Word Search(Day 18)

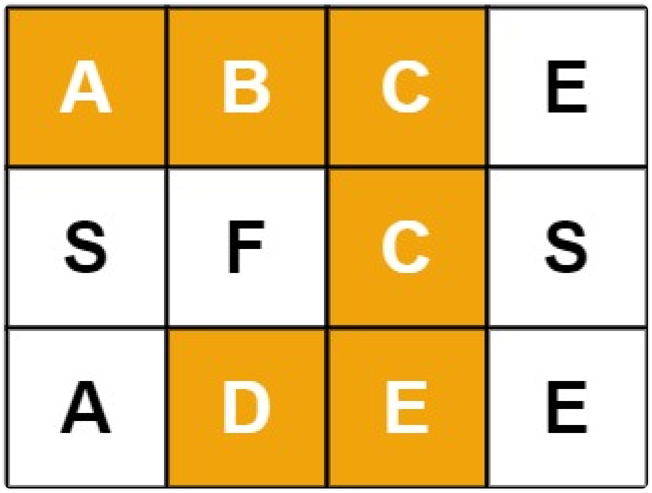
Prepared By Azan Imtiaz

Word Search (Leatcode)

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

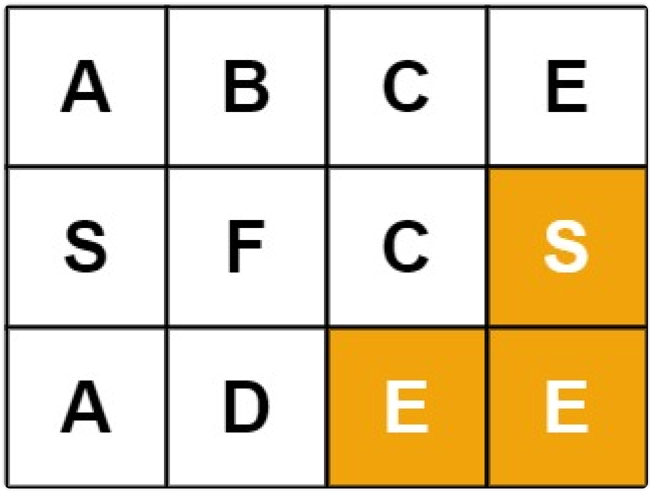
The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example 1:



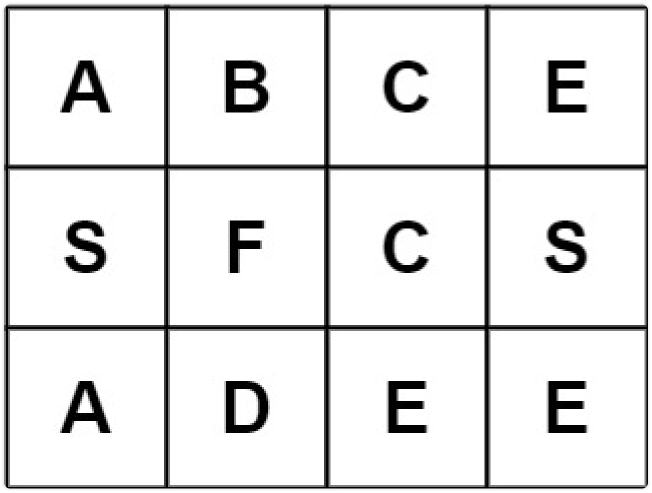
```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"
Output: true
```

Example 2:



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

Example 3:



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

Constraints:

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

Intuition

The problem is to determine if a given word exists in a 2D grid of characters. The word can be constructed by starting at any cell in the grid and moving in any of the four cardinal directions (up, down, left, right). Each cell in the grid can be used only once for constructing the word.

To solve this problem, we need to explore all possible paths in the grid starting from each cell. For each starting cell, we perform a depth-first search (DFS) to see if we can construct the word by moving through adjacent cells.

Approach

1. **Start at Each Cell**: Iterate through each cell in the grid.
2. **Perform DFS**: For each cell, if it matches the first character of the word, start a DFS to see if the rest of the word can be found.
3. **Check Boundaries**: During DFS, ensure that you stay within the boundaries of the grid.
4. **Mark Visited Cells**: Temporarily mark cells as visited to avoid revisiting them during the DFS.
5. **Restore Cells**: After exploring all possible paths from a cell, restore its original value to allow other paths to use it.
6. **Return Result**: If any DFS call finds the complete word, return `true`. If no such path is found after exploring all cells, return `false`.

Solution

The solution uses a recursive DFS approach with backtracking. The `exists` method performs the DFS while the `exist` method initiates the search from each cell in the grid.

```
java
Copy code
class Solution {
    public boolean exist(char[][] board, String word) {
        char[] word_arr = word.toCharArray();

        // Iterate through each cell in the gridfor (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                if (board[i][j] == word_arr[0] && exists(board, i, j, word_arr, 0)) {
                    return true;
                }
            }
        }
        return false; // Return false if the word is not found
    }

    private boolean exists(char[][] board, int i, int j, char[] word, int index) {
        // Check boundaries and if the current cell matches the current character in the
wordif (i < 0 || i >= board.length || j < 0 || j >= board[0].length || board[i][j] !=
word[index]) {
            return false;
        }

        // Check if we've matched all characters in the wordif (index == word.length -
1) {
            return true;
        }

        char ch = board[i][j];
        board[i][j] = '*'; // Mark the cell as visited// Explore all four possible
directionsboolean res = exists(board, i + 1, j, word, index + 1)
                || exists(board, i - 1, j, word, index + 1)
                || exists(board, i, j + 1, word, index + 1)
                || exists(board, i, j - 1, word, index + 1);

        board[i][j] = ch; // Restore the original value of the cellreturn res;
    }
}
```

Dry Runs

Dry Run 1

Input:

- `board = [['A', 'B', 'C', 'E'], ['S', 'F', 'C', 'S'], ['A', 'D', 'E', 'E']]`
- `word = "ABCCED"`

Steps:

1. Start at (0,0): 'A' matches 'A'. Proceed to (0,1).
2. At (0,1): 'B' matches 'B'. Proceed to (0,2).
3. At (0,2): 'C' matches 'C'. Proceed to (0,3).
4. At (0,3): 'E' does not match 'C'. Backtrack to (0,2).
5. Explore other directions from (0,2) and find the path: (0,2) -> (1,2) -> (1,1) -> (0,1) -> (0,0) successfully constructs "ABCCED".

Output: `true`

Dry Run 2

Input:

- `board = [['A', 'B', 'C', 'E'], ['S', 'F', 'C', 'S'], ['A', 'D', 'E', 'E']]`
- `word = "SEE"`

Steps:

1. Start at (2,2): 'E' matches 'E'. Proceed to (2,3).

2. At (2,3): 'E' matches 'E'. Need to find the next 'E'.
3. Explore (1,2), (2,2), and (2,3) directions. Find path (2,2) -> (2,3) -> (1,3).

Output: `true`

Dry Run 3

Input:

- `board = [['A', 'B', 'C', 'E'], ['S', 'F', 'C', 'S'], ['A', 'D', 'E', 'E']]`
- `word = "ABCB"`

Steps:

1. Start at (0,0): 'A' matches 'A'. Proceed to (0,1).
2. At (0,1): 'B' matches 'B'. Proceed to (0,2).
3. At (0,2): 'C' matches 'C'. Proceed to (0,3).
4. At (0,3): 'E' does not match 'B'. Backtrack and explore other paths.
5. No path constructs "ABCB".

Output: `false`