conclusion:

By using a combination of Trie and backtracking, this solution efficiently finds all the valid words from the list on the board. The Trie allows for quick lookups and pruning of invalid paths, while backtracking enables exploring different paths in the grid. This approach balances between exploring all possibilities and minimizing unnecessary work by terminating early when no valid word can be formed. With careful attention to pruning and backtracking, this method handles both small and large boards efficiently.