Flood Fill Algorithm

Given a 2D screen **arr[][]** where each **arr[i][j]** is an integer representing the color of that pixel, also given the location of a pixel **(X, Y)** and a color **C**, the task is to replace the color of the given pixel and all the adjacent same-colored pixels with the given color.

Example:

Input: *arr[][] = {*
{1, 1, 1, 1, 1, 1, 1, 1},
{1, 1, 1, 1, 1, 1, 0, 0},
{1, 0, 0, 1, 1, 0, 1, 1},
*{1, **2, 2, 2, 2,** 0, 1, 0},*
*{1, 1, 1, **2, 2,** 0, 1, 0},*
*{1, 1, 1, **2, 2, 2, 2,** 0},*
*{1, 1, 1, 1, 1, **2,** 1, 1},*
*{1, 1, 1, 1, 1, **2, 2,** 1}}*
X = 4, Y = 4, C = 3

Output:
1 1 1 1 1 1 1 1
1 1 1 1 1 1 0 0
1 0 0 1 1 0 1 1
*1 **3 3 3 3** 0 1 0*
*1 1 1 **3 3** 0 1 0*
*1 1 1 **3 3 3 3** 0*
*1 1 1 1 1 **3** 1 1*
*1 1 1 1 1 **3 3** 1*

Explanation:
The values in the given 2D screen indicate colors of the pixels. X and Y are coordinates of the brush, C is the color that should replace the previous color on screen[X][Y] and all surrounding pixels with the same color. Hence all the 2 are replaced with 3.

BFS Approach: The idea is to use BFS traversal to replace the color with the new color.

- Create an empty queue let's say *Q*.
- Push the starting location of the pixel as given in the input and apply the replacement color to it.
- Iterate until **Q** is not empty and pop the front node (pixel position).

- Check the pixels adjacent to the current pixel and push them into the queue if valid (had not been colored with replacement color and have the same color as the old color).

Below is the implementation of the above approach:

C++

```cpp
// C++ implementation of the approach
#include <bits/stdc++.h>
using namespace std;

// Function that returns true if
// the given pixel is valid
bool isValid(int screen[][8], int m, int n, int x, int y,
             int prevC, int newC)
{
    if (x < 0 || x >= m || y < 0 || y >= n
        || screen[x][y] != prevC || screen[x][y] == newC)
        return false;
    return true;
}

// FloodFill function
void floodFill(int screen[][8], int m, int n, int x, int y,
               int prevC, int newC)
{
    queue<pair<int, int> > queue;

    // Append the position of starting
    // pixel of the component
```

```
pair<int, int> p(x, y);
queue.push(p);

// Color the pixel with the new color
screen[x][y] = newC;

// While the queue is not empty i.e. the
// whole component having prevC color
// is not colored with newC color
while (queue.size() > 0) {
    // Dequeue the front node
    pair<int, int> currPixel = queue.front();
    queue.pop();

    int posX = currPixel.first;
    int posY = currPixel.second;

    // Check if the adjacent
    // pixels are valid
    if (isValid(screen, m, n, posX + 1, posY, prevC,
                newC)) {
        // Color with newC
        // if valid and enqueue
        screen[posX + 1][posY] = newC;
        p.first = posX + 1;
        p.second = posY;
        queue.push(p);
        }

        if (isValid(screen, m, n, posX - 1, posY, prevC,
                    newC)) {
            screen[posX - 1][posY] = newC;
            p.first = posX - 1;
            p.second = posY;
            queue.push(p);
        }

        if (isValid(screen, m, n, posX, posY + 1, prevC,
                    newC)) {
            screen[posX][posY + 1] = newC;
            p.first = posX;
            p.second = posY + 1;
            queue.push(p);
        }

        if (isValid(screen, m, n, posX, posY - 1, prevC,
                    newC)) {
            screen[posX][posY - 1] = newC;
            p.first = posX;
            p.second = posY - 1;
            queue.push(p);
        }
    }
}
```

```
int main()
{
    int screen[][8] = { { 1, 1, 1, 1, 1, 1, 1, 1 },
                        { 1, 1, 1, 1, 1, 1, 0, 0 },
                        { 1, 0, 0, 1, 1, 0, 1, 1 },
                        { 1, 2, 2, 2, 2, 0, 1, 0 },
                        { 1, 1, 1, 2, 2, 0, 1, 0 },
                        { 1, 1, 1, 2, 2, 2, 2, 0 },
                        { 1, 1, 1, 1, 1, 2, 1, 1 },
                        { 1, 1, 1, 1, 1, 2, 2, 1 } };

    // Row of the display
    int m = 8;

    // Column of the display
    int n = 8;

    // Co-ordinate provided by the user
    int x = 4;
    int y = 4;

    // Current color at that co-ordinate
    int prevC = screen[x][y];

    // New color that has to be filled
    int newC = 3;
    floodFill(screen, m, n, x, y, prevC, newC);

    // Printing the updated screen
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cout << screen[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output

```
1 1 1 1 1 1 1 1
1 1 1 1 1 1 0 0
1 0 0 1 1 0 1 1
1 3 3 3 3 0 1 0
1 1 1 3 3 0 1 0
1 1 1 3 3 3 3 0
1 1 1 1 1 3 1 1
1 1 1 1 1 3 3 1
```

Time Complexity: O(N*M)
Auxiliary Space: O(N*M)

An Approach using DFS:

- Change the color of the source row and source column with the given color

- Do DFS in four direction

Below is the implementation of the above approach:

C++

```
// C++ implementation of the approach

#include <bits/stdc++.h>

using namespace std;


// DFS Approach

void dfs(int row,int col,vector<vector<int>> &image,vector<vector<int>>& ans,int newColor,int
iniColor,int n,int m,int delrow[],int delcol[]){


    // Marking it as the newColor

    ans[row][col] = newColor;

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

        int nrow = row + delrow[i];

        int ncol = col + delcol[i];

          // Checking Out Of Bound Condition

        if(nrow>=0 && ncol>=0 && nrow<n && ncol<m && image[nrow][ncol]==iniColor &&
ans[nrow][ncol]!=newColor){

            dfs(nrow,ncol,image,ans,newColor,iniColor,n,m,delrow,delcol);

        }

    }


}

// FloodFill Function

vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int newColor) {

        // Code here

        vector<vector<int>> ans = image;

        int n = image.size();
        int m = image[0].size();
          // Initial Color
        int iniColor = image[sr][sc];
          // vectors for changing of rows and column direction
          // UP LEFT DOWN RIGHT
        int delrow[] = {-1,0,+1,0};
        int delcol[] = {0,+1,0,-1};
          // Calling dfs function
        dfs(sr,sc,image,ans,newColor,iniColor,n,m,delrow,delcol);
        return ans;
    }

// Driver code
int main()
{
    vector<vector<int> > screen
        = { {1, 1, 1},
            {1, 1, 0},
            {1, 0, 1} };

    int n = screen.size();
      int m = screen[0].size();
    // Co-ordinate provided by the user
    int x = 1;
    int y = 1;

    // New color that has to be filled
    int newC = 3;
    vector<vector<int>> ans = floodFill(screen, x, y, newC);



    // Printing the updated screen
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cout << ans[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output

```
1 1 1 1 1 1 1 1
1 1 1 1 1 1 0 0
1 0 0 1 1 0 1 1
1 3 3 3 3 0 1 0
1 1 1 3 3 0 1 0
1 1 1 3 3 3 3 0
1 1 1 1 1 3 1 1
1 1 1 1 1 3 3 1
```

Time Complexity: O(m*n)
Auxiliary Space: O(m + n), due to the recursive call stack.