JavaScript Word Search

Challenge

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.

1st Example

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

2nd Example

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

3rd Example

```
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.

Solution

```
const exist = (board, word) => {
    let result = false;

    const check = (r, c, i) => {
        if (!result) {
            if (r < 0 ||
                c < 0 ||
                r >= board.length ||
                c >= board[0].length) {
                return;
            }
```

Solution continues on next page...

```
            if (board[r][c] != word[i]) {
                return;
            }

            if (i == word.length - 1) {
                result = true;

                return;
            }

            board[r][c] = null;
            check(r + 1, c, i + 1);
            check(r - 1, c, i + 1);
            check(r, c + 1, i + 1);
            check(r, c - 1, i + 1);
            board[r][c] = word[i];
        }
    };

    for (let i = 0; i < board.length; i++) {
        for (let j = 0; j < board[0].length; j++) {
            if (board[i][j] == word[0]) {
                check(i, j, 0);

                if (result) {
                    return result;
                }
            }
        }
    }

    return result;
};
```

Explanation

I've coded a function called `exist` that checks if a given word exists in a 2D board. It uses a depth-first search algorithm to traverse the board and find the word.

The function initializes a variable called `result` as `false`, which will store the final result of whether the word exists in the board or not.

It defines a helper function called `check` that takes in three parameters: `r` (row index), `c` (column index), and `i` (current index of the word). This function performs the depth-first search to check if the word can be formed starting from the given position `(r, c)`.

Inside the `check` function, several checks are performed. If the `result` is already true, the function returns. This is because the word has already been found, and there is no need to continue searching. If the current position `(r, c)` is out of bounds of the board, the function returns. This handles cases where the current position is outside the boundaries of the board. If the character at the current position does not match the character at the current index `i` of the word, the function returns. This handles cases where the characters do not match.

If the current index `i` is equal to the length of the word minus `1`, it means that we have reached the end of the word. In this case, the `result` is set to true, indicating that the word has been found, and the function returns.

The character at the current position `(r, c)` of the board is set to null to mark it as visited. Then, the `check` function is recursively

called for the adjacent positions: `(r + 1, c)`, `(r - 1, c)`, `(r, c + 1)`, and `(r, c - 1)` with the updated index `i + 1`. This continues the depth-first search in all four directions.

After the recursive calls, the character at the current position `(r, c)` of the board is set back to the original character from the word. This is done to backtrack and restore the original state of the board.

The function uses a nested loop to iterate through all positions of the board. If the character at the current position matches the first character of the word, the `check` function is called with the initial index `0`. This starts the depth-first search from that position.

After the nested loop, the final value of `result` is returned. This indicates whether the word exists in the board or not.

Author: Trevor Morin