1591. Strange Printer II

[Leetcode Link]

Problem Explanation

Given a strange printer that can print only rectangular patterns of a single color in a single turn and an m x n grid, we need to determine if it is possible to print the matrix targetGrid. The printer has two unique characteristics:

```
1 - On each turn, the printer will print a solid rectangular pattern of a single color on the grid.
2 - This will overwrite whatever color was previously in that rectangle.
3 - Once the printer has used a color for the above operation, the same color cannot be used again.
```

The colors in the targetGrid are represented as integers. If it's possible to print the targetGrid, return true. Otherwise, return false.

An important point to consider here is the printer's mechanic. As in Example 3, the targetGrid = [[1,2,1], [2,1,2], [1,2,1]], the output is "false" because you cannot print 2 after printing 1, because 2 is already covered by 1 from the left and right side.

Approach Explanation

To solve this problem we can create a directed graph where the edges represent a "covers" relationship. Each node represents a color, and there is a directed edge from one node to another if the color represented by the first node must be printed before the color represented by the second node.

Given this relationship, what we need to verify in order to answer the problem is if this graph has a cycle. If there's a cycle, it's impossible to print the targetGrid, because a color would need to be printed before and after another one, which is not allowed.

In the implementation, We initialize the graph and states arrays. `graph[u]` contains all colors v1, v2, ... etc, that are under the color u. `States[u]` keeps track of whether we've visited color u yet. If while doing a DFS traversal of graph we find a color that we'd encountered earlier in the current traversal chain, then we've detected a cycle and return false right away. If no cycles are detected for any color, then our printer can print the targetGrid and we return true.

Sample Walkthrough

Let's consider an example with the targetGrid matrix as [[1,1,1,1], [1,2,2,1], [1,2,2,1], [1,1,1,1]]

- The color 1 rectangle boundaries are minI=0, minJ=0, maxI=3 & maxJ=3
- The color 2 rectangle boundaries are minI=1, minJ=1, maxI=2 & maxJ=2
- Color 1 covers color 2. Hence, there would be a directed edge in the graph from color 2 to color 1.
- While traversing the graph there are no cyclical dependencies and hence it is possible to print the targetGrid and the function would return true.

Python Solution

```python

from collections import defaultdict

class Solution:
    def isPrintable(self, targetGrid: List[List[int]]) -> bool:
        colorRect = defaultdict(lambda: [float('inf'), float('inf'), float('-inf'), float('-inf')])

        for i in range(len(targetGrid)):
            for j in range(len(targetGrid[0])):
                color = targetGrid[i][j]
                colorRect[color][0] = min(colorRect[color][0], i)
                colorRect[color][1] = min(colorRect[color][1], j)
                colorRect[color][2] = max(colorRect[color][2], i)
                colorRect[color][3] = max(colorRect[color][3], j)

        graph = defaultdict(set)

        for color in colorRect:
            iStart, jStart, iEnd, jEnd = colorRect[color]
            for i in range(iStart, iEnd+1):
                for j in range(jStart, jEnd+1):
                    if targetGrid[i][j] != color:
                        graph[color].add(targetGrid[i][j])

        colorState = defaultdict(int)

        def hasCycle(v):
            colorState[v] = 1
            for u in graph[v]:
                if colorState[u] == 1 or ((u not in colorState or colorState[u] == 0) and hasCycle(u)):
                    return True
            colorState[v] = 2
            return False

        return not any(hasCycle(n) for n in range(1, 61) if n not in colorState or colorState[n] == 0)
```

In the python solution, for every color, a rectangle loop checks if there are any colors which can be covered by the current color. Then this creates the graph accordingly. We run a Depth-First Search (DFS) to detect a cycle in the graph. As soon as a cycle detected the function would return false. If there are no cycles then return true.# JavaScript Solution

The JavaScript solution follows a similar approach. Since JavaScript doesn't have equivalent Python's defaultdict and list comprehension features, we have improvised with the help of plain old JavaScript objects and Array map and filter methods.

```javascript
class Solution {
    isPrintable(targetGrid) {
        let colorRect = {};
        let n = targetGrid.length;
        let m = targetGrid[0].length;

        for(let i = 0; i < n; i++) {
            for(let j = 0; j < m; j++) {
                const color = targetGrid[i][j];
                if(!colorRect[color]) {
                    colorRect[color] = [i, j, i, j];
                } else {
                    let rect = colorRect[color];
                    rect[0] = Math.min(rect[0], i);
                    rect[1] = Math.min(rect[1], j);
                    rect[2] = Math.max(rect[2], i);
                    rect[3] = Math.max(rect[3], j);
                }
            }
        }

        let graph = {};
        for(let color in colorRect) {
            for(let i = colorRect[color][0]; i <= colorRect[color][2]; i++) {
                for(let j = colorRect[color][1]; j <= colorRect[color][3]; j++) {
                    if(targetGrid[i][j] != color) {
                        if(!graph[color]) graph[color] = new Set();
                        graph[color].add(targetGrid[i][j]);
                    }
                }
            }
        }

        let colorState = {};
        let hasCycle = v => {
            if(colorState[v] == 1) return true;
            if(colorState[v] == 2) return false;
            colorState[v] = 1;
            if(graph[v]) {
                for(let u of graph[v]) {
                    if(hasCycle(u)) return true;
                }
            }
            colorState[v] = 2;
            return false;
        };

        for(let i in colorRect) {
            if(hasCycle(i)) return false;
        }

        return true;
    }
}
```

Java Solution

Here's how you would implement the solution in Java. Since it's a statically typed language, defining the data structures is a bit more verbose compared to Python and Javascript.

```java
class Solution {
    public boolean isPrintable(int[][] targetGrid) {
        HashMap<Integer, int[]> pos = new HashMap<>();
        for (int i = 0; i < targetGrid.length; i++) {
            for (int j = 0; j < targetGrid[0].length; j++) {
                int[] p = pos.getOrDefault(targetGrid[i][j], new int[]{i, j, i, j});
                p[0] = Math.min(p[0], i);
                p[1] = Math.min(p[1], j);
                p[2] = Math.max(p[2], i);
                p[3] = Math.max(p[3], j);
                pos.put(targetGrid[i][j], p);
            }
        }

        ArrayList<HashSet<Integer>> graph = new ArrayList<>();
        for (int i = 0; i < 61; i++) graph.add(new HashSet<Integer>());
        int[] colorState = new int[61];

        for (int[] p: pos.values()) {
            for (int i = p[0]; i <= p[2]; i++)
                for (int j = p[1]; j <= p[3]; j++)
                    if (targetGrid[i][j] != pos.get(targetGrid[i][j])) graph.get(targetGrid[i][j]).add(pos.get(targetGrid[i][j]));
        }

        for (int i = 1; i <= 60; i++) {
            if (dfs(i, colorState, graph)) return false;
        }

        return true;
    }

    private boolean dfs(int i, int[] colorState, ArrayList<HashSet<Integer>> graph) {
        if (colorState[i] > 0) return colorState[i] == 1;
        colorState[i] = 1;
        for (int j: graph.get(i)) if (dfs(j, colorState, graph)) return true;
        colorState[i] = 2;
        return false;
    }
}
```

In the Java solution, it contains a depth first search method (dfs), which is used to detect a cycle in our graph. It takes in the current color (i), the colorState, and the graph as arguments. It checks if colorState[i] > 0 if so it returns if the colorState is equal to 1. If not, it sets colorState[i] to 1 and proceeds to loop through each color in the graph that is connected to the current color. If there is a cycle, it returns true, else it sets colorState[i] to 2 and returns false. The main isPrintable function makes use of this method and ultimately, the solution would return true if the grid is printable and false in the case that it isn't, based on the logic described above.