2500. Delete Greatest Value in Each Row

Problem Description

In this problem, we are working with a matrix `grid` with dimensions `m x n` (where `m` is the number of rows and `n` is the number of columns), filled with positive integers. The primary objective is to continually perform a specific operation on the grid until it becomes empty. The operations can be detailed as follows:

- Firstly, we delete the elements with the greatest value from each row. This operation may remove more than one element per row if multiple elements share the greatest value.
- Secondly, we identify the maximum of the values deleted in this round and add it to the answer. This step essentially means that after each round of deletions, only the single highest value from among all the rows is added to the running total.

It is crucial to note that eliminating elements results in the grid having one less column after each operation. Consequently, with each iteration, the grid's dimensions change, shrinking in the column direction. The task is to return the cumulative total of the maximum values added after each operation has been completed and the grid is empty.

Intuition

To arrive at a solution for this problem, we should recognize that we are not required to modify the original grid structure. Instead, we need to repeatedly find and sum the largest elements removed based on the current iteration.

As we are required to sum the maximum elements of each row, and then take the maximum of these per operation, the key insight is that sorting each row will facilitate easy access to the maximum values. By sorting, we always have the largest value at the end of each row, and this enables quick removal and processing.

To sum the largest elements removed from each row, we utilize Python's `zip` function combined with an unpacking operator `*`, which effectively transposes the sorted grid. Transposing the grid after sorting allows us to use the intuitive `max` function on the columns, which now represent the original rows' maximum elements.

This leads to a concise and efficient process:

1. Sort each row of `grid` so that the element with the greatest value is at the end of each row (making it easy to identify).
2. Transpose the grid, turning rows into columns, effectively grouping the elements to be deleted together.
3. Sum the maximum elements across what are now the columns, which correspond to the greatest elements of the original rows.

The final sum is the answer we return. This solution avoids manipulating the matrix extensively. Instead, it leverages Python's built-in functions for sorting, transposing, and finding the max to perform the necessary computations efficiently.

Solution Approach

The implementation of the solution follows a straightforward approach that heavily utilizes built-in Python functions to handle the primary operations. Here is a step by step walk-through of the algorithm:

1. **Sorting Rows**: The initial step involves sorting each row in the grid. This is important because, according to the problem description, we need to delete the greatest value from each row. Sorting each row guarantees that the largest element is at the end of the row, which makes identifying these elements straightforward.

   Algorithmically, the `sort()` method in Python rearranges the elements in each list (which corresponds to a row in the grid) in ascending order in-place, meaning no additional data structures are necessary for this step.

   ```
   for row in grid:
       row.sort()
   ```

2. **Transposing the Matrix**: After sorting, the solution transposes the grid using the `zip(*grid)` function. In effect, this creates an iterator of tuples where each tuple contains elements from the rows of the grid that are in the same column position. Since the grid rows are sorted, the last elements (which are now at the first position in each tuple) are the largest elements from each row.

   Transposing is a pattern that allows us to treat each tuple as a representation of a column, making it easy to find the maximum element across these columns. By following this pattern, we avoid having to manually loop through each row to find and delete the maximum element, which can be computationally expensive.

3. **Summing the Maximum Elements**: The final step involves summing the maximum elements of the previously transposed 'columns' (which, in reality, are representing the maximum elements from each row). This is done using a generator expression within the `sum` function. The `max` function is called on each column, and the return values, which are the largest elements from each original row, are summed up to form the answer.

   ```
   return sum(max(col) for col in zip(*grid))
   ```

It's important to note that this approach doesn't change the original grid size since we don't need to keep track of the number of columns decreased after each operation. The `zip()` function naturally ends when the shortest row (or column after transposition) is exhausted, which aligns with the original grid shrinking in dimensions as the problem statement suggests.

The algorithm makes a single pass through each row to sort it and a single pass to sum the maxima, resulting in a time complexity that is efficient given the constraints of the problem. The spatial complexity is also minimal as no additional data structures are required; the sorting and transposition operations are performed in-place (except for the space utilized by zip, which is negligible as it returns an iterator).

By carefully using these built-in functions and patterns, we obtain an elegant solution that requires only a few lines of code but still adheres to the requirements of the problem and achieves optimal performance.

Example Walkthrough

Let's take a small example to illustrate the solution approach with a grid that looks as follows:

```
grid = [
  [3, 1, 2],
  [4, 5, 6],
  [9, 8, 7]
]
```

Step-by-step, we would perform the following actions:

1. **Sorting Rows**: We start by sorting each row of our `grid`. After sorting, the grid becomes:

   ```
   [
     [1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]
   ]
   ```

   This is achieved using the `sort()` function in Python on all the rows:

   ```
   for row in grid
       row.sort()
   ```

2. **Transposing the Matrix**: After sorting, we transpose the grid to work with the maximum elements more conveniently. The transposed grid (`zip(*grid)`) turns our rows into tuples that act like columns:

   ```
   [
     (1, 4, 7),
     (2, 5, 8),
     (3, 6, 9)
   ]
   ```

   Each tuple contains elements that were previously in the same column. For example, 1, 4, 7 were the first elements of their respective rows but are now grouped together after transposition.

3. **Summing the Maximum Elements**: We find the maximum value in each of these tuples (which are our transposed 'columns') and sum them together. These maximum values are 3, 6, and 9 from the 'columns':

   ```
   max(1, 4, 7) = 7
   max(2, 5, 8) = 8
   max(3, 6, 9) = 9
   ```

   The sum of these max values gives us the total: `7 + 8 + 9 = 24`. This is the final answer to our problem.

   In Python, this can be done with the following code:

   ```
   return sum(max(col) for col in zip(*grid))
   ```

And that's our expected result, a total of 24. This walkthrough demonstrates the usage of sorting rows, transposing the matrix, and summing the maximum elements, thus achieving the goal of efficiently computing the cumulative total after performing the operations dictated by the problem.

Python Solution

```python
class Solution:
    def deleteGreatestValue(self, grid: List[List[int]]) -> int:
        # Sort each row within the grid to have elements in ascending order
        for row in grid:
            row.sort()

        # Transpose the grid to iterate over columns instead of rows
        transposed_grid = zip(*grid)

        # Find the maximum element in each column and calculate their sum
        # This works because after sorting, the maximum elements are at the end of each row,
        # which after transposing, makes them the end of each column
        sum_of_max_values = sum(max(column) for column in transposed_grid)

        # Return the sum of the greatest values in each column
        return sum_of_max_values
```

Java Solution

```java
public class Solution {
    // Method to delete the greatest value in each column of a 2D grid and return
    // the sum of all the maximum values removed.
    public int deleteGreatestValue(int[][] grid) {
        // Step 1: Sort each row of the grid in ascending order
        for (int[] row : grid) {
            Arrays.sort(row);
        }

        int sumOfMaxValues = 0; // This will store the sum of the maximum values removed

        // Step 2: Iterate through each column to find the maximum value in that column
        for (int colIndex = 0; colIndex < grid[0].length; colIndex++) {
            int columnMax = 0; // Store the maximum value of the current column

            // Step 3: Iterate through each row for the current column
            for (int rowIndex = 0; rowIndex < grid.length; rowIndex++) {
                // Update columnMax if the current element is greater than columnMax
                columnMax = Math.max(columnMax, grid[rowIndex][colIndex]);
            }

            // Step 4: Add the largest value of the current column to the sumOfMaxValues
            sumOfMaxValues += columnMax;
        }

        // Step 5: Return the sum of all the maximum values removed from the grid
        return sumOfMaxValues;
    }
}
```

C++ Solution

```cpp
#include <vector> // Include necessary header for the vector
#include <algorithm> // Include for the std::sort and std::max functions

class Solution {
public:
    // Function to delete the greatest value from each column of a grid
    // and return the sum of those greatest values.
    int deleteGreatestValue(vector<vector<int>>& grid) {
        // Sort each row of the grid in non-decreasing order.
        for (auto& row : grid) {
            sort(row.begin(), row.end());
        }

        int sumOfMaxValues = 0; // Initialize the sum of max values to 0

        // Iterate through each column of the grid.
        for (int j = 0; j < grid[0].size(); ++j) {
            int maxInColumn = 0; // Variable to hold the max value in current column

            // Iterate through each row of the grid to find the greatest value in the current column.
            for (int i = 0; i < grid.size(); ++i) {
                // Update maximum value in the column.
                maxInColumn = max(maxInColumn, grid[i][j]);
            }

            // Add the greatest value from the current column to the sum.
            sumOfMaxValues += maxInColumn;
        }

        // Return the sum of the greatest values from all columns.
        return sumOfMaxValues;
    }
};
```

Typescript Solution

```typescript
function deleteGreatestValue(grid: number[][]): number {
  // Sort each row of the grid in non-decreasing order
  for (const row of grid) {
    row.sort((a, b) => a - b);
  }

  // Initialize the answer variable to accumulate the greatest values
  let answer = 0;

  // Iterate over each column of the grid
  for (let columnIndex = 0; columnIndex < grid[0].length; ++columnIndex) {
    // Initialize a variable to keep track of the greatest value in the current column
    let greatestValueInColumn = 0;

    // Iterate over each row to find the greatest value in the current column
    for (let rowIndex = 0; rowIndex < grid.length; ++rowIndex) {
      greatestValueInColumn = Math.max(greatestValueInColumn, grid[rowIndex][columnIndex]);
    }

    // Accumulate the greatest values of each column
    answer += greatestValueInColumn;
  }

  // Return the total sum of greatest values from all columns
  return answer;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code consists of two parts:

1. Sorting each row in the `grid` list.
2. Finding the maximum value of each column after the rows have been sorted.

The grid size is defined by `m x n`, where `m` is the number of rows and `n` is the number of columns.

For the first part, we sort each row individually. The sort operation on a row of length `n` has a time complexity of $O(n \log n)$. Since there are `m` such rows, this part of the process has a total time complexity of $O(m * n \log n)$.

For the second part, after sorting, we find the maximum element in each column using a generator expression that zips the sorted rows together. This operation is $O(m)$ for each column since it involves iterating through every row for the given column to find the max. There are `n` columns, so the total time complexity for this part is $O(n * m)$.

Therefore, the total time complexity of the entire code is $O(m * n \log n + n * m)$. Since $m * n \log n$ would likely dominate for larger values of `n`, we might consider this the more significant term, and the overall time complexity can be approximated by $O(m * n \log n)$.

Space Complexity

For space complexity, we need to consider the additional space used by the algorithm excluding the input itself:

1. Sorting is done in-place in the provided `grid` list, not requiring additional space that depends on the size of the input. However, Python's `sort` method under the hood may use $O(n)$ space for each row due to the Timsort algorithm it implements.
2. The generator expression used to calculate the sum of maximum values in each column does not require significant additional space; it will take $O(1)$ space as it evaluates each value on-the-fly and does not store the entire column.

Thus, the additional space is $O(n)$, needed for the in-place sorting of one row at a time, which is the most significant additional space used by the algorithm.

Considering the constraints above, the overall space complexity of the algorithm is $O(n)$.