2132. Stamping the Grid

Hard Greedy Array Matrix Prefix Sum Leetcode Link

Problem Description

The problem presents a grid consisting of cells marked either 0 (empty) or 1 (occupied). The challenge is to cover all the empty cells with stamps of a pre-defined size (`stampHeight x stampWidth`) without overlapping any occupied cells. Stamps can overlap each other, they cannot be rotated, and they must fit entirely within the grid boundaries.

Intuition

To solve this problem, we first need to quickly determine whether a stamp can be placed onto a certain position on the grid. The key is to check that all cells under the stamp are empty. To do this efficiently, we employ a "prefix sum matrix." A prefix sum allows us to calculate the sum of any sub-matrix in constant time.

The prefix sum is constructed such that each cell in this auxiliary matrix contains the sum of all cells above and to the left, including the current cell in the original grid. With this prefix sum matrix, we can quickly determine the sum of any sub-matrix by subtracting the appropriate prefix sums.

Now, when placing a stamp on the grid, we mark the cells in the difference matrix that corresponds to the stamp's area. A difference matrix allows us to record changes to a range within the matrix in constant time. By incrementing the top-left corner of the stamp area and decrementing the point just outside the bottom-right corner of the proposed stamp area, we define a range that can receive a stamp.

Next, we build another prefix sum from the difference matrix. This new matrix helps us understand how many stamps cover each cell. As we iterate through the entire grid, we check each empty cell to see if it has been covered by at least one stamp. If we find any empty cell not covered by a stamp, we know the task is impossible, and we return `false`.

If all empty cells are covered, we've met the requirements and can return `true`. Each step of the solution builds on a logical progression from determining single-cell coverage to ensuring full-grid compliance with the stamp placement rules.

Solution Approach

The provided solution can be broken down into the following steps, utilizing concepts such as prefix sums and difference matrices:

1. **Constructing the Prefix Sum Matrix (`s`):** The prefix sum matrix is built so that each cell represents the sum of all cells above and to the left in the grid, including the cell itself. This is calculated by `s[i + 1][j + 1] = s[i + 1][j] + s[i][j + 1] - s[i][j] + grid[i][j]`. This step is the foundation for efficiently checking whether a stamp can be placed in a certain area.
2. **Stamp Placement Check:** For a given cell `(i, j)` that we are trying to cover with a stamp, we check whether the sub-matrix defined by the bottom-right corner `(x, y)` — where `x` equals `i + stampHeight` and `y` equals `j + stampWidth` — is within the bounds and contains only zeroes. This is done by verifying `s[x][y] - s[x][j] - s[i][y] + s[i][j] == 0`.
3. **Updating the Difference Matrix (`d`):** If the stamp can be placed, we update the difference matrix to reflect this. We increment `d[i][j]` and decrement `d[i][y]`, `d[x][j]`, and `d[x][y]` to ensure that the affected range captures the stamp placement.
4. **Applying the Difference Matrix:** Another two-dimensional prefix sum is calculated from the difference matrix. For each cell `(i, j)`, `cnt[i + 1][j + 1]` is updated to the sum of the current cell plus the cells above and to the left. This represents how many stamps cover each specific cell.
5. **Validation Check:** As we iterate through each cell in the grid again, we verify if it's empty `grid[i][j] == 0`. If it's uncovered by any stamp `cnt[i + 1][j + 1] == 0`, we immediately return `false` as the requirement is violated.
6. **Returning the Result:** If all empty cells are covered without breaking any of the rules (all visited cells pass the validation check), we conclude that it's possible to stamp all empty cells and return `true`.

By using a prefix sum to enable quick sum calculations of sub-matrices and a difference matrix to handle the range updates, the solution efficiently determines whether it's possible to satisfy the stamp placement conditions for the entire grid.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using a grid of size `3x4` with a stamp size of `2x2`.

Consider the grid:

```
0 1 0 0
1 0 0 0
0 0 0 0
```

Let's go through the solution steps on this grid:

1. **Construct Prefix Sum Matrix (`s`):** We start by creating a prefix sum matrix `s` of size `4x5` (one more in each dimension for easier calculation).

```
To build prefix sum matrix `s`, we follow the rule: s[i + 1][j + 1] = s[i + 1][j] + s[i][j + 1] - s[i][j] + grid[i][j].

Starting with `s` all zeros:
0 0 0 0 0
0
0
0

We add each cell of the grid, cumulatively:
0 0 0 0 0
0 0 1 1 1
0 1 1 2 2
0 1 2 3 3
```

2. **Stamp Placement Check:** Let's check if we can place the `2x2` stamp at the top-left corner `(0,0)`.

   We check the sub-matrix given by `s[2][2]` is zero:

```
s[2][2] - s[0][2] - s[2][0] + s[0][0] = 1 - 0 - 1 + 0 = 0
```

   Since it's not zero, we cannot place the stamp here because there's an occupied cell `(1,0)` within the range of the stamp.

3. **Updating the Difference Matrix (`d`):** Now, let's attempt to place a stamp at `(2,0)`. We can confirm it fits by checking the respective prefix sum sub-matrix. Since it fits, we update the difference matrix `d`:

```
d starts as all zeros:
0 0 0 0 0
0
0
0

We increment `d[2][0]` and decrement the cells just outside the bottom-right of stamp area `d[4][2]`:
0 0 0 0 0
0
0 1
0   -1
```

4. **Applying the Difference Matrix:** Next, we construct a new prefix sum from the `d`:

```
We calculate using d's values:
0  0  0  0  0
0  0  0  0  0
0  1  1  1  1
0  1  1  1  0
```

   This matrix now indicates the number of stamps covering each cell.

5. **Validation Check:** We go back to our original grid and check each cell:
   - `grid[0][0] == 0` and `cnt[1][1] == 0` (no stamp covers this cell), so we return `false`.
6. **Returning the Result:** There is no need to proceed because we already determined that it's impossible to cover all empty cells with stamps, as per the previous step's validation check.

Using these steps serves to highlight how the algorithm leverages the prefix sum and difference matrix to efficiently solve the problem. In this example, the given grid configuration and stamp size do not allow us to cover all empty cells without overlapping an occupied cell. Therefore, the final output would be `false`.

Python Solution

```python
class Solution:
    def possibleToStamp(self, grid: List[List[int]], stampHeight: int, stampWidth: int) -> bool:
        # Get the dimensions of the grid
        rows, cols = len(grid), len(grid[0])

        # Initialize prefix sum matrix
        prefix_sum = [[0] * (cols + 1) for _ in range(rows + 1)]
        # Calculate the prefix sum of the grid
        for i in range(rows):
            for j in range(cols):
                prefix_sum[i + 1][j + 1] = (
                    prefix_sum[i][j + 1] + prefix_sum[i + 1][j]
                    - prefix_sum[i][j] + grid[i][j]
                )

        # Initialize difference matrix for stamp placements
        diff_matrix = [[0] * (cols + 1) for _ in range(rows + 1)]
        # Determine if a stamp can be placed on an empty area
        for i in range(rows):
            for j in range(cols):
                if grid[i][j] == 0:
                    row_end, col_end = i + stampHeight, j + stampWidth
                    # Check if the stamp fits in the current position
                    if row_end <= rows and col_end <= cols and prefix_sum[row_end][col_end] - prefix_sum[row_end][j] - prefix_sum[i
                        diff_matrix[i][j] += 1
                        diff_matrix[i][col_end] -= 1
                        diff_matrix[row_end][j] -= 1
                        diff_matrix[row_end][col_end] += 1

        # Create matrix to keep track of covered cells
        coverage_matrix = [[0] * (cols + 1) for _ in range(rows + 1)]
        # Apply the difference matrix to calculate the number of stamps covering each cell
        for i in range(rows):
            for j in range(cols):
                coverage_matrix[i + 1][j + 1] = (
                    coverage_matrix[i + 1][j] + coverage_matrix[i][j + 1]
                    - coverage_matrix[i][j] + diff_matrix[i][j]
                )
                # If a cell is supposed to be stamped but has no stamp coverage, return False
                if grid[i][j] == 0 and coverage_matrix[i + 1][j + 1] == 0:
                    return False
        # If every empty cell is covered appropriately with stamps, return True
        return True
```

Java Solution

```java
class Solution {
    public boolean possibleToStamp(int[][] grid, int stampHeight, int stampWidth) {
        int numRows = grid.length, numCols = grid[0].length;
        // Prefix sum array to quickly calculate sum of submatrices.
        int[][] prefixSum = new int[numRows + 1][numCols + 1];

        // Build the prefixSum array.
        for (int row = 0; row < numRows; ++row) {
            for (int col = 0; col < numCols; ++col) {
                prefixSum[row + 1][col + 1] = prefixSum[row + 1][col] + prefixSum[row][col + 1] - prefixSum[row][col] + grid[row][c
            }
        }

        // Difference array to apply range updates (stamping).
        int[][] diff = new int[numRows + 1][numCols + 1];

        // Iterate over the entire grid to check where stamps can be placed.
        for (int row = 0; row < numRows; ++row) {
            for (int col = 0; col < numCols; ++col) {
                // If we have an empty cell and can fit a stamp starting at (row, col),
                // then update the difference array.
                if (grid[row][col] == 0) {
                    int endRow = row + stampHeight, endCol = col + stampWidth;
                    if (endRow <= numRows && endCol <= numCols &&
                        prefixSum[endRow][endCol] - prefixSum[endRow][col] - prefixSum[row][endCol] + prefixSum[row][col] == 0) {
                        diff[row][col]++;
                        diff[row][endCol]--;
                        diff[endRow][col]--;
                        diff[endRow][endCol]++;
                    }
                }
            }
        }

        // Use a running sum to apply the difference array updates to the grid.
        int[][] coverCount = new int[numRows + 1][numCols + 1];
        for (int row = 0; row < numRows; ++row) {
            for (int col = 0; col < numCols; ++col) {
                coverCount[row + 1][col + 1] = coverCount[row + 1][col] + coverCount[row][col + 1] - coverCount[row][col] + diff[ro
                // If there is an empty cell that is not covered by a stamp, return false.
                if (grid[row][col] == 0 && coverCount[row + 1][col + 1] == 0) {
                    return false;
                }
            }
        }

        // All empty cells are covered by stamps; return true.
        return true;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    bool possibleToStamp(vector<vector<int>>& grid, int stampHeight, int stampWidth) {
        int rows = grid.size(), cols = grid[0].size();
        // Use an auxiliary matrix to perform prefix sum computations
        vector<vector<int>> prefixSum(rows + 1, vector<int>(cols + 1));
        // Calculate the prefix sums for all cells
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                prefixSum[i + 1][j + 1] = prefixSum[i + 1][j] + prefixSum[i][j + 1] - prefixSum[i][j] + grid[i][j];
            }
        }

        // Initialize a difference matrix to mark stampable regions
        vector<vector<int>> diff(rows + 1, vector<int>(cols + 1));
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // If the current cell is filled, it cannot be stamped, skip it
                if (grid[i][j]) continue;
                int x = i + stampHeight, y = j + stampWidth;
                // Check if it's possible to stamp the area starting at (i, j)
                if (x <= rows && y <= cols && prefixSum[x][y] - prefixSum[i][y] - prefixSum[x][j] + prefixSum[i][j] == 0) {
                    // Mark corners of the stamp region in the difference matrix
                    diff[i][j]++;
                    diff[x][j]--;
                    diff[i][y]--;
                    diff[x][y]++;
                }
            }
        }

        // Initialize a matrix to hold the stamp count for each cell
        vector<vector<int>> stampCount(rows + 1, vector<int>(cols + 1));
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // Calculate the cumulative stamp count for the current cell
                stampCount[i + 1][j + 1] = stampCount[i + 1][j] + stampCount[i][j + 1] - stampCount[i][j] + diff[i][j];
                // If the current cell is empty and has no stamps, return false
                if (grid[i][j] == 0 && stampCount[i + 1][j + 1] == 0) return false;
            }
        }

        // If all constraints are satisfied, it's possible to stamp
        return true;
    }
};
```

Typescript Solution

```typescript
// Function to determine if it's possible to stamp the entire grid
// with a stamp of given height and width.
/**
 * @param grid - 2D grid representing the areas to be stamped (0) or not (1)
 * @param stampHeight - Height of the stamp
 * @param stampWidth - Width of the stamp
 * @returns boolean indicating whether it's possible to stamp the whole grid
 */
function possibleToStamp(grid: number[][], stampHeight: number, stampWidth: number): boolean {
    const m: number = grid.length;
    const n: number = grid[0].length;

    // Initialize prefixes sums arrays with zeros
    let prefixSums: number[][] = new Array(m + 1).fill(0).map(() => new Array(n + 1).fill(0));
    let diff: number[][] = new Array(m + 1).fill(0).map(() => new Array(n + 1).fill(0));
    let count: number[][] = new Array(m + 1).fill(0).map(() => new Array(n + 1).fill(0));

    // Compute the prefix sums of the grid
    for (let i = 0; i < m; ++i) {
        for (let j = 0; j < n; ++j) {
            prefixSums[i + 1][j + 1] = prefixSums[i + 1][j] + prefixSums[i][j + 1] - prefixSums[i][j] + grid[i][j];
        }
    }

    // Determine where stamping is possible and mark in the diff array
    for (let i = 0; i < m; ++i) {
        for (let j = 0; j < n; ++j) {
            if (grid[i][j] == 0) {
                let x: number = i + stampHeight;
                let y: number = j + stampWidth;
                if (x <= m && y <= n && prefixSums[x][y] - prefixSums[i][y] - prefixSums[x][j] + prefixSums[i][j] == 0) {
                    diff[i][j]++;
                    diff[i][y]--;
                    diff[x][j]--;
                    diff[x][y]++;
                }
            }
        }
    }

    // Calculate the influence of stamping using prefix sums of the diff array
    for (let i = 0; i < m; ++i) {
        for (let j = 0; j < n; ++j) {
            count[i + 1][j + 1] = count[i + 1][j] + count[i][j + 1] - count[i][j] + diff[i][j];
            if (grid[i][j] == 0 && count[i + 1][j + 1] == 0) {
                return false; // Unstamped cell found, stamping not possible
            }
        }
    }

    return true; // All cells can be stamped
}

// Example usage
// let result: boolean = possibleToStamp([[1, 0, 0, 0], [1, 0, 0, 0], [1, 1, 1, 0]], 2, 2);
// console.log(result); // Should return true or false based on stampability of grid
```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by several nested loops that iterate over the entire `grid` and the use of prefix sums.

1. The first double loop (calculating `s[i + 1][j + 1]`) iterates through all `m * n` cells of the grid once, thus it has a complexity of `O(m * n)`.
2. The second double loop (calculating `d[i][j]` and checking conditions) also iterates through all `m * n` cells of the grid once. Inside this loop, it performs constant-time operations and a check that involves accessing the precomputed prefix sum array `s`. Again, the complexity is `O(m * n)`.
3. The last double loop (calculating `cnt[i + 1][j + 1]` and verifying that there are no zeros uncovered by stamps) iterates through all `m * n` cells of the grid. Since it only performs constant-time operations, the complexity is `O(m * n)`.

Since these loops are sequential and not nested within each other (other than the initialization loops for prefix sums which also have the same complexity), the overall time complexity is the sum of the individual complexities, which remains `O(m * n)` because the number of iterations is proportional to the size of the grid.

Space Complexity

The space complexity is determined by additional arrays `s`, `d`, and `cnt` which are used for computing prefix sums and keeping track of the stamps.

1. The array `s` has dimensions `(m + 1) x (n + 1)` which adds up to a space complexity of `O((m + 1) * (n + 1))`. However, when considering Big O notation, constant factors are dropped, so the complexity is `O(m * n)`.
2. Similarly, the arrays `d` and `cnt` also have dimensions `(m + 1) x (n + 1)`, contributing an additional `O(m * n)` space complexity each.

The space needed for the input array `grid` is not considered in the space complexity analysis since it is the input to the function, not additional space allocated by the function itself.

In total, since all three arrays are maintained independently, the overall space complexity is `O(3 * m * n)`, which simplifies to `O(m * n)` under Big O notation since constant factors are ignored.

Therefore, the final space complexity is `O(m * n)`.