<https://leetcode.com/problems/coloring-a-border/description/>

You are given an m x n integer matrix grid, and three integers row, col, and color. Each value in the grid represents the color of the grid square at that location.

Two squares are called **adjacent** if they are next to each other in any of the 4 directions.

Two squares belong to the same **connected component** if they have the same color and they are adjacent.

The **border of a connected component** is all the squares in the connected component that are either adjacent to (at least) a square not in the component, or on the boundary of the grid (the first or last row or column).

You should color the **border** of the **connected component** that contains the square grid[row][col] with color.

Return the final grid.

**Example 1:**

**Input:** grid = [[1,1],[1,2]], row = 0, col = 0, color = 3

**Output:** [[3,3],[3,2]]

**Example 2:**

**Input:** grid = [[1,2,2],[2,3,2]], row = 0, col = 1, color = 3

**Output:** [[1,3,3],[2,3,3]]

**Example 3:**

**Input:** grid = [[1,1,1],[1,1,1],[1,1,1]], row = 1, col = 1, color = 2

**Output:** [[2,2,2],[2,1,2],[2,2,2]]

**Constraints:**

m == grid.length

n == grid[i].length

1 <= m, n <= 50

1 <= grid[i][j], color <= 1000

0 <= row < m

0 <= col < n

**Attempt 1: 2024-12-21**

**Solution 1: DFS + Backtracking (360 min, the current cell's stateus correlate to its neighbours, have to wait all 4 directions further DFS finished and then update the final status of current cell)**

**Wrong Solution (147/155)**

class Solution {

    public int[][] colorBorder(int[][] grid, int r0, int c0, int color) {

        if(grid == null || grid.length == 0)

            return null;

        boolean[][] visited = new boolean[grid.length][grid[0].length];

        dfs(grid, visited, r0, c0, grid[r0][c0], color);

        return grid;

    }

    public void dfs(int[][] grid, boolean[][] visited, int i, int j, int oldColor, int newColor) {

        if (i > grid.length - 1 || i < 0 || j > grid[0].length - 1 || j < 0 || grid[i][j] != oldColor || visited[i][j])

            return;

        visited[i][j] = true;

        boolean border = false;

        if(i == 0 || j == 0 || j == grid[0].length - 1 || i == grid.length - 1 || grid[i+1][j] != oldColor || grid[i-1][j] != oldColor || grid[i][j-1] != oldColor || grid[i][j+1] != oldColor)

            border = true;

        if(border)

            grid[i][j] = newColor;

        dfs(grid, visited, i+1, j, oldColor, newColor);

        dfs(grid, visited, i-1, j, oldColor, newColor);

        dfs(grid, visited, i, j+1, oldColor, newColor);

        dfs(grid, visited, i, j-1, oldColor, newColor);

    }

}

**Test case**

grid =

[1,2,1,2,1,2]

[2,2,2,2,1,2]

[1,2,2,2,1,2]

row = 1, col = 3, color = 1

Output:

[1,1,1,1,1,2]

[1,1,1,1,1,2]

[1,1,1,1,1,2]

Expect:

The difference happen on position {1, 1}, wrong output update color from 2 to 1,

correct output keep color as 2 since {1, 1} is not a border cell in original grid

[1,1,1,1,1,2]

[1,2,1,1,1,2]

[1,1,1,1,1,2]

**Refer to chatGPT**

**Question: the current cell's 'isBorder' flag only can be set after all its four neighbors cell's further DFS traversal finished ?**

I have seen below solution and explanation provided:

NEW SOLUTION

The primary intuition is to do a DFS from the starting cell and find all the cells of the oldColor that needs to be changed. We mark these cells as visited. Once this is done, we need to find out which among those cells lies interior and which lies exterior. Interior cells have all 4 neighboring cells(top, bottom, left and right) to have the oldColor value.

1.Check for existence of null or empty grid and return null if so.

2.Initiate a DFS from starting cell.

3.Check if the current cell lies out of bounds off the grid or if current cell does not have the same color as starting cell and return if so.

4.Otherwise, mark the current cell as visited for us to remember that we have processed this cell..

**5.Declare a boolean flag to check if the current cell can be colored (i.e. to determine whether it belongs to the border in which case it should be colored or if it is just an interior node that shouldn't be colored)**

**6.Do a DFS for all neighboring points that are up, down, left and right from current cell.**

**7.Once DFS returns back for the current cell after processing all directions from it, change the current cell's color if you find that the current cell belongs to the border.**

class Solution {

public int[][] colorBorder(int[][] grid, int r0, int c0, int color) {

if(grid == null || grid.length == 0)

return null;

boolean[][] visited = new boolean[grid.length][grid[0].length];

dfs(grid, visited, r0, c0, grid[r0][c0], color);

return grid;

}

public void dfs(int[][] grid, boolean[][] visited, int i, int j, int oldColor, int newColor) {

if (i > grid.length - 1 || i < 0 || j > grid[0].length - 1 || j < 0 || grid[i][j] != oldColor || visited[i][j])

return;

visited[i][j] = true;

boolean border = false;

if(i == 0 || j == 0 || j == grid[0].length - 1 || i == grid.length - 1 || grid[i+1][j] != oldColor || grid[i-1][j] != oldColor || grid[i][j-1] != oldColor || grid[i][j+1] != oldColor)

border = true;

dfs(grid, visited, i+1, j, oldColor, newColor);

dfs(grid, visited, i-1, j, oldColor, newColor);

dfs(grid, visited, i, j+1, oldColor, newColor);

dfs(grid, visited, i, j-1, oldColor, newColor);

if(border)

grid[i][j] = newColor;

}

}

**The question is the order of steps mentioned in explanation as 5, 6, 7, looks like 5 -> 6 -> 7 is the only correct order ? If we move 7 before 6 such as below, the 'isBorder' judgement logic will be wrong ?**

boolean border = false;

if(i == 0 || j == 0 || j == grid[0].length - 1 || i == grid.length - 1 || grid[i+1][j] != oldColor || grid[i-1][j] != oldColor || grid[i][j-1] != oldColor || grid[i][j+1] != oldColor)

border = true;

if(border)

grid[i][j] = newColor;

dfs(grid, visited, i+1, j, oldColor, newColor);

dfs(grid, visited, i-1, j, oldColor, newColor);

dfs(grid, visited, i, j+1, oldColor, newColor);

dfs(grid, visited, i, j-1, oldColor, newColor);

**In another word, is that means the current cell's 'isBorder' flag only can be set after all its four neighbors cell's further DFS traversal finished ?**

**Answer:**

Yes, the order of steps 5, 6, and 7 matters in this case, and your observation is correct. Let’s break it down:

**Key Insight:**

The determination of whether a cell is a **border cell** depends on its neighbors. If any neighbor is not of the same oldColor, or if the cell lies on the edge of the grid, it is marked as a border. However, if you check and process the border condition (if (border) grid[i][j] = newColor) **before** performing DFS on the neighbors, your judgment may be incorrect because you haven't explored the neighbors yet.

**Why Step 7 Must Come After Step 6:**

Premature Border Judgments: If you mark the current cell as a border and recolor it **before** visiting its neighbors, you may miss important neighbor relationships. For example:

A neighboring cell that should be part of the connected region might not have been explored yet.

The DFS traversal might incorrectly treat some interior cells as border cells because their neighbors haven’t been visited.

Correct Border Judgment: By processing all neighbors first (dfs calls in step 6) and only then checking the border condition (step 7), you ensure:

All neighboring cells of the current cell have been visited.

You have the complete context to determine whether the current cell is truly a border cell.

How DFS Works:

DFS is a depth-first traversal. It goes as deep as possible before backtracking.

During backtracking, the algorithm collects information about the entire connected component, ensuring accurate decisions for the current cell.

**Why Moving Step 7 Before Step 6 Causes Problems:**

If you move the border-checking logic (if (border) grid[i][j] = newColor) **before** exploring neighbors, the border flag may be set based on incomplete or outdated information. Specifically:

At the time you check if a neighbor has the oldColor, the neighbor might not have been visited yet.

The algorithm may incorrectly conclude that a cell is on the border when, in reality, it's surrounded by cells of the same color.

**Example of Incorrect Behavior:**

Grid:

1 1 1

1 1 1

1 1 1

Starting Cell: (1, 1), oldColor = 1, newColor = 2.

If you prematurely recolor the cell (1, 1) before exploring its neighbors, the algorithm might treat it as a border cell because its neighbors have not yet been fully processed. This could lead to an incorrect result where interior cells are treated as borders.

**Correct Approach:**

Explore Neighbors First (Step 6):

Perform DFS for all four directions to mark all connected cells and gather information about the neighbors.

Determine Border Status After (Step 7):

Use the complete neighbor information to check if the current cell is a border and recolor it if necessary.

**Conclusion:**

The correct order is **5 -> 6 -> 7** because the border status of the current cell can only be accurately determined after fully exploring its neighbors. Moving step 7 before step 6 leads to incomplete and potentially incorrect judgments about the cell's status as a border.

**Style 1: Using 'visited' array**

class Solution {

    public int[][] colorBorder(int[][] grid, int row, int col, int color) {

        int originalColor = grid[row][col];

        int m = grid.length;

        int n = grid[0].length;

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

        // We have to differentiate trinary states:

        // 1. Not connected component (not same original color)

        // 2. Non-border connected component (same original color but non-border)

        // 3. Border connected component (same original color and border)

        // DFS to identify all same original color cells(2 + 3) and update

        // all 'Border connected component' ones(3) as new color

        helper(grid, row, col, m, n, originalColor, color, visited);

        return grid;

    }

    int[] dx = new int[] {0, 0, 1, -1};

    int[] dy = new int[] {1, -1, 0, 0};

    private void helper(int[][] grid, int x, int y, int m, int n, int originalColor, int color, boolean[][] visited) {

        // Out of boundary, not same as original color, already visited should stop

        if(x < 0 || x >= m || y < 0 || y >= n || grid[x][y] != originalColor || visited[x][y]) {

            return;

        }

        // Mark current cell as visited

        visited[x][y] = true;

        // Declare a boolean flag to check if the current cell can be colored

        // (i.e. to determine whether it belongs to the border in which

        // case it should be colored or if it is just an interior node that

        // shouldn't be colored)

        boolean isBorder = false;

        // An interior cell have all 4 neighboring cells(top, bottom, left and

        // right) to have the original color value, otherwise its border cell

        if(x == 0 || x == m - 1 || y == 0 || y == n - 1

            || grid[x + 1][y] != originalColor

            || grid[x - 1][y] != originalColor

            || grid[x][y + 1] != originalColor

            || grid[x][y - 1] != originalColor) {

            isBorder = true;

        }

        // Why cannot assign new color on current cell before further DFS ?

        // The current cell's 'isBorder' flag correlated to its neighboring cells

        // condition, we have to perform further DFS for all four directions to

        // mark all connected cells and gather information about the neighbors,

        // use the complete neighbor information to check if the current cell is

        // a border and recolor it if necessary

        // DFS essence:

        // DFS is a depth-first traversal. It goes as deep as possible before

        // backtracking. During backtracking, the algorithm collects information

        // about the entire connected component, ensuring accurate decisions for

        // the current cell

        for(int k = 0; k < 4; k++) {

            // Do a DFS for all neighboring points that are up, down, left and right

            // from current cell.

            helper(grid, x + dx[k], y + dy[k], m, n, originalColor, color, visited);

        }

        // Once DFS returns back for the current cell after processing all directions

        // from it, change the current cell's color if you find that the current cell

        // belongs to the border

        if(isBorder) {

            grid[x][y] = color;

        }

    }

}

Time Complexity: O(m \* n)

Space Complexity: O(m \* n)

**Refer to**

<https://leetcode.com/problems/coloring-a-border/solutions/284935/java-dfs-easy-to-understand/>

**OLD SOLUTION**

The primary intuition is to do a DFS from the starting cell and find all the cells of the oldColor that needs to be changed. We mark these cells with a negative value of the oldColor. Once this is done, we need to find out which among those cells lies interior and which lies exterior. Interior cells have all 4 neighboring cells(top, bottom, left and right) to have either the oldColor value or -oldColor value. Make these interior cells positive again. Once we have processed this for all necessary nodes from the starting cell, we will get a grid containing negative cells that denote the boundary. We need to sweep through the entire grid and change these negative values to the new color.

Check for existence of null or empty grid and return null if so.

Store the color of starting cell grid[r0][c0] in oldColor.

Initiate a DFS from starting cell.

Check if the current cell lies out of bounds off the grid or if current cell does not have the same color as starting cell and return if so.

Otherwise, change the current cell's color to a negative value for us to remember that we have processed this cell.

Do a DFS for all neighboring points that are up, down, left and right from current cell.

Once DFS returns back for the current cell after processing all directions from it, change the current cell's color back to positive value if you find that the current cell lies within adjacent cells top, bottom, left and right with the same value.

Once the entire DFS has been processed, we now have a grid containing negative values representing the border which needs to be recolored to the new color.

class Solution {

public int[][] colorBorder(int[][] grid, int r0, int c0, int color) {

if(grid == null || grid.length == 0)

return null;

int oldColor = grid[r0][c0];

dfs(grid, r0, c0, oldColor);

for(int i = 0; i < grid.length; i++) {

for(int j = 0; j < grid[0].length; j++) {

if(grid[i][j] < 0)

grid[i][j] = color;

}

}

return grid;

}

public void dfs(int[][] grid, int i, int j, int oldColor) {

if(i < 0 || j < 0 || i >= grid.length || j >= grid[0].length || grid[i][j] != oldColor)

return;

grid[i][j] = -oldColor;

dfs(grid, i+1, j, oldColor);

dfs(grid, i-1, j, oldColor);

dfs(grid, i, j+1, oldColor);

dfs(grid, i, j-1, oldColor);

&& oldColor == Math.abs(grid[i+1][j])

&& oldColor == Math.abs(grid[i-1][j])

&& oldColor == Math.abs(grid[i][j+1])

&& oldColor == Math.abs(grid[i][j-1]))

grid[i][j] = oldColor;

}

}

**UPDATED** - There was a test case added recently that makes the above solution timeout (especially if Leetcode has restricted the upper bound runtime). While the above solution is elegant, if time is crucial, then we can expend more memory to declare a visited array.

In that case, please use the solution below, which more or less follows the same principle of marking nodes visited by using an external boolean array rather than marking the cell to have a negative value like in solution A. While this approach wastes a lot of memory, it would help save some time. You can ask your interviewer what they would prefer(i.e. save memory or make it faster), in order to use solution A or B.

**NEW SOLUTION**

The primary intuition is to do a DFS from the starting cell and find all the cells of the oldColor that needs to be changed. We mark these cells as visited. Once this is done, we need to find out which among those cells lies interior and which lies exterior. Interior cells have all 4 neighboring cells(top, bottom, left and right) to have the oldColor value.

Check for existence of null or empty grid and return null if so.

Initiate a DFS from starting cell.

Check if the current cell lies out of bounds off the grid or if current cell does not have the same color as starting cell and return if so.

Otherwise, mark the current cell as visited for us to remember that we have processed this cell..

Declare a boolean flag to check if the current cell can be colored (i.e. to determine whether it belongs to the border in which case it should be colored or if it is just an interior node that shouldn't be colored)

Do a DFS for all neighboring points that are up, down, left and right from current cell.

Once DFS returns back for the current cell after processing all directions from it, change the current cell's color if you find that the current cell belongs to the border.

class Solution {

public int[][] colorBorder(int[][] grid, int r0, int c0, int color) {

if(grid == null || grid.length == 0)

return null;

boolean[][] visited = new boolean[grid.length][grid[0].length];

dfs(grid, visited, r0, c0, grid[r0][c0], color);

return grid;

}

public void dfs(int[][] grid, boolean[][] visited, int i, int j, int oldColor, int newColor) {

if (i > grid.length - 1 || i < 0 || j > grid[0].length - 1 || j < 0 || grid[i][j] != oldColor || visited[i][j])

return;

visited[i][j] = true;

boolean border = false;

if(i == 0 || j == 0 || j == grid[0].length - 1 || i == grid.length - 1 || grid[i+1][j] != oldColor || grid[i-1][j] != oldColor || grid[i][j-1] != oldColor || grid[i][j+1] != oldColor)

border = true;

dfs(grid, visited, i+1, j, oldColor, newColor);

dfs(grid, visited, i-1, j, oldColor, newColor);

dfs(grid, visited, i, j+1, oldColor, newColor);

dfs(grid, visited, i, j-1, oldColor, newColor);

if(border)

grid[i][j] = newColor;

}

}

**Style 2: Using 'visited' array**

class Solution {

    public int[][] colorBorder(int[][] grid, int row, int col, int color) {

        int originalColor = grid[row][col];

        int m = grid.length;

        int n = grid[0].length;

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

        // We have to differentiate trinary states:

        // 1. Not connected component (not same original color)

        // 2. Non-border connected component (same original color but non-border)

        // 3. Border connected component (same original color and border)

        // DFS to identify all same original color cells(2 + 3) and update

        // all 'Border connected component' ones(3) as new color

        helper(grid, row, col, m, n, originalColor, color, visited);

        return grid;

    }

    int[] dx = new int[] {0, 0, 1, -1};

    int[] dy = new int[] {1, -1, 0, 0};

    private void helper(int[][] grid, int x, int y, int m, int n, int originalColor, int color, boolean[][] visited) {

        // Mark current cell as visited

        visited[x][y] = true;

        // Declare a boolean flag to check if the current cell can be colored

        // (i.e. to determine whether it belongs to the border in which

        // case it should be colored or if it is just an interior node that

        // shouldn't be colored)

        boolean isBorder = false;

        // Check if the current cell is a border cell and should be colored

        if(x == 0 || x == m - 1 || y == 0 || y == n - 1) {

            isBorder = true;

        }

        for(int k = 0; k < 4; k++) {

            int new\_x = x + dx[k];

            int new\_y = y + dy[k];

                if(!visited[new\_x][new\_y]) {

                    // If the neighboring cell of current cell has not been visited

                    // and has the same original color, continue DFS

                    if(grid[new\_x][new\_y] == originalColor) {

                        helper(grid, new\_x, new\_y, m, n, originalColor, color, visited);

                    // If the neighboring cell of current cell has a different color,

                    // current cell is a border cell

                    } else {

                        isBorder = true;

                    }

                }

            } else {

                // If the current cell is on the grid edge (because the new\_x or new\_y

// out of boundary as first time, so current cell's index {x, y} must

// in boundary, and must be on border), it is a border cell

                isBorder = true;

            }

            // If the current cell is a border cell, color it with the new color

            if(isBorder) {

                grid[x][y] = color;

            }

        }

    }

}

Time Complexity: O(m \* n)

Space Complexity: O(m \* n)

**Refer to**

<https://algo.monster/liteproblems/1034>

**Problem Description**

You are given a matrix of integers where each integer represents a color. You need to identify a connected component in the matrix based on a starting cell given by its row and column indexes. A connected component entails all adjacent cells (up, down, left, or right) with the same color. The border of this connected component includes all cells in the component adjacent to cells with different colors or cells lying on the grid's boundary.

Your task is to recolor only the cells on the border of this connected component with a new color. The index of the starting cell within the connected component and the new color are given. The updated grid should be returned, showing the new border color while leaving the interior of the connected component and the rest of the grid unchanged.

**Flowchart Walkthrough**

For leetcode 1034. Coloring A Border, let's analyze the problem using the [Flowchart](https://algo.monster/flowchart). Here's how we'd follow the steps to determine the best approach:

Is it a graph?

Yes: The grid can be seen as a graph where each cell is a node, connected to its adjacent neighbors.

Is it a tree?

No: The grid structure allows cells to create cycles and isn't necessarily hierarchical.

Is the problem related to directed acyclic graphs (DAGs)?

No: The focus here is coloring the border of a connected component, not processing elements in a directed acyclic order.

Is the problem related to shortest paths?

No: We are not seeking the shortest path; we need to identify and modify the boundary of a connected region.

Does the problem involve connectivity?

Yes: The task is to find the boundary of a connected component that matches a given condition.

Does the problem have small constraints?

Yes: We deal with a grid (typically small in competitive programming contexts), suggesting that DFS/backtracking might be efficient enough.

Since we reach the node "DFS/backtracking" from "Does the problem have small constraints?" and answer positively, the recommended algorithm is **Depth-First Search** (DFS) for solving this problem.

Conclusion: Using Depth-First Search (DFS) is ideal for traversing the connected component, identifying border cells, and modifying them as required.

**Intuition**

The solution to this problem uses [Depth-First Search](https://algo.monster/problems/dfs_intro) (DFS), an algorithm well-suited for exploring and marking the structure of a connected component within a grid. We begin at a cell (grid[row][col]) and look for all adjacent cells that share the same color, which signifies that they are part of the same connected component.

The intuition behind using DFS is to traverse through all the cells in the connected component starting from the specified cell (grid[row][col]). As we do this, we need to determine which cells are on the border of the component. A border cell is defined as a cell that:

**Is on the edge of the grid, or**

**Is adjacent to a cell that has a different color than the one we started with.**

**The DFS function checks all four directions around a cell. For every cell that we visit:**

**If the cell is within the grid and the adjacent cell is not visited yet, we check:**

**If the adjacent cell has the same color, we continue DFS on that cell since it's part of the same component.**

**If the adjacent cell has a different color, we know that we are on the border, and thus we change the color of the current cell.**

**If the cell is on the edge of the grid, it's a border cell, so we change the color of the current cell.**

We iterate in this manner until all the cells in the connected component have been visited and the appropriate border cells have been recolored. An auxiliary vis matrix is used to keep track of visited cells to avoid processing a cell more than once, which helps in preventing an infinite loop.

Finally, we return the modified grid with the newly colored borders, while the interior of the connected component and all other parts of the grid remain unchanged.

**Solution Approach**

The solution utilizes a **Depth-First Search (DFS)** approach to explore the grid and identify border cells of the connected component. The following are the key aspects of the implementation:

DFS Method: The solution defines a nested dfs function which is responsible for the recursive search. It accepts the current cell's indices i and j, as well as the original color c of the connected component.

Tracking Visited Cells: A two-dimensional list vis is created with the same dimensions as the input grid grid, initialized with False values. This list is used to keep track of cells that have already been visited to prevent infinite recursion and repeated processing.

Checking Neighbors: In each call to the dfs function, it iterates over the four adjacent cells (up, down, left, right) using a for-loop and the pairwise helper to generate pairs of directions.

Edge Conditions: The algorithm checks if a neighboring cell is within the boundaries of the grid. If it is outside, it means the current cell is on the border of the grid and thus should be recolored.

Coloring Border Cells:

If the adjacent cell is within the grid and hasn't been visited (vis[x][y] is False), there are two conditions:

If the adjacent cell has the same color as the original (grid[x][y] == c), the dfs function is called recursively for that cell to continue the component traversal.

If the adjacent cell has a different color, the current cell is on the border of the connected component and its color is changed to color.

Calling DFS: After initializing the vis grid, the dfs function is invoked starting from the cell grid[row][col], using its color as the reference for the connected component.

Returning the Result: Once the DFS traversal is complete and all border cells are recolored, the modified grid is returned.

The implementation creatively uses recursion to deeply explore the connected component and the auxiliary vis matrix to manage state across recursive calls. The determination of border cells is carried out in a granular fashion by checking each neighbor, which effectively allows the coloring to occur only where it is required.

**Solution Implementation**

class Solution {

private int[][] grid; // Represents the grid of colors.

private int newColor; // The color to be used for the border.

private int rows; // Number of rows in the grid.

private int cols; // Number of columns in the grid.

private boolean[][] visited; // To keep track of visited cells during DFS.

public int[][] colorBorder(int[][] grid, int row, int col, int color) {

this.grid = grid;

this.newColor = color;

rows = grid.length;

cols = grid[0].length;

visited = new boolean[rows][cols];

// Perform DFS starting from the given cell.

dfs(row, col, grid[row][col]);

return grid;

}

private void dfs(int i, int j, int originalColor) {

visited[i][j] = true; // Mark the current cell as visited.

int[] directions = {-1, 0, 1, 0, -1}; // To traverse in the 4 possible directions: up, right, down, left.

// Check if the current cell is a border cell and should be colored.

boolean isBorderCell = i == 0 || i == rows - 1 || j == 0 || j == cols - 1;

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

int x = i + directions[k], y = j + directions[k + 1];

// If the neighboring cell has not been visited and has the original color, continue DFS.

if (!visited[x][y]) {

if (grid[x][y] == originalColor) {

dfs(x, y, originalColor);

} else {

// If the neighboring cell has a different color, this is a border cell.

isBorderCell = true;

}

}

} else {

// If the cell is on the grid edge, it is a border cell.

isBorderCell = true;

}

}

// If the current cell is a border cell, color it with the new color.

if (isBorderCell) {

grid[i][j] = newColor;

}

}

}

**Time and Space Complexity**

The provided Python code performs a depth-first search (DFS) to color the borders of a connected component in a grid based on the given conditions. The complexity analysis is as follows:

**Time Complexity:**

The time complexity of this DFS algorithm is mainly determined by the number of cells in the grid that are visited. In the worst case, the algorithm will visit every cell in the grid once. There are m \* n cells, where m is the number of rows and n is the number of columns in the grid. Therefore, the DFS will have a time complexity of O(m \* n).

**Space Complexity:**

The space complexity includes the memory taken by the vis matrix to keep track of visited cells, as well as the recursion call stack for DFS. The vis matrix has the same dimensions as the input grid, so it takes O(m \* n) space.

The recursion depth can go as deep as the maximum size of the connected component in the grid. In the worst case, this could be the entire grid, which would mean that the call stack can potentially grow up to m \* n calls deep. Hence, the space complexity due to the recursion stack could also be O(m \* n).

Therefore, the overall space complexity is O(m \* n) as well.

Note: The provided code contains a pairwise function which is not defined within the code snippet or as part of the standard Python library up to the knowledge cutoff date. Assuming it's intended to generate pairs of coordinates for traversing adjacent cells, its behavior seems to be that of incorrectly iterating over the grid coordinates without actually yielding pairs.

**Style 3: Not use 'visited' array, then we have to modify original grid cell's value (TLE 154/155)**

class Solution {

    public int[][] colorBorder(int[][] grid, int row, int col, int color) {

        int originalColor = grid[row][col];

        int m = grid.length;

        int n = grid[0].length;

        // We have to differentiate trinary states:

        // 1. Not connected component (not same original color)

        // 2. Non-border connected component (same original color but non-border)

        // 3. Border connected component (same original color and border)

        // DFS to identify all same original color cells(2 + 3) and update

        // all 'Border connected component' ones(3) as new color

        helper(grid, row, col, m, n, originalColor);

        for(int i = 0; i < m; i++) {

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

                if(grid[i][j] < 0) {

                    grid[i][j] = color;

                }

            }

        }

        return grid;

    }

    int[] dx = {0, 0, 1, -1};

    int[] dy = {1, -1, 0, 0};

    private void helper(int[][] grid, int x, int y, int m, int n, int originalColor) {

        if(x < 0 || x >= m || y < 0 || y >= n || grid[x][y] != originalColor) {

            return;

        }

        grid[x][y] = -originalColor;

        // Declare a boolean flag to check if the current cell can be colored

        // (i.e. to determine whether it belongs to the border in which

        // case it should be colored or if it is just an interior node that

        // shouldn't be colored)

        boolean isBorder = false;

        // An interior cell have all 4 neighbor cells(top, bottom, left and

        // right) to have the original color value, otherwise its border cell.

        // Don't forget to add 'Math.abs()' to check value since we manually

        // flip cell's original color to negative, even the cell is not on

        // border but just same as original color, so when try to determine

        // neighbor cell's color equal to original color or not, have to

        // consider previously flipped to negative ones

        // If not add 'Math.abs()' test out by:

        // grid =      [1,2,1,2,1,2]

        //             [2,2,2,2,1,2]

        //             [1,2,2,2,1,2]

        // row = 1, col = 3, color = 1

        // Output   =  [1,1,1,1,1,2]

        //             [1,1,1,1,1,2]

        //             [1,1,1,1,1,2]

        // Expected =  [1,1,1,1,1,2]

        //             [1,2,1,1,1,2]

        //             [1,1,1,1,1,2]

        // The '2' at {1, 3} is interior cell only, its four directions neighbour

        // color also the same as '2', and we flip all of them to -2, so when

        // check if {1, 3} is an interior or border cell, we check its four

        // neighbour's original color(before flip) equal to original color

        // 2 or not, to restore to original color, use Math.abs(...), in

        // this example, it will restore -2 at {0, 1}, {1, 0}, {1, 2}, {2, 1}

        // back to 2 and inequality not 'TRUE', then 'isBorder' not set to

        // 'TRUE', hence we get correct flag for {1, 3} as 'isBorder = FALSE'

        if(x == 0 || x == m - 1 || y == 0 || y == n - 1

            || Math.abs(grid[x + 1][y]) != originalColor

            || Math.abs(grid[x - 1][y]) != originalColor

            || Math.abs(grid[x][y + 1]) != originalColor

            || Math.abs(grid[x][y - 1]) != originalColor) {

            isBorder = true;

        }

        for(int k = 0; k < 4; k++) {

            helper(grid, x + dx[k], y + dy[k], m, n, originalColor);

        }

        // We only restore value for cells not on border, the cells

        // on border we keep as negative value

        if(!isBorder) {

            grid[x][y] = originalColor;

        }

    }

}

**Refer to**

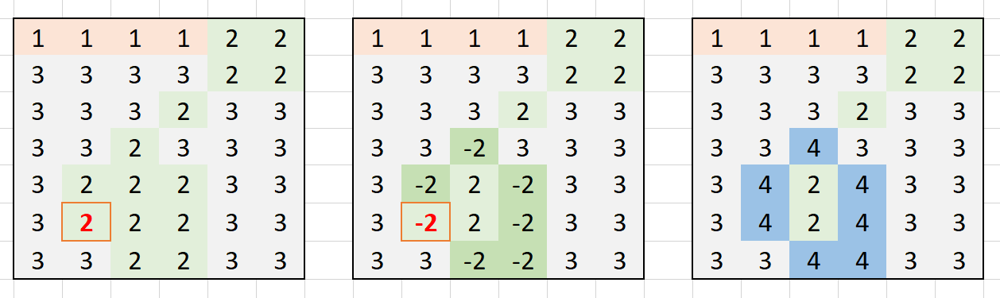
<https://leetcode.com/problems/coloring-a-border/solutions/282847/c-with-picture-dfs/>

<https://leetcode.com/problems/coloring-a-border/solutions/282847/c-with-picture-dfs/comments/1919239>

**From an initial point, perform DFS and flip the cell color to negative to track visited cells.**

**After DFS is complete for the cell, check if this cell is inside. If so, flip its color back to the positive.**

In the end, cells with the negative color are on the border. Change their color to the target color.



class Solution {

public int[][] colorBorder(int[][] grid, int r0, int c0, int color) {

dfs(grid, r0, c0, grid[r0][c0]);

for (int i = 0; i < grid.length; i++) {

for (int j = 0; j < grid[i].length; j++) {

if (grid[i][j] < 0) {

grid[i][j] = color;

}

}

}

return grid;

}

private void dfs(int[][] grid, int r, int c, int cl) {

if (r < 0 || c < 0 || r >= grid.length || c >= grid[0].length || grid[r][c] != cl) {

return;

}

grid[r][c] = -cl;

dfs(grid, r - 1, c, cl);

dfs(grid, r + 1, c, cl);

dfs(grid, r, c - 1, cl);

dfs(grid, r, c + 1, cl);

cl == Math.abs(grid[r - 1][c]) &&

cl == Math.abs(grid[r + 1][c]) &&

cl == Math.abs(grid[r][c - 1]) &&

cl == Math.abs(grid[r][c + 1])) {

grid[r][c] = cl;

}

}

}

**Why we should not change original grid value but only use 'visited' array ?**

**TLE test case**

grid =

[[1,1,1,1,1,1,1,1,1,1],

[1,1,1,1,1,1,1,1,1,1],

[1,1,1,1,1,1,1,1,1,1],

[1,1,1,1,1,1,1,1,1,1],

[1,1,1,1,1,1,1,1,1,1],

[1,1,1,1,1,1,1,1,1,1],

[1,1,1,1,1,1,1,1,1,1],

[1,1,1,1,1,1,1,1,1,1],

[1,1,1,1,1,1,1,1,1,1],

[1,1,1,1,1,1,1,1,1,1]]

row = 0, col = 0, color = 3

**This solution is not optimal because you will be visiting same node more than once. This will happen because you are not using visited grid and using -ve sign to check if it is visited or not.**

**So, what you are doing, you are reverting the sign of node if it is surrounded by all node of the same color which will mark that node again as unvisited and next time some other node will also call the same node, which is equivalent to backtracking.**

**Why your code is submitted successfully because leetcode has week test cases for this problem.**

Someone has reported to add the strong testcases in this issue: <https://github.com/LeetCode-Feedback/LeetCode-Feedback/issues/13932>

**If still want to use flip value strategy, we can use it along with visited array**

**Refer to chatGPT**

class Solution {

public int[][] colorBorder(int[][] grid, int row, int col, int color) {

int rows = grid.length;

int cols = grid[0].length;

// To mark visited cells

// visited[x][y] = 0 -> not visited

// visited[x][y] = 1 -> visited

// visited[x][y] = 2 -> border cell

int[][] visited = new int[rows][cols];

int originalColor = grid[row][col]; // The color of the connected component

dfs(grid, visited, row, col, originalColor);

// Color the border cells

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

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

if (visited[i][j] == 2) { // Border cells marked during DFS

grid[i][j] = color;

}

}

}

return grid;

}

private void dfs(int[][] grid, int[][] visited, int row, int col, int originalColor) {

// Mark the cell as visited

visited[row][col] = 1;

// Track if the cell is part of the border

boolean isBorder = false;

// Directions for moving up, down, left, and right

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

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

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

int newRow = row + dx[k];

int newCol = col + dy[k];

// Check if the new cell(neighbour cell) is out of bounds, if yes means current cell on border

if (newRow < 0 || newRow >= grid.length || newCol < 0 || newCol >= grid[0].length) {

isBorder = true;

} else if (grid[newRow][newCol] != originalColor) {

// Check if the new cell(neighbour cell) has a different color, if yes means current cell

// on border since it adjacent to a different color cell

isBorder = true;

} else if (visited[newRow][newCol] == 0) {

// Recurse into the connected component since current cell

// not on border and has same color but not visited

dfs(grid, visited, newRow, newCol, originalColor);

}

}

// If the cell is a border cell, mark it with 2

if (isBorder) {

visited[row][col] = 2;

}

}

}

**Refer to**

[L463.Island Perimeter](note://WEBd2e0876218f1490cdf71a48b5b1c5a7d)

[L695.Max Area of Island (Ref.L200,L1568)](note://WEBe10a2b6bab245be1e94ccd874af71ec6)

[L733.Flood Fill (Ref.L200,L1034)](note://1310F1DAD23741B1A0C4C1EA338D0ACE)

[L1568.Minimum Number of Days to Disconnect Island (Ref.L200)](note://WEBe1f9dea2f2571376df84b64404041136)