807. Max Increase to Keep City Skyline

Medium Greedy Array Matrix Leetcode Link

Problem Description

In the given problem, we have a simulated city that's made up of `n x n` blocks, and each block contains a building. The `grid` matrix provided as input indicates the heights of these buildings, with each cell `grid[r][c]` showing the height of the building at the `r` row and `c` column. The city's skyline is the outer contour of buildings when they are viewed from far away in any of the cardinal (North, East, South, West) directions. What we need to determine is the maximum height we can add to each of the buildings without changing the skyline from any direction. The question asks for the total sum of the heights that we could add to these buildings.

Understanding the problem involves realizing that:

- We can increase the height of any building.
- The increase in height for each building may be different.
- The skyline must not be altered by these height increases.

The key idea here is that the skyline is determined by the tallest buildings in each row when viewing from the East or West, and in each column when viewed from the North or South.

Intuition

To arrive at the solution:

1. We first determine what the current skyline looks like from each direction. For the North/South view, we need the maximum heights in each column, and for the East/West perspective, we need the maximum heights in each row.

2. The key insight is that the maximum height to which a particular building can be increased without altering the skyline is the minimum of the maximums of its row and column.

3. For each building, we calculate this potential increase by taking the minimum of the two maximum heights (of the row and column it's in) and subtracting the current height of the building. This is how we abide by the condition of not changing the skyline after increasing the building's height.

4. By summing these potential increases for each building, we get the total sum that the heights of buildings can be increased by without affecting the skyline.

In terms of implementation:

- First, `rmx` stores the maximum height for each row while `cmx` stores the maximum heights for each column, which are the skylines when looking from East/West and North/South respectively.
- Then, we iterate over each cell in the grid, and for each building, calculate the potential increase as the difference between the smaller of the two maximum heights (`min(rmx[i], cmx[j])`) and the current height (`grid[i][j]`).
- The `sum()` function accumulates these positive differences to provide the answer: the total sum of height increases across the grid.

Solution Approach

The implementation of the solution utilizes a simple yet efficient approach combining elements of array manipulation and mathematics.

Here's a step-by-step breakdown of the implementation:

1. **Calculate Row Maxima (`rmx`):**
   - We iterate through each row of the `grid` matrix to find the maximum height of the buildings in each row.
   - The built-in `max()` function applied to each row results in a list of the tallest building per row, which represents the East/West skyline.
   - This list is stored in the variable `rmx`.
2. **Calculate Column Maxima (`cmx`):**
   - We use the `zip(*grid)` function to transpose the original `grid` matrix. This effectively converts the columns into rows for easy traversal.
   - Applying the `max()` function on the transposed grid gives us the maximum heights for each column, which presents the North/South skyline.
   - The resulting maximum values for each column are stored in the variable `cmx`.
3. **Evaluate the Height Increase Limit:**
   - The solution utilizes a nested for-loop to iterate through each cell in the `grid`. For each cell located at `(i, j)`, it calculates the minimum height that can be achieved between the maximum heights of the row and column which the cell belongs to, using `min(rmx[i], cmx[j])`.
   - It subtracts the current building height `grid[i][j]` from this minimum value to find the possible height increase without changing the skyline. This calculation is in direct correlation with the mathematical definition of the problem statement.
4. **Summing the Results:**
   - Each of these height increase values for the individual buildings are added together using the `sum()` function. The addition is done through a generator expression which runs through each cell coordinate `(i, j)` and applies the previous step's logic.
   - This sum is the maximum total sum that building heights can be increased without affecting the city's skyline from any cardinal direction.

The algorithm's essence is in leveraging the minimum of the maximum heights of rows and columns to find the optimal height increase for each building. A crucial factor is that the algorithm runs in `O(n^2)` time complexity, where `n` is the dimension of the input matrix, making it suitable for reasonably large values of `n`. No additional space is used besides the two arrays storing maxima of rows and columns, resulting in `O(n)` space complexity.

Example Walkthrough

Let's consider an example with a `3 x 3` grid, represented by the matrix:

```
grid = [
    [3, 0, 8],
    [4, 5, 7],
    [9, 2, 6]
]
```

Following the steps outlined in the solution approach:

1. **Calculate Row Maxima (`rmx`):**

   We look for the tallest building in each row (East/West skyline):

   - Row 0: max is 8.
   - Row 1: max is 7.
   - Row 2: max is 9.

   So, `rmx = [8, 7, 9]`.

2. **Calculate Column Maxima (`cmx`):**

   By transposing the `grid` and finding the tallest building in each column (North/South skyline):

   - Column 0: max is 9 (from transposed grid row 0).
   - Column 1: max is 5 (from transposed grid row 1).
   - Column 2: max is 8 (from transposed grid row 2).

   Hence, `cmx = [9, 5, 8]`.

3. **Evaluate the Height Increase Limit:**

   We iterate over the grid and for each cell `(i, j)`, we find the minimum of the maximum height of the ith row and jth column (`min(rmx[i], cmx[j])`) to determine the limit to which we can raise each building:

   At `grid[0][0]` (`3`):

   - Minimum of row max and column max is `min(8, 9) = 8`.
   - Maximum height increase: `8 - 3 = 5`.

   At `grid[0][1]` (`0`):

   - Minimum of row max and column max is `min(8, 5) = 5`.
   - Maximum height increase: `5 - 0 = 5`.

   And so on for the other buildings.

4. **Summing the Results:**

   We sum up the calculated increases for each building to get the total height increase without changing the skyline:

   `grid[0][0]`: 5 + `grid[0][1]`: 5 + `grid[0][2]`: 0 (since 8 - 8 = 0, no increase needed) + `grid[1][0]`: 2 + `grid[1][1]`: 0 (since 7 - 7 = 0, no increase needed) + `grid[1][2]`: 0 (since 7 - 7 = 0, no increase needed) + `grid[2][0]`: 0 (since 9 - 9 = 0, no increase needed) + `grid[2][1]`: 3 + `grid[2][2]`: 2

   = 5 + 5 + 0 + 2 + 0 + 0 + 0 + 3 + 2

   = 17

So according to the algorithm, the maximum total sum that building heights can be increased by, without affecting the city's skyline from any cardinal direction, is 17.

Python Solution

```python
class Solution:
    def maxIncreaseKeepingSkyline(self, grid: List[List[int]]) -> int:
        # Find the maximum heights in each row (the row skylines)
        max_height_in_row = [max(row) for row in grid]

        # Find the maximum heights in each column (the column skylines)
        # We do this by using zip to iterate over columns instead of rows
        max_height_in_column = [max(column) for column in zip(*grid)]

        # Calculate the sum of the possible height increase for each building
        # without exceeding the row and column skylines.
        total_increase = sum(
            min(max_height_in_row[i], max_height_in_column[j]) - grid[i][j]
            for i in range(len(grid))
            for j in range(len(grid[0]))
        )

        # Return the total possible increase sum.
        return total_increase
```

Java Solution

```java
class Solution {

    public int maxIncreaseKeepingSkyline(int[][] grid) {
        // Get the number of rows and columns of the grid.
        int numRows = grid.length;
        int numCols = grid[0].length;

        // Initialize arrays to store the max height of the skyline
        // for each row (maxRowHeights) and column (maxColHeights).
        int[] maxRowHeights = new int[numRows];
        int[] maxColHeights = new int[numCols];

        // Compute the max height for each row and column.
        for (int row = 0; row < numRows; ++row) {
            for (int col = 0; col < numCols; ++col) {
                maxRowHeights[row] = Math.max(maxRowHeights[row], grid[row][col]);
                maxColHeights[col] = Math.max(maxColHeights[col], grid[row][col]);
            }
        }

        // Initialize a variable to keep track of the total increase in height.
        int totalIncrease = 0;

        // Calculate the maximum possible increase for each building
        // while keeping the skyline unchanged.
        for (int row = 0; row < numRows; ++row) {
            for (int col = 0; col < numCols; ++col) {
                // The new height is the minimum of the max heights of the
                // current row and column.
                int newHeight = Math.min(maxRowHeights[row], maxColHeights[col]);
                // Increase by the difference between the new height and the original height.
                totalIncrease += newHeight - grid[row][col];
            }
        }

        // Return the total increase in the height of the buildings.
        return totalIncrease;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Required for std::max and std::min functions

class Solution {
public:
    int maxIncreaseKeepingSkyline(vector<vector<int>>& grid) {
        // Determine the number of rows and columns in the grid
        int rowCount = grid.size(), colCount = grid[0].size();

        // Create vectors to store the max values for each row and column
        vector<int> rowMax(rowCount, 0);
        vector<int> colMax(colCount, 0);

        // Iterate through the grid to find the max values for each row and column
        for (int i = 0; i < rowCount; ++i) {
            for (int j = 0; j < colCount; ++j) {
                // Update the maximum in the current row
                rowMax[i] = std::max(rowMax[i], grid[i][j]);
                // Update the maximum in the current column
                colMax[j] = std::max(colMax[j], grid[i][j]);
            }
        }

        // Initialize the answer variable to accumulate the total increase in height
        int totalIncrease = 0;

        // Iterate through the grid to compute the maximum possible increase
        // while maintaining the skylines
        for (int i = 0; i < rowCount; ++i) {
            for (int j = 0; j < colCount; ++j) {
                // The increase is the smaller of the two max values for the
                // current row and column minus the current grid height
                totalIncrease += std::min(rowMax[i], colMax[j]) - grid[i][j];
            }
        }

        // Return the total possible increase in height for the buildings
        return totalIncrease;
    }
};
```

Typescript Solution

```typescript
function maxIncreaseKeepingSkyline(grid: number[][]): number {
    // Create an array to store the maximum height in each row.
    let rowMaxes = grid.map(row => Math.max(...row));

    // Create an array to store the maximum height in each column.
    let colMaxes = new Array(grid[0].length).fill(0).map((_, colIndex) =>
        Math.max(...grid.map(row => row[colIndex]))
    );

    // Initialize a variable to keep track of the total increase in height.
    let totalIncrease = 0;

    // Calculate the maximum increase in height for each building while
    // keeping the skyline from changing.
    for (let rowIndex = 0; rowIndex < grid.length; rowIndex++) {
        for (let colIndex = 0; colIndex < grid[0].length; colIndex++) {
            // Find the minimum of the maximum heights of the current row and column.
            let limitHeight = Math.min(rowMaxes[rowIndex], colMaxes[colIndex]);

            // Increase the total height by the difference between the limit height and the current building's height.
            totalIncrease += limitHeight - grid[rowIndex][colIndex];
        }
    }

    // Return the total increase in height.
    return totalIncrease;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code can be broken down into two parts:

1. Calculating the maximum value for each row and column:
   - We iterate through each row with `max(row)` which takes `O(N)` time for each row, where `N` is the number of columns. This is done for each of the `M` rows, resulting in `O(M*N)` time.
   - Similarly, zipping (`zip(*grid)`) the columns takes `O(M*N)` because it iterates through all elements, and computing max for each column is `O(M)`, which is also done for `N` columns. This does not increase the total time complexity, so it remains `O(M*N)`.
2. Calculating the increment for each building (sum):
   - The double for loop iterates over all elements of the matrix, which is `M*N` operations. Within each operation, we perform a constant-time calculation (`min(rmx[i], cmx[j]) - grid[i][j]`). Hence, this part is also `O(M*N)`.

Since both steps are `O(M*N)` and they are performed sequentially, the total time complexity of the function is `O(M*N)`.

Space Complexity

The space complexity is determined by the additional space used besides the input grid:

1. The space to store maximums of rows (`rmx`) is `O(M)`.
2. The space for maximums of columns (`cmx`) is `O(N)`.

Here, M is the number of rows and N is the number of columns in the grid. Thus, the total additional space used is `O(M + N)`.

Therefore, the overall space complexity is `O(M + N)`.