**212. Word Search II**

<https://leetcode.com/problems/word-search-ii/>

1. **Listen**

**Problem Statement:**

Given an m x n board of characters and a list of strings words, return *all words on the board*.

Each word must be constructed from letters of sequentially adjacent cells, where **adjacent cells** are horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

**Input:**

char[][] **board**: m x n grid of characters

String[] **words**: list of **words** that may or may exist in **board**

**Goal:**

Determine what words from the **words** array exist in **board**.

**Return:**

return ***all words on the board***.

1. **Examples**

**Example 1:**

Calendar

Description automatically generated

**Input:** board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]], words = ["oath","pea","eat","rain"]

**Output:** ["eat","oath"]

**Example 2:**

Calendar

Description automatically generated

**Input:** board = [["a","b"],["c","d"]], words = ["abcb"]

**Output:** []

**Constraints:**

* The **word** can be constructed from letters of **sequentially** **adjacent** cells, where adjacent cells are horizontally or vertically neighboring.
* The same letter cell **may not be used more than once**.
* **board** and **words** consist of only lowercase English letters.
* All the strings of **words** are unique.
* m == board.length
* n == board[i].length
* 1 <= m, n <= 12
* 1 <= words.length <= 3 \* 104
* 1 <= words[i].length <= 10

**Test Cases:**

* word is in board
* word is not in board
* board is empty
* word is empty string

1. **Brute Force**

**Solution 1: Backtracking**

**Intuition:**

We can use a similar solution that we used in Word Search I.

We can run a dfs search on each cell of the board.

Each time we call the dfs function, we check to see if we have found a word in the board that matches one of the given words.

Therefore, we could solve the problem the same way, except we need to add a factor to account for the fact that we now have a list of words, rather than just a single word.

**Runtime:**

The runtime of Word Search I was O(M \* N \* 4^L).

Where M and N are the dimensions of the board. We traverse over every cell in matrix, and run dfs for each cell.

Where L is the length of the word. Each time we visit a new cell in the board, we recursively call dfs in four different direction. Therefore, we have a branching factor of 4 each time we call dfs. This continues until we reach the end of the string, which has length L.

The runtime of Word Search II is O(W \* M \* N \* 4^MN).

Where M and N are the dimensions of the board. We traverse over every cell in matrix, and run dfs for each cell.

Except now, we have to do an M\*N search over the entire board, while checking for W possible different strings. We run a M\*N for each string in the list of length W, therefore making it a WMN for all strings in the words list.

Where before L was the length of the single word, we now have to find all possible words in the board. Before, we had to only look for the single word, and stopped if we found it. Now, we are looking for many possible words, so we must traverse over the entire matrix. Each time we visit a new cell in the board, we recursively call dfs in four different direction (branching factor of 4) until we reach the end of the board, which has length MN.

Therefore, the runtime is O(WMN \* 4^MN)

**Space:**

The space complexity is O(MN) where MN are the dimensions of the board. This comes from the recursion – we will at most have MN frames on the stack when we run dfs on the first cell in the board.

1. **Optimize**

**Solution 2: Backtracking with Trie**

We can get rid of the W constant by trading the extra time of looking if any words from the words list exist in the graph on every dfs call, by storing a Trie and HashSet of already visited words. In this way, we are trading time for space, making our space complexity MN + W.

1. **Walkthrough**
2. **Implement**
3. **Test**