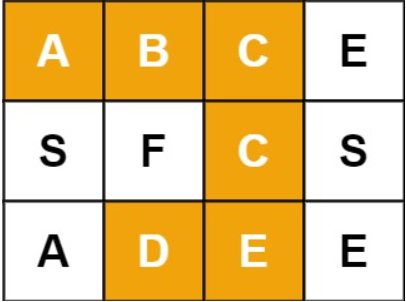
<https://leetcode.com/problems/word-search/description/>

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.

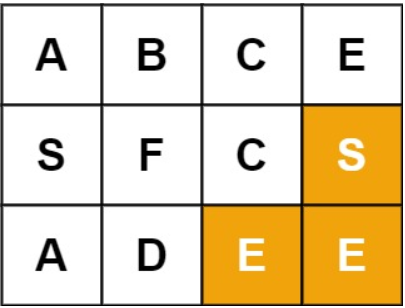
**Example 1:**



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

Output: true

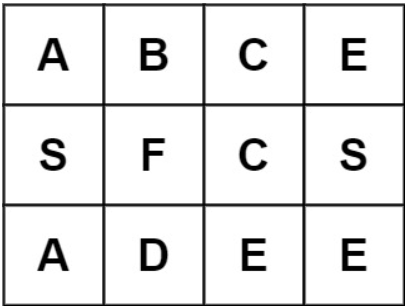
**Example 2:**



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.