750. Number Of Corner Rectangles

Medium | Array | Math | Dynamic Programming | Matrix

[Leetcode Link]

Problem Description

The given LeetCode problem asks us to count the number of distinct "corner rectangles" in a 2D grid. A "corner rectangle" is defined by four '1's that are placed in the grid in such a way that they form the corners of an axis-aligned rectangle. Importantly, the interior of this rectangle can contain any mix of '0's and '1's; the only requirement is that the four corners are '1's. The grid, `grid`, is composed entirely of '0's and '1's, and its dimensions are `m x n` where `m` is the number of rows and `n` is the number of columns. The task is to return the total count of such distinct corner rectangles.

Intuition

To solve this problem, we leverage the fact that a rectangle is defined by its two opposite corners. Here, we iterate over all potential corner pairs in the grid and try to determine if these pairs can form the upper two or lower two corners of a rectangle. We do this by checking if we picked two '1's in the same row. For each pair of '1's in the row, we check to see if the same pair of columns also have '1's in another row, forming a rectangle. To efficiently track this, we employ a counting strategy that uses a dictionary to remember how many times we've seen each pair of '1's in the same row.

Whenever we encounter a pair of '1's in the same row, we check the dictionary to see if this pair has been seen before. If it has, then for each previous occurrence, there is a distinct rectangle. Therefore, the count of rectangles is incremented by the number of times the pair has been seen before. Afterwards, we increment the counter for that pair, indicating that we've found another potential set of top corners for future rectangles.

The algorithm iterates over all rows and, within each row, all pairs of '1's. By using a counter dictionary that keeps track of pairs of indices where '1's are found, it ensures that the process of finding rectangular corners is efficient, avoiding checking every possible rectangle explicitly, which would be computationally expensive especially for large grids.

Solution Approach

The solution employs a simple yet efficient approach using a hash table to keep track of the count of '1' pairs encountered so far. Here's a step-by-step walkthrough of how the code works:

1. **Initialize Variables**:
   - Initialize a variable `ans` to 0, which will hold the final count of corner rectangles.
   - Create a `Counter` object named `cnt` from the Python `collections` module. This counter will map each pair of column indices `(i, j)` to the number of times that pair has been observed in `grid` with '1's.
2. **Iterate Over the Grid**:
   - The first `for` loop iterates through each row of the grid.
3. **Find Pairs of '1's in the Same Row**:
   - Within each row, use a nested `for` loop with `enumerate` to get both the index `i` and value `c1` at that index.
   - If `c1` is '1', it means we have a potential top/left corner of a rectangle.
4. **Check for Potential Top/Right Corners**:
   - Another nested `for` loop is used to check to the right of the current '1' for another '1' at index `j`, forming a pair of top corners `(i, j)` of a potential rectangle.
5. **Count and Update Rectangles**:
   - For each such pair `(i, j)`, if they can form the top two corners of a rectangle, we increment `ans` by `cnt[(i, j)]` since each count represents another row where a rectangle with these top corners can be completed.
   - After counting the rectangles for `(i, j)`, update `cnt[(i, j)]` by incrementing by 1 which signifies that we have found another pair of '1's that can form the top corners of potential future rectangles.
6. **Return the Answer**:
   - After all rows and valid pairs are processed, return `ans` as the total count of corner rectangles found in the grid.

This solution utilizes a smart counting approach to avoid checking each potential rectangle directly, which drastically reduces the time complexity. The main algorithmic techniques include iterating over array elements and using hash-based counters for an elegant and efficient solution. By checking only rows for pairs of '1's and then counting and updating the number of potential rectangles as more pairs are found, the solution effectively captures all corner rectangles without unnecessary computations.

Example Walkthrough

Let's walk through a small example to illustrate how the solution approach works. Consider the following `grid` which is a 4×3 matrix containing both `0's` and `1's`.

```
grid = [
    [1, 0, 1],
    [1, 1, 1],
    [1, 0, 0],
    [1, 0, 1]
]
```

1. **Initialize Variables**:
   - `ans` starts at 0.
   - `cnt` is an empty `Counter` object.
2. **Iterate Over the Grid**:
   - Start with the first row `[1, 0, 1]`.
3. **Find Pairs of '1's in the Same Row**:
   - Identify two '1's in the row at index `i = 0` and `j = 2`.
4. **Check for Potential Top/Right Corners**:
   - These are the potential top corners of multiple rectangles.
5. **Count and Update Rectangles**:
   - Since `cnt[(0, 2)]` is not in `cnt`, `ans` remains 0, but `cnt[(0, 2)]` becomes 1.

Repeat steps 2-5 for remaining rows:

- Second row `[1, 1, 1]` has '1's at indices `0`, `1`, and `2`.
- Pairs `(0, 1)`, `(0, 2)`, and `(1, 2)` each have the potential to form corner rectangles.
- `cnt[(0, 1)]` becomes 1, `cnt[(0, 2)]` is incremented to 2 (since we already have one from the first row), and `cnt[(1, 2)]` becomes 1.

As we process this row, `ans` gets updated because `cnt[(0, 2)]` was already 1 (one rectangle can now be formed with the previous row as the bottom corners).

Now, `ans` is incremented by 1.

Continuing on to the third and fourth rows, you'll do the same pair checks and counts, but no updates to `ans` occur until the fourth row.

- In the fourth row `[1, 0, 1]`, the same pair `(0, 2)` is found as in the first row, meaning that for every previous occurrence of this pair (which is two times so far), a rectangle can be completed, so `ans` gets incremented by 2.

Finally, after iterating through all rows, we conclude with:

- `ans` = 3 (total count of corner rectangles found in the grid).

This example illuminated how only pairs of '1's in the same rows are used to determine the possibility of forming rectangles, and with each row's pairs, we effectively keep a running count of potential rectangles without the need for checking all possible rectangles directly. The `Counter` efficiently handles this incrementation and checking for previously seen pairs.

Python Solution

```python
from collections import Counter

class Solution:
    def countCornerRectangles(self, grid: List[List[int]]) -> int:
        rectangle_count = 0  # This will hold the final count of corner rectangles
        pair_counter = Counter()  # Counter to track pairs of columns that have a 1 at the same row
        num_cols = len(grid[0])  # The number of columns in the grid

        # Iterate through each row in the grid
        for row in grid:
            # Enumerate over the row to get both column index and value
            for col_index_first, cell_first in enumerate(row):
                # Only process if the cell at the current column has a 1
                if cell_first:
                    # Consider pairs of columns, starting from the current one
                    for col_index_second in range(col_index_first + 1, num_cols):
                        # Check if the second column also has a 1 at the current row
                        if row[col_index_second]:
                            # If both columns have a 1 at the current row, this forms a potential rectangle corner
                            # Increase the count for this column pair as we found another rectangle corner
                            rectangle_count += pair_counter[(col_index_first, col_index_second)]
                            # Update the counter for the current pair, adding one more occurrence
                            pair_counter[(col_index_first, col_index_second)] += 1

        # Return the total count of rectangles found
        return rectangle_count
```

Changes and comments explanation:

1. `rectangle_count`: Renamed `ans` to `rectangle_count` to better describe what the variable is used for.
2. `pair_counter`: Renamed `cnt` to `pair_counter` which is a Counter object to keep track of pairs of columns that make the corners of rectangles.
3. `num_cols`: Introduced this variable as a clearer name for the number of columns in the grid (`n` in the original code).
4. `col_index_first` and `cell_first`: Renamed `i` and `c1` for clarity in the enumeration of the first column index and cell value.
5. Comments: Added explanatory comments throughout the code to provide clarity on each step of the process.
6. Standard imports: Included the import statement for the 'Counter' class from the 'collections' module explicitly, which allows for counting occurrences of elements in a hashable object.

Make sure to replace `List` by the correct import statement from 'typing' at the beginning of the file:

```python
from typing import List

```

Java Solution

```java
import java.util.HashMap;
import java.util.List;
import java.util.Map;

class Solution {
    public int countCornerRectangles(int[][] grid) {
        // Number of columns in the grid.
        int numCols = grid[0].length;
        // This will store the final count of corner rectangles.
        int cornerRectanglesCount = 0;
        // A map to store the counts of 1's pairs across rows.
        Map<List<Integer>, Integer> pairsCount = new HashMap<>();

        // Loop through each row in the grid.
        for (int[] row : grid) {
            // Iterate over every possible pair of columns within this row
            for (int leftCol = 0; leftCol < numCols; ++leftCol) {
                // If the current cell is a 1, explore further for a rectangle.
                if (row[leftCol] == 1) {
                    for (int rightCol = leftCol + 1; rightCol < numCols; ++rightCol) {
                        // Only if the paired cell is also a 1, do we consider it.
                        if (row[rightCol] == 1) {
                            // Create a pair to check in our current map.
                            List<Integer> pair = List.of(leftCol, rightCol);
                            // Increment the count of found rectangles with these 1's as the top corners.
                            cornerRectanglesCount += pairsCount.getOrDefault(pair, 0);
                            // Increment the count of this pair in our map.
                            pairsCount.merge(pair, 1, Integer::sum);
                        }
                    }
                }
            }
        }
        // The result is the total count of rectangles found.
        return cornerRectanglesCount;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <map>
using namespace std;

class Solution {
public:
    int countCornerRectangles(vector<vector<int>>& grid) {
        // n represents the number of columns in the grid
        int num_columns = grid[0].size();
        // Initialize the answer to 0
        int answer = 0;
        // Define a map to store the count of pairs of columns that form the vertical sides of potential rectangles
        map<pair<int, int>, int> column_pairs_count;

        // Iterate through each row of the grid
        for (const auto& row : grid) {
            // Check each pair of columns within the row
            for (int i = 0; i < num_columns; ++i) {
                // If the current cell contains a 1, search for a potential second column to form a rectangle
                if (row[i]) {
                    for (int j = i + 1; j < num_columns; ++j) {
                        // If we find a pair of 1s, this could form the corner of a rectangle
                        if (row[j]) {
                            // Increase the answer by the count of rectangles that can be formed using this column pair
                            answer += column_pairs_count[{i, j}];
                            // Increment the count for this column pair
                            ++column_pairs_count[{i, j}];
                        }
                    }
                }
            }
        }
        // Return the total count of corner rectangles
        return answer;
    }
};

```

Typescript Solution

```typescript
function countCornerRectangles(grid: number[][]): number {
    // Initialization of n to represent the number of columns
    const columnsCount = grid[0].length;
    // Initialization of the answer variable to count corner rectangles
    let cornerRectanglesCount = 0;
    // Using a Map to keep track of the count of pairs of cells with value 1
    const pairCounts: Map<number, number> = new Map();

    // Looping through each row in the grid
    for (const row of grid) {
        // Looping through each cell in the row
        for (let i = 0; i < columnsCount; ++i) {
            // Check if the current cell has value 1
            if (row[i] === 1) {
                // Nested loop to find another cell with value 1 in the same row
                for (let j = i + 1; j < columnsCount; ++j) {
                    if (row[j] === 1) {
                        // Creating a unique key for the pair of cells
                        const pairKey = i * 200 + j;
                        // Increment count for the current pair of cells
                        cornerRectanglesCount += pairCounts.get(pairKey) ?? 0;
                        // Update the pair count map with the new count
                        pairCounts.set(pairKey, (pairCounts.get(pairKey) ?? 0) + 1);
                    }
                }
            }
        }
    }

    // Returning the total count of corner rectangles found in the grid
    return cornerRectanglesCount;
}

```

Time and Space Complexity

Time Complexity

The given code's time complexity primarily comes from the nested loops that it uses to iterate over each pair of columns for each row.

- The first loop (`for row in grid`) goes through each row in the grid, which occurs `m` times where `m` is the number of rows.
- Inside this loop, the second loop (`for i, c1 in enumerate(row)`) iterates over each element in the row, which occurs `n` times where `n` is the number of columns.
- The innermost loop (`for j in range(i + 1, n)`) iterates from the current column `i` to the last column, with the average number of iterations being roughly `n/2` since it's a triangular iteration overall.

Considering the iterations of the third nested loop, the number of column pairs `(i, j)` that will be considered for each row follows the pattern of a combination of selecting two out of `n` columns or `nC2`.

Therefore, the time complexity is `O(m * nC2)` or `O(m * n * (n - 1) / 2)`, which simplifies to `O(m * n^2)`.

Space Complexity

The space complexity is determined by the storage used by the `cnt` Counter, which keeps track of the frequency of each pair of columns that have been seen with `'1'` at both the `i`th and `j`th positions.

- In the worst case scenario, the `cnt` Counter would have an entry for every possible pair of columns. As there are `n` columns, the number of column pairs would be the same `nC2` we calculated before, or `n * (n - 1) / 2` pairs.
- Therefore, the space complexity for the `cnt` Counter is `O(n^2)`.

Thus, the overall space complexity is `O(n^2)` since the Counter's size is the dominant term, and this space is additional to the input (`grid`) since the `grid` itself is not being modified.