Solution: Word Search

Let's solve the Word Search problem using the Backtracking pattern.

We'll cover the following

Statement

Given an $m \times n$ 2D grid of characters and `word` as a string, we need to determine if the word can be constructed from letters of sequentially adjacent cells. The cells are considered sequentially adjacent when they are neighbors to each other either horizontally or vertically. The function should return TRUE if the word can be constructed and FALSE otherwise.

Constraints:

- `m` $=$ `board.length`
- `n` $=$ `board[i].length`, where $0 \leq$ `i` $<$ `m`
- $1 \leq$ `m`, `n` ≤ 6
- $1 \leq$ `word.length` ≤ 15
- `board` and `word` consist of only lowercase or uppercase English letters.
- The search is not case-sensitive.

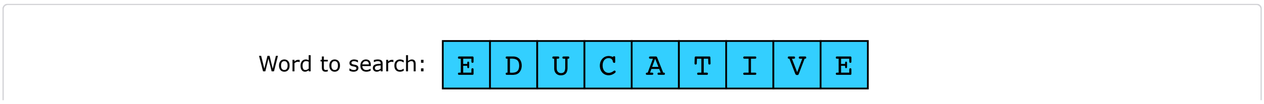
Pattern: Backtracking

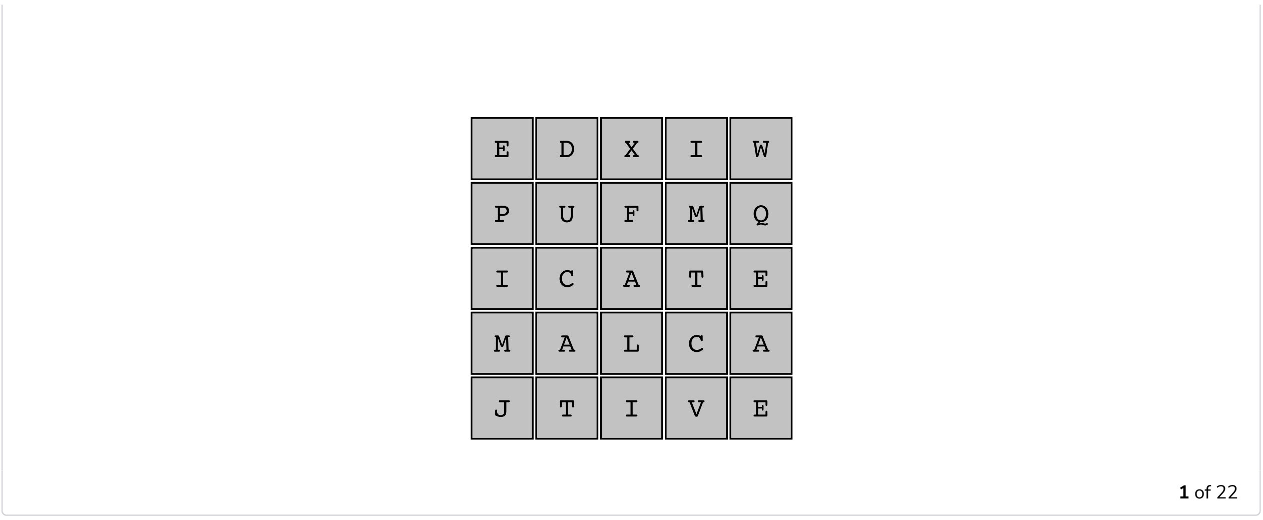
In order to search for a word in a 2-D grid, we need to traverse all the elements of the grid. The sequence of letters that combine to make a word can be found starting from any index in the grid. Therefore, we need to find that word in all possible directions, starting from a particular index. However, what would we do if we learned that the current path would not lead to the solution?

The answer to the problem above lies in **backtracking**. We can move backward from the current index and resume the search for the required word along one of the other possible paths.

Solution

This problem can be solved efficiently by implementing the backtracking technique. This will help us traverse all the possibilities to find the specific word without using the extra space needed to keep track of the visited or non-visited status of the cells.





E	D	X	I	W
P	U	F	M	Q
I	C	A	T	E
M	A	L	C	A
J	T	I	V	E

> **Note:** In the following section, we’ll gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

For each cell on the 2D grid, we have four paths we can take next. If the chosen path is incorrect, we’ll backtrack and choose a different path until the word is found or all possibilities are exhausted. We can implement the DFS algorithm using a recursive function.

> **Note:** Although the DFS algorithm is classically implemented on a tree or graph data structure, it may also be applied wherever the concept of connectedness is found in a data structure. In the 2D array provided as input, we see that each character located at cell $\{i, j\}$ is connected to its four potential neighbors, each addressable as cells $\{i+1, j\}$ (the character below), $\{i-1, j\}$ (the character above), $\{i, j+1\}$ (the character to the right), and $\{i, j-1\}$ (the character to the left). In graph terminology, we can say that each cell in the input grid has a minimum of two and a maximum of four edges.

We’ll apply a depth-first search for each cell of the grid. We’ll check for the base case in the `depthFirstSearch` function. The base case occurs when we have matched all characters in the word individually, indicating that we have found the whole word in the grid. If this is the case, we have no more letters to explore in the word, so the length of the word becomes 0. Therefore, we return TRUE.

Java

```java
         {'Z', 'A', 'P', 'E'},
         {'J', 'V', 'T', 'K'}},


        {{'O', 'Y', 'O', 'I'},
        {'B', 'I', 'E', 'M'},
        {'K', 'D', 'Y', 'R'},
        {'M', 'T', 'W', 'I'},
        {'Z', 'I', 'T', 'O'}}
        };
        String[] words = {"educative", "PACANS", "WARRIOR", "save", "DYNAMIC"};
        for(int i=0;i<words.length;i++){
          System.out.println((i+1) + ".\tGrid = ");
          printGrid(grids[i]);
          System.out.println("\tWord = "+ words[i]);
          System.out.println("\n\tProcessing...");
          Boolean result = wordSearch(grids[i], words[i]);
```

```
76          if(result == true){
77              System.out.println("\n\tSearch result = Found Word");
78          }
79          else{
80              System.out.println("\n\tSearch result = Word couldn't found");
81          }
82
83          System.out.println(new String(new char[100]).replace('\0', '-'));
84        }
85    }
86  }
```

Word Search

After the base case, we need to check if the current cell is a valid candidate for the next character of the word. In **line 23**, we do this by checking if the row and column indices are within the bounds of the grid and we also check if the character in the cell matches the next character in the word. If any of these conditions are not satisfied, we cannot proceed with the current cell, and we return FALSE to backtrack and explore other paths.

Java

```java
1   class WordSearch {
2       // Function to search a specific word in the grid
3       public static boolean wordSearch(char[][] grid, String word){
4           int n = grid.length;
5           int m = grid[0].length;
6           for(int row = 0; row < grid.length; row++){
7               for(int col = 0; col < grid[0].length; col++){
8                   if(depthFirstSearch(row, col, word, grid)){
9                       return true;
10                  }
11              }
12          }
13          return false;
14      }
15      //Apply backtracking on every element to search the required word
16      public static boolean depthFirstSearch(int row, int col, String word, char[][] grid){
17          // base case
18          if (word.length() == 0){
19              return true;
20          }
21          // Check if the cell is not out of bounds or particular grid
22          // element is not among required characters
23          if (row < 0 || row == grid.length || col < 0 || col == grid[0].length || (grid[row][col] != word.
24              return false;
25          }
26          return false;
27      }
```

Word Search

After ensuring that the current cell is not out of bounds and contains the next character of the word, we mark the cell as visited by setting it to $*$. Then, we iterate over the four possible choices (up, down, right, left) and call the `depthFirstSearch` function recursively with the updated row and column indices and the remaining portion of the word.

For each possible choice, we check if the recursive call to `depthFirstSearch` returns TRUE. If the call returns TRUE, indicating that we have found the whole word, we return TRUE. If not, we will continue the loop and try the next possible choice. If none of the choices lead to finding the whole word, we will return FALSE.

Java

```java
class WordSearch {
    // Function to search a specific word in the grid
    public static boolean wordSearch(char[][] grid, String word){
        int n = grid.length;
        int m = grid[0].length;
        for(int row = 0; row < grid.length; row++){
            for(int col = 0; col < grid[0].length; col++){
                if(depthFirstSearch(row, col, word, grid)){
                    return true;
                }
            }
        }
        return false;
    }
    // Apply backtracking on every element to search the required word
    public static boolean depthFirstSearch(int row, int col, String word, char[][] grid) {
        // base case
        if (word.length() == 0) {
            return true;
        }
        // Check if the cell is not out of bounds or particular grid
        // element is not among required characters
        if (row < 0 || row == grid.length || col < 0 || col == grid[0].length || (grid[row][col] != word.
            System.out.println("\tDid not find " + word.charAt(0) + " at cell [" + row + ", " + col + "]"
            return false;
        }

        boolean result = false;
```

Word Search

When the function has finished exploring all possible paths from the current cell and returns FALSE, we'll revert the cell's original value to unmark it and allow it to be visited in other paths. This is important because, if we don't reset the cell's value, it would still be marked as visited, and the search would be incomplete or miss out on other possible paths.

Java

```java
class WordSearch {
    // Function to search a specific word in the grid
    public static boolean wordSearch(char[][] grid, String word){
        int n = grid.length;
        int m = grid[0].length;
        for(int row = 0; row < grid.length; row++){
            for(int col = 0; col < grid[0].length; col++){
                if(depthFirstSearch(row, col, word, grid)){
                    return true;
                }
            }
        }
        return false;
    }
    //Apply backtracking on every element to search the required word
    public static boolean depthFirstSearch(int row, int col, String word, char[][] grid) {
        // base case
        if (word.length() == 0) {
            return true;
        }
        // Check if the cell is not out of bounds or particular grid
        // element is not among required characters
        if (row < 0 || row == grid.length || col < 0 || col == grid[0].length || (grid[row][col] != word.
            System.out.println("\tDid not find " + word.charAt(0) + " at cell [" + row + ", " + col + "]"
            return false;
        }

        boolean result = false;
```

Word Search

Just the code

Here’s the complete solution to this problem:

Java

```java
class WordSearch {
    // Function to search a specific word in the grid
    public static boolean wordSearch(char[][] grid, String word){
        int n = grid.length;
        int m = grid[0].length;
        for(int row = 0; row < grid.length; row++){
            for(int col = 0; col < grid[0].length; col++){
                if(depthFirstSearch(row, col, word, grid)){
                    return true;
                }
            }
        }
        return false;
    }
    //Apply backtracking on every element to search the required word
    public static boolean depthFirstSearch(int row, int col, String word, char[][] grid) {

        if (word.length() == 0) {
            return true;
        }

        if (row < 0 || row == grid.length || col < 0 || col == grid[0].length || (grid[row][col] != word.
            return false;
        }

        boolean result = false;

        grid[row][col] = '*';
```

Word Search

To recap, the solution to this problem can be divided into the following parts:

- Check the base case to see if the whole word has been found in the grid.
- Check if the current cell is out of bounds or doesn’t contain the required character.
- Explore the four neighboring cells to see if the next character of the word can be found.
- Repeat the process from the neighboring cell until the word is found or the entire grid is traversed.

Time complexity

In the `depthFirstSearch` function, we initially have *four* directions to explore. However, there are only *three* choices left in each cell afterward because one has already been explored. Therefore, the time complexity of the algorithm above is $O(n \times 3^l)$, where n is the number of cells and l is the length of the word we are searching for.

Space complexity

The space complexity is $O(l)$, where l is the length of the word to be searched in the grid. This is the maximum depth of the recursive call tree at any point during the search.

← Back

Next →

✓ Mark as Completed