Solution: Word Search II

Let's solve the Word Search II problem using the Trie pattern.

We'll cover the following

Statement

You are given a list of strings that you need to find in a 2D grid of letters such that the string can be constructed from letters in sequentially adjacent cells. The cells are considered sequentially adjacent when they are neighbors to each other either horizontally or vertically. The solution should return a list containing the strings from the input list that were found in the grid.

Constraints:

- $1 \leq$ rows, columns ≤ 12
- $1 \leq$ `words.length` $\leq 3 \times 10^3$
- $1 \leq$ `words[i].length` ≤ 10
- All the strings are unique.

> **Note:** The order of the strings in the output does *not* matter.

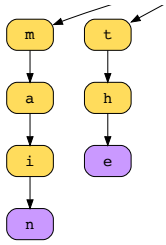
Solution

By using backtracking, we can explore different paths in the grid to search the string. We can backtrack and explore another path if a character is not a part of the search string. However, backtracking alone is an inefficient way to solve the problem, since several paths have to be explored to search for the input string.

By using the trie data structure, we can reduce this exploration or search space in a way that results in a decrease in the time complexity:

- First, we'll construct the `Trie` using all the strings in the list. This will be used to match prefixes.
- Next, we'll loop over all the cells in the `grid` and check if any string from the list starts from the letter that matches the letter of the cell.
- Once an letter is matched, we use depth-first-search recursively to explore all four possible neighboring directions.
- If all the letters of the string are found in the `grid`. This string is stored in the output `result` array.
- We continue the steps of all our input strings.

z	i	a	f
h	n	l	s
e	g	m	w



Given the 2D grid and input list ["main", "the"], add the input list strings in the Trie.

1 of 50

Let's look at the code for this solution below:

Java

main.java

Trie.java

TrieNode.java

```
import java.util.*;

class WordSearch {
    public static List<String> findStrings(char[][] grid, String[] words) {
        Trie trieForWords = new Trie();
        List<String> result = new ArrayList<>();

        // Inserting strings in the dictionary
        for (String word : words) {
            trieForWords.insert(word);
        }

        // Calling dfs for all the cells in the grid
        for (int j = 0; j < grid.length; j++) {
            for (int i = 0; i < grid[0].length; i++) {
                dfs(trieForWords, trieForWords.root, grid, j, i, result, new
            }
        }

        return result;
    }

    private static void dfs(Trie wordsTrie, TrieNode node, char[][] grid, int
```

```
            node.isString = false;
            // remove the characters in the word that are not shared
```

Word Search II

Time complexity

The time complexity will be $O(n * 3^l)$, where n is equal to rows*columns, and l is the length of the longest string in the list. The factor 3^l means that, in the `dfs()` function, we have four directions to explore initially, but only three choices remain in each cell because one has already been explored. In the worst case, none of the strings will have the same prefix, so we cannot skip any string from the list.

Space complexity

In the worst case, none of the strings will have the same prefix, so the space used by the trie data structure will be $O(m)$, where m is the total count of all the characters in all the strings present in the input list.

If there is a string of length equal to the length of the grid and the elements on the grid are arranged in such a way that the string can be traced, then the space acquired on the stack to store the recursive calls of `dfs()` function would be the size of the grid, which is $O(n)$.

Hence, the overall space complexity of the solution will be $O(m + n)$.

← Back
Word Search II

Next →
Lexicographical Numb...

Mark as Completed