**79. Word Search**

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

1. **Listen**

**Problem Statement:**

Given an m x n grid of characters board and a string word, return true *if* word *exists in the grid*.

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.

**Input:**

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

String **word**: **word** that may or may exist in **board**

**Goal:**

Determine if **word** exists in **board**.

**Return:**

return true *if* *(****word****exists in the grid)*

1. **Examples**

**Example 1:**

Calendar

Description automatically generated

**Input:** board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"

**Output:** true

**Example 2:**

Calendar

Description automatically generated

**Input:** board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"

**Output:** true

**Example 3:**

A picture containing text, crossword puzzle

Description automatically generated

**Input:** board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"

**Output:** false

**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 **word** consists of only lowercase and uppercase English letters.
* m == board.length
* n = board[i].length
* 1 <= m, n <= 6
* 1 <= word.length <= 15

**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 run a dfs search on each cell of the board. Each time we call the dfs function, we check to see if it matches the current index of the given word.

The problem description states that we can only visit sequentially adjacent cells:

1. UP
2. DOWN
3. LEFT
4. RIGH

We call dfs to go in each of the four directions from the current cell.

1. row-1, col
2. row+1, col
3. row, col-1
4. row, col+1

This matches our usual definition of a **Constraint** in backtracking:

* *Choose*: Choose the potential candidate. Here, our potential candidates are the four directions in which we can travel.
* *Constraint*: Define a constraint that must be satisfied by the chosen candidate. In this case, the constraint would be that current cell char matches the current index of the word.
* *Goal*: We must define the goal that determines if have found the required solution and we must backtrack. Here, our goal is achieved if we have reached the end of the string.

**Runtime:**

The runtime of this algorithm is 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.

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

**Space:**

The space complexity is O(L) where L is the length of the word. This comes from the recursion – we will at most have L frames on the stack if we have found the word.

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