<https://leetcode.com/problems/flood-fill/>

You are given an image represented by an m x n grid of integers image, where image[i][j] represents the pixel value of the image. You are also given three integers sr, sc, and color. Your task is to perform a **flood fill** on the image starting from the pixel image[sr][sc].

To perform a **flood fill**:

Begin with the starting pixel and change its color to color.

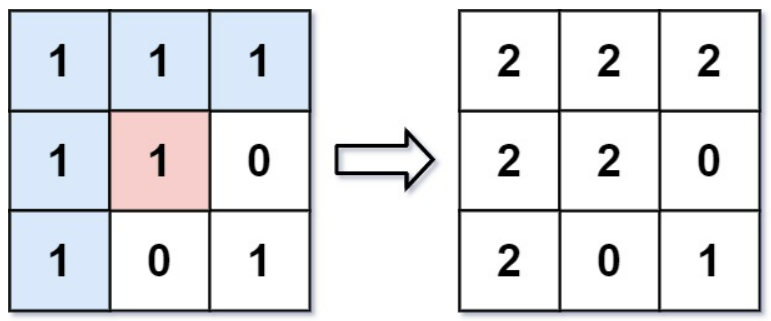
Perform the same process for each pixel that is **directly adjacent** (pixels that share a side with the original pixel, either horizontally or vertically) and shares the **same color** as the starting pixel.

Keep **repeating** this process by checking neighboring pixels of the updated pixels and modifying their color if it matches the original color of the starting pixel.

The process **stops** when there are **no more** adjacent pixels of the original color to update.

Return the **modified** image after performing the flood fill.

**Example 1:**



Input: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2

Output: [[2,2,2],[2,2,0],[2,0,1]]

Explanation: From the center of the image with position (sr, sc) = (1, 1) (i.e., the red pixel), all pixels connected by a path of the same color as the starting pixel (i.e., the blue pixels) are colored with the new color.

Note the bottom corner is not colored 2, because it is not 4-directionally connected to the starting pixel.

**Example 2:**

Input: image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, color = 0

Output: [[0,0,0],[0,0,0]]

Explanation: The starting pixel is already colored 0, so no changes are made to the image.

**Constraints:**

m == image.length

n == image[i].length

1 <= m, n <= 50

0 <= image[i][j], color < 2^16

0 <= sr < m

0 <= sc < n

**Attempt 1: 2022-12-16**

**Solution 1:  DFS (10 min)**

class Solution {

    int[] dx = new int[]{0,0,1,-1};

    int[] dy = new int[]{1,-1,0,0};

    public int[][] floodFill(int[][] image, int sr, int sc, int color) {

        helper(image, sr, sc, image[sr][sc], color);

        return image;

    }

    private void helper(int[][] image, int sr, int sc, int initial\_color, int color) {

        // image[sr][sc] != initial\_color --> stop because of different initial color

        // image[sr][sc] == color --> stop because of already visited and updated to same color

        if(sr < 0 || sr >= image.length || sc < 0 || sc >= image[0].length || image[sr][sc] != initial\_color || image[sr][sc] == color) {

            return;

        }

        image[sr][sc] = color;

        for(int i = 0; i < 4; i++) {

            int new\_sr = sr + dx[i];

            int new\_sc = sc + dy[i];

            helper(image, new\_sr, new\_sc, initial\_color, color);

        }

    }

}

Time Complexity: O(N), where N is the number of pixels in the image. We might process every pixel.

Space Complexity: O(N), the size of the implicit call stack when calling dfs.

**Refer to**

<https://leetcode.com/problems/flood-fill/solutions/127585/flood-fill/>

**Approach #1: Depth-First Search [Accepted]**

**Intuition**

We perform the algorithm explained in the problem description: paint the starting pixels, plus adjacent pixels of the same color, and so on.

**Algorithm**

Say color is the color of the starting pixel. Let's flood fill the starting pixel: we change the color of that pixel to the new color, then check the 4 neighboring pixels to make sure they are valid pixels of the same color, and of the valid ones, we flood fill those, and so on.

We can use a function dfs to perform a flood fill on a target pixel.

class Solution {

public int[][] floodFill(int[][] image, int sr, int sc, int newColor) {

int color = image[sr][sc];

if (color != newColor) {

dfs(image, sr, sc, color, newColor);

}

return image;

}

public void dfs(int[][] image, int r, int c, int color, int newColor) {

if (image[r][c] == color) {

image[r][c] = newColor;

if (r >= 1) {

dfs(image, r - 1, c, color, newColor);

}

if (c >= 1) {

dfs(image, r, c - 1, color, newColor);

}

if (r + 1 < image.length) {

dfs(image, r + 1, c, color, newColor);

}

if (c + 1 < image[0].length) {

dfs(image, r, c + 1, color, newColor);

}

}

}

}

**Complexity Analysis**

Time Complexity: *O*(*N*), where *N* is the number of pixels in the image. We might process every pixel.

Space Complexity: *O*(*N*), the size of the implicit call stack when calling dfs.

**Refer to**

[L200.Number of Islands (Ref.L1568)](note://33605BDB820F44709185B3524EB4775A)