Problem 733: Flood Fill

Problem Information

Difficulty: Easy
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

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:

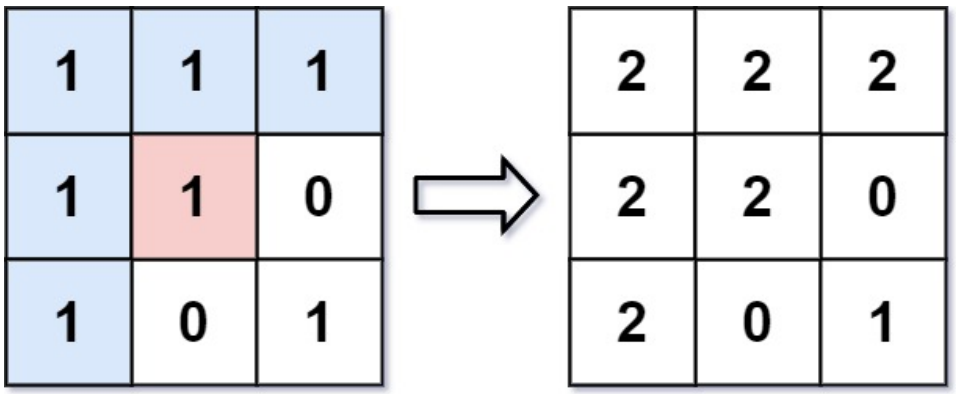
Input:

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 horizontally or vertically 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 with 0, which is the same as the target color. Therefore, 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

Code Snippets

C++:

```
class Solution {
public:
vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int
color) {

}
};
```

Java:

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

}
}
```

Python3:

```
class Solution:
def floodFill(self, image: List[List[int]], sr: int, sc: int, color: int) ->
List[List[int]]:
```

Python:

```
class Solution(object):
def floodFill(self, image, sr, sc, color):
"""
:type image: List[List[int]]
:type sr: int
:type sc: int
:type color: int
:rtype: List[List[int]]
"""
```

JavaScript:

```
/**
* @param {number[][]} image
* @param {number} sr
* @param {number} sc
* @param {number} color
* @return {number[][]}
*/
var floodFill = function(image, sr, sc, color) {

};
```

TypeScript:

```
function floodFill(image: number[][], sr: number, sc: number, color: number):
number[][] {

};
```

C#:

```
public class Solution {
public int[][] FloodFill(int[][] image, int sr, int sc, int color) {

}
}
```

C:

```
/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** floodFill(int** image, int imageSize, int* imageColSize, int sr, int
sc, int color, int* returnSize, int** returnColumnSizes) {

}
```

Go:

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

```
}
```

Kotlin:

```
class Solution {
fun floodFill(image: Array<IntArray>, sr: Int, sc: Int, color: Int):
Array<IntArray> {

}
}
```

Swift:

```
class Solution {
func floodFill(_ image: [[Int]], _ sr: Int, _ sc: Int, _ color: Int) ->
[[Int]] {

}
}
```

Rust:

```
impl Solution {
pub fn flood_fill(image: Vec<Vec<i32>>, sr: i32, sc: i32, color: i32) ->
Vec<Vec<i32>> {

}
}
```

Ruby:

```
# @param {Integer[][]} image
# @param {Integer} sr
# @param {Integer} sc
# @param {Integer} color
# @return {Integer[][]}
def flood_fill(image, sr, sc, color)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $image
* @param Integer $sr
* @param Integer $sc
* @param Integer $color
* @return Integer[][]
*/
function floodFill($image, $sr, $sc, $color) {

}
}
```

Dart:

```
class Solution {
List<List<int>> floodFill(List<List<int>> image, int sr, int sc, int color) {

}
}
```

Scala:

```
object Solution {
def floodFill(image: Array[Array[Int]], sr: Int, sc: Int, color: Int):
Array[Array[Int]] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec flood_fill(image :: [[integer]], sr :: integer, sc :: integer, color ::
integer) :: [[integer]]
def flood_fill(image, sr, sc, color) do

end
end
```

Erlang:

```
-spec flood_fill(Image :: [[integer()]], Sr :: integer(), Sc :: integer(),
Color :: integer()) -> [[integer()]].
flood_fill(Image, Sr, Sc, Color) ->
.
```

Racket:

```
(define/contract (flood-fill image sr sc color)
(-> (listof (listof exact-integer?)) exact-integer? exact-integer?
exact-integer? (listof (listof exact-integer?)))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Flood Fill
* Difficulty: Easy
* Tags: array, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int
color) {

}
};
```

Java Solution:

```
/**
* Problem: Flood Fill
* Difficulty: Easy
* Tags: array, search
*
```

```
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

}
}
```

Python3 Solution:

```
"""
Problem: Flood Fill
Difficulty: Easy
Tags: array, search

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def floodFill(self, image: List[List[int]], sr: int, sc: int, color: int) ->
List[List[int]]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def floodFill(self, image, sr, sc, color):
"""
:type image: List[List[int]]
:type sr: int
:type sc: int
:type color: int
:rtype: List[List[int]]
"""
```

JavaScript Solution:

```
/**
* Problem: Flood Fill
* Difficulty: Easy
* Tags: array, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number[][]} image
* @param {number} sr
* @param {number} sc
* @param {number} color
* @return {number[][]}
*/
var floodFill = function(image, sr, sc, color) {

};
```

TypeScript Solution:

```
/**
* Problem: Flood Fill
* Difficulty: Easy
* Tags: array, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function floodFill(image: number[][], sr: number, sc: number, color: number):
number[][] {

};
```

C# Solution:

```
/*
* Problem: Flood Fill
* Difficulty: Easy
* Tags: array, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public int[][] FloodFill(int[][] image, int sr, int sc, int color) {

}
}
```

C Solution:

```
/*
* Problem: Flood Fill
* Difficulty: Easy
* Tags: array, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** floodFill(int** image, int imageSize, int* imageColSize, int sr, int
sc, int color, int* returnSize, int** returnColumnSizes) {

}
```

Go Solution:

```
// Problem: Flood Fill
// Difficulty: Easy
// Tags: array, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

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

}
```

Kotlin Solution:

```
class Solution {
fun floodFill(image: Array<IntArray>, sr: Int, sc: Int, color: Int):
Array<IntArray> {

}
}
```

Swift Solution:

```
class Solution {
func floodFill(_ image: [[Int]], _ sr: Int, _ sc: Int, _ color: Int) ->
[[Int]] {

}
}
```

Rust Solution:

```
// Problem: Flood Fill
// Difficulty: Easy
// Tags: array, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn flood_fill(image: Vec<Vec<i32>>, sr: i32, sc: i32, color: i32) ->
```

```
Vec<Vec<i32>> {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} image
# @param {Integer} sr
# @param {Integer} sc
# @param {Integer} color
# @return {Integer[][]}
def flood_fill(image, sr, sc, color)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $image
* @param Integer $sr
* @param Integer $sc
* @param Integer $color
* @return Integer[][]
*/
function floodFill($image, $sr, $sc, $color) {

}
}
```

Dart Solution:

```
class Solution {
List<List<int>> floodFill(List<List<int>> image, int sr, int sc, int color) {

}
}
```

Scala Solution:

```
object Solution {
def floodFill(image: Array[Array[Int]], sr: Int, sc: Int, color: Int):
Array[Array[Int]] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec flood_fill(image :: [[integer]], sr :: integer, sc :: integer, color ::
integer) :: [[integer]]
def flood_fill(image, sr, sc, color) do

end
end
```

Erlang Solution:

```
-spec flood_fill(Image :: [[integer()]], Sr :: integer(), Sc :: integer(),
Color :: integer()) -> [[integer()]].
flood_fill(Image, Sr, Sc, Color) ->
.
```

Racket Solution:

```
(define/contract (flood-fill image sr sc color)
(-> (listof (listof exact-integer?)) exact-integer? exact-integer?
exact-integer? (listof (listof exact-integer?)))
)
```