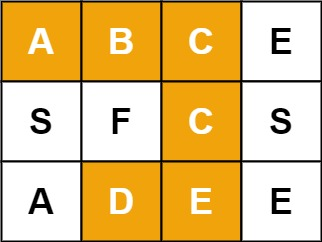
<https://leetcode.com/problems/word-search/description/>

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.

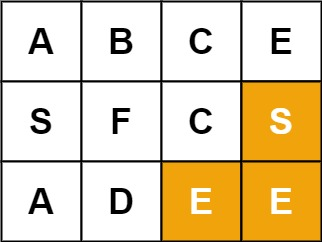
**Example 1:**



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

Output: true

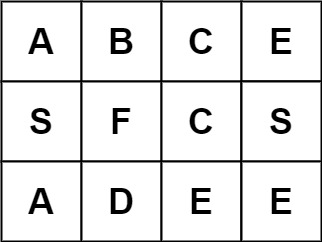
**Example 2:**



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.

 **Follow up:** Could you use search pruning to make your solution faster with a larger board?

**Attempt 1: 2023-10-25**

**Solution 1: Backtracking (10 min)**

**Style 1: With extra space**

class Solution {

    public boolean exist(char[][] board, String word) {

        int m = board.length;

        int n = board[0].length;

        boolean[][] visited = new boolean[m][n];

        for(int i = 0; i < m; i++) {

            for(int j = 0; j < n; j++) {

                if(helper(board, i, j, 0, word, visited)) {

                    return true;

                }

            }

        }

        return false;

    }

    int[] dx = new int[] {0,0,1,-1};

    int[] dy = new int[] {1,-1,0,0};

    private boolean helper(char[][] board, int x, int y, int index, String word, boolean[][] visited) {

        if(index == word.length()) {

            return true;

        }

        if(x < 0 || x >= board.length || y < 0 || y >= board[0].length

        || visited[x][y] || word.charAt(index) != board[x][y]) {

            return false;

        }

        visited[x][y] = true;

        for(int k = 0; k < 4; k++) {

            int new\_x = x + dx[k];

            int new\_y = y + dy[k];

            if(helper(board, new\_x, new\_y, index + 1, word, visited)) {

                return true;

            }

        }

        // Important: Don't forget to restore the boolean flag

        // for next round detect

        // What is the reason behind marking "visited[i][j] = false;" at the end ?

        // If I understood correctly after you have tried all the neighbors

        // of the board[i][j] and couldn't find the matches, then you have

        // to search the word from another position. let's say that position

        // is board[m][n]. so from board[m][n] perspective, it hasn't visited

        // board[i][j], so you have to set visited[i][j] back to false to allow

        // other calls use it.

        // Note: visited is a static variable which is common to all the instances

        // (or objects) of the class because it is a class level variable.

        visited[x][y] = false;

        return false;

    }

}

Time Complexity: O(n \* (4 ^ w)), where n is number of cells and w is word length.

To avoid confusion: O(r \* c \* (4 ^ w)), where r x c are the dimensions of the board.

Space Complexity: O(n)

To avoid confusion: O(r \* c), where r x c are the dimensions of the board.

------------------------------------------------------------------------------------

For each cell, we initially have 4 DIRS to go. And each way has additional 4 DIRS, which that means each word length will cost 4 times more.

Thus the total TC would be O(4 ^ L \* m \* n) where L is word.length().

**Style 2: Without extra space**

class Solution {

    public boolean exist(char[][] board, String word) {

        int m = board.length;

        int n = board[0].length;

        for(int i = 0; i < m; i++) {

            for(int j = 0; j < n; j++) {

                if(helper(board, i, j, 0, word)) {

                    return true;

                }

            }

        }

        return false;

    }

    int[] dx = new int[] {0,0,1,-1};

    int[] dy = new int[] {1,-1,0,0};

    private boolean helper(char[][] board, int x, int y, int index, String word) {

        if(index == word.length()) {

            return true;

        }

        if(x < 0 || x >= board.length || y < 0 || y >= board[0].length

        || board[x][y] == '#' || word.charAt(index) != board[x][y]) {

            return false;

        }

        char tmp = board[x][y];

        board[x][y] = '#';

        for(int k = 0; k < 4; k++) {

            int new\_x = x + dx[k];

            int new\_y = y + dy[k];

            if(helper(board, new\_x, new\_y, index + 1, word)) {

                return true;

            }

        }

        board[x][y] = tmp;

        return false;

    }

}

Time Complexity: O(n \* (4 ^ w)), where n is number of cells and w is word length.

To avoid confusion: O(r \* c \* (4 ^ w)), where r x c are the dimensions of the board.

Space Complexity: O(n),

To avoid confusion: O(r \* c), where r x c are the dimensions of the board.

Because if consider recursion it will have additional space, since it's basically a DFS using recursion. The recursion will open space on call stack, which consumes additional space. The space is O(size of board), since at a moment, the worst DFS branch traversed entire board.

------------------------------------------------------------------------------------

For each cell, we initially have 4 DIRS to go. And each way has additional 4 DIRS, which that means each word length will cost 4 times more.

Thus the total TC would be O(4 ^ L \* m \* n) where L is word.length().

**Refer to**

<https://leetcode.com/problems/word-search/solutions/27658/accepted-very-short-java-solution-no-additional-space/>

Here accepted solution based on recursion. To save memory I decuded to apply bit mask for every visited cell. Please check board[y][x] ^= 256;

public boolean exist(char[][] board, String word) {

    char[] w = word.toCharArray();

    for (int y=0; y<board.length; y++) {

    for (int x=0; x<board[y].length; x++) {

    if (exist(board, y, x, w, 0)) return true;

    }

    }

    return false;

}

private boolean exist(char[][] board, int y, int x, char[] word, int i) {

if (i == word.length) return true;

if (y<0 || x<0 || y == board.length || x == board[y].length) return false;

if (board[y][x] != word[i]) return false;

board[y][x] ^= 256;

boolean exist = exist(board, y, x+1, word, i+1)

|| exist(board, y, x-1, word, i+1)

|| exist(board, y+1, x, word, i+1)

|| exist(board, y-1, x, word, i+1);

board[y][x] ^= 256;

return exist;

}

**What is the reason behind marking "visited[i][j] = false;" at the end ?**

**Refer to**

<https://leetcode.com/problems/word-search/solutions/27811/my-java-solution/comments/265883>

If I understood correctly after you have tried all the neighbors of the board[i][j] and couldn't find the matches, then you have to search the word from another position. let's say that position is board[m][n]. so from board[m][n] perspective, it hasn't visited board[i][j], so you have to set visited[i][j] back to false to allow other calls use it.

Note: visited is a static variable which is common to all the instances (or objects) of the class because it is a class level variable.

**关于为何和如何使用Backtrack的解释**

**Refer to**

<https://algo.monster/liteproblems/79>

**Problem Description**

The problem gives us a 2D grid of characters called board and a string word. Our task is to determine if word exists in the grid. A word is said to exist in the grid if it can be formed by tracing a path through adjacent cells. Cells are considered adjacent if they are directly above, below, to the left, or to the right of one another (diagonal adjacency is not allowed). As we trace the path to form the word, we cannot use the same cell more than once. The goal is to return true if the word can be constructed from the grid following these rules, otherwise, we return false.

**Intuition**

To solve this problem, we employ a technique known as Depth-First Search (DFS). The intuition behind using DFS is that it allows us to explore all potential paths from a starting cell to see if we can spell out the word.

Here's the thinking process for arriving at the DFS solution approach:

**Start at Every Cell:** We need to consider every cell in the grid as a potential starting point of our word.

**Explore Adjacent Cells:** Once we've picked a starting cell, we look at its adjacent cells. If the current cell contains the correct letter (the next letter in word that we're looking for), we can move onto this adjacent cell.

**Track Where We've Been: In order not to use the same cell twice, we temporarily modify the current cell in the grid to mark it as 'visited'. This way, we avoid re-visiting cells on the current path. It's important to remember to reset this cell back to its original state after we finish searching from that cell.**

**Use Recursion for Simplicity:** **Implementing our DFS as a recursive function is convenient because it allows us to easily backtrack (undo our choice of the current letter and try a different path) if we realize we can't spell out the entire word from the current path.**

**Termination Conditions:** The DFS should terminate when we have successfully found the last letter of word, or when we have run out of valid paths to pursue. In the first case, we immediately return true, and in the second case, once all possibilities have been exhausted from a certain cell, we return false.

Writing this logic into a recursive function dfs, we're able to perform a thorough search from each starting cell until we find a path that matches word. By using the function any(), we execute our DFS starting from each cell and return true as soon as at least one call to dfs returns true.

The function pairwise() used in the code appears to be a utility that generates pair-wise combinations of provided elements, but it's not part of standard Python libraries as of the last update and might be custom-defined. It is effectively being used to iterate through the possible adjacent cells coordinates with respect to the current cell.

**Solution Approach**

The solution utilizes a recursive Depth-First Search (DFS) algorithm to navigate through the board. Let's walk through the key components of the implementation.

**Base Case**

Each recursive dfs call has a base case that compares the length of the word we're looking for with the index k:

if k == len(word) - 1:

return board[i][j] == word[k]

This condition checks if we've reached the end of the word. If so, we compare the current cell's character with the last character of the word. If they match, it means we've successfully found word in the board.

**Current Letter Match Check**

Before diving deeper, we ensure that the current cell's character matches the current letter in word we're looking for:

if board[i][j] != word[k]:

return False

If it doesn't match, there's no point in continuing from this cell, so we return False.

**Avoiding Re-use of the Same Cell**

To avoid revisiting the same cell, the solution marks the cell with a placeholder "0":

c = board[i][j]

board[i][j] = "0"

After the placeholder assignment, DFS exploration continues, and once done, the cell is reset to its original state:

board[i][j] = c

**Exploring Adjacent Cells**

To navigate through adjacent cells, we use a loop that iterates through coordinate offsets. Instead of a standard for loop, it uses pairwise() combined with a tuple (-1, 0, 1, 0, -1) to generate the pairs of offsets representing up, right, down, and left moves:

for a, b in pairwise((-1, 0, 1, 0, -1)):

x, y = i + a, j + b

The move is valid if it's within the bounds of the grid and the cell has not been visited (marked as '0'):

ok = 0 <= x < m and 0 <= y < n and board[x][y] != "0"

If the move is valid, the dfs is called recursively for the next cell with the index k + 1 to check the following character in the word:

if ok and dfs(x, y, k + 1):

return True

**Kickstarting DFS and Returning Result**

The dfs function is invoked for each cell in the grid by using a combination of any() and a nested for loop, which iterates through each cell's indices (i, j):

return any(dfs(i, j, 0) for i in range(m) for j in range(n))

The search terminates and returns true as soon as any starting cell leads to a successful dfs exploration that finds the word.

By employing a recursive approach, the solution ensures that all possible paths are explored from each starting cell, without revisiting cells and without departing from the constraints defined by the problem—namely the adjacency restriction and the uniqueness of the path's cells.

**Example Walkthrough**

Let's demonstrate how the Depth-First Search (DFS) solution method works using a simplified example.

Imagine we have the following grid and word:

board = [

['A', 'B', 'C', 'E'],

['S', 'F', 'C', 'S'],

['A', 'D', 'E', 'E']

]

word = "SEE"

We want to know whether we can find the word "SEE" in the grid by moving to adjacent cells without reusing any cell.

**Start at Every Cell:** We initiate our DFS from board[0][0] with A. Since A is not S (the first letter of "SEE"), the search does not continue from here. Next, we try board[0][1] which is B, again not matching 'S'. We continue this process until we find S at board[1][0].

**Explore Adjacent Cells:** From board[1][0] (which is S), we look at the adjacent cells (board[0][0], board[1][1], board[2][0] and board[1][0] as vowel cells are not considered).

**Track Where We've Been:** We mark board[1][0] as visited by setting it to "0".

**Use Recursion for Simplicity:** We start a recursive call from board[1][1] where we find F—not the correct letter, so the call returns False, and we continue the DFS.

**Termination Conditions:** Continuing the search, board[2][0] is A (also not E), so we move on to board[2][1] (which is D), and finally to board[2][2] which is E - this matches the second letter of "SEE".

The next recursive call starts from board[2][2], and we mark it visited. Its adjacent cells (excluding the one we just came from and marked ones) are checked. board[2][3] is E—the last letter we're looking for.

The base case is met (we are at the last letter of the word), and since board[2][3] matches the last letter of "SEE", the recursive call returns True.

We backtrack, unmarking the visited cells, and propagate the True value upwards through the recursion stack, ultimately return True from the initial call.

Since we've found at least one path that spells "SEE", we can stop the search and return True. No need to check further cells.

Here's a visual representation of the path that spells "SEE":

[ ]

[S]

[E]

[E]

Using this approach, we verify that "SEE" exists within the board as per the rules given, and the function would return True.

**Solution Implementation**

class Solution {

// Class level variables to hold the dimensions of the board, the word, and the board itself

private int rows;

private int cols;

private String targetWord;

private char[][] gameBoard;

// Method to determine if the target word exists in the board

public boolean exist(char[][] board, String word) {

rows = board.length; // Number of rows in the board

cols = board[0].length; // Number of columns in the board

targetWord = word; // The word we are searching for

gameBoard = board; // The game board

// Iterate over every cell in the board

for (int i = 0; i < rows; ++i) {

for (int j = 0; j < cols; ++j) {

// If the first letter matches and dfs search is successful, return true

if (dfs(i, j, 0)) {

return true;

}

}

}

// If we have not returned true at this point, the word does not exist in the board

return false;

}

// Helper method to perform Depth First Search (DFS)

private boolean dfs(int row, int col, int index) {

// Check if we are at the last character of the word

if (index == targetWord.length() - 1) {

return gameBoard[row][col] == targetWord.charAt(index);

}

// Check if current cell does not match the character at index in word

if (gameBoard[row][col] != targetWord.charAt(index)) {

return false;

}

// Temporarily mark the current cell as visited by replacing its value

char tempChar = gameBoard[row][col];

gameBoard[row][col] = '0';

// Define an array of directions (up, right, down, left)

int[] directions = {-1, 0, 1, 0, -1};

// Explore all possible adjacent cells (up, right, down, left)

for (int d = 0; d < 4; ++d) {

int newRow = row + directions[d];

int newCol = col + directions[d + 1];

// Check if the new position is within bounds and not visited

// If the dfs search from the adjacent cell is successful, return true

if (dfs(newRow, newCol, index + 1)) {

return true;

}

}

}

// Reset the cell's value back from '0' to its original character

gameBoard[row][col] = tempChar;

// If none of the adjacent cells leads to a solution, return false

return false;

}

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the given code depends on the dimensions of the board (m x n) and the length of the word to be found (l).

The exist method involves iterating over each element in the board to start a depth-first search (DFS). This introduces a factor of mn.

For every element (i, j), we are potentially doing a DFS. The maximum depth of the DFS is the length of the word l, because we are searching for the word and we return either when we find the whole word or cannot proceed further.

In each DFS call, we check all four directions (up, down, left, and right). However, since we replace the current character with "0" to mark it as visited, every recursive call will have at most 3 unvisited neighbors to explore. This results in 3^(l-1) for the DFS branching factor, as the first call has 4 directions and subsequent calls have only 3 directions because the previous position wouldn't be considered.

So, the worst-case time complexity is O(mn\*3^(l-1)), where m is the number of rows in the board, n is the number of columns in the board, and l is the length of the word.

**Space Complexity**

The space complexity also has two main contributors:

The recursive call stack due to DFS. In the worst case, the recursion goes as deep as the length of the word l, so the call stack can grow up to O(l).

We are modifying the board in place to mark visited elements, hence not using any additional space proportional to the size of the board (m\*n), except for the space used by the call stack.

Considering that, the total space complexity is O(l) due to the recursion call stack, which depends on the length of the word being searched for.