598. Range Addition II

Easy Array Math

Leetcode Link

Problem Description

In this problem, we have a matrix of size `m x n`, which is initially filled with zeros. We are also given a list of operations, where each operation is represented by a two-element array. The first element, `a_i`, signifies the number of rows (from the top), and the second element, `b_i`, signifies the number of columns (from the left), which will be affected by the operation. An operation increments by one each element in the submatrix defined by the given number of rows and columns. After applying all the operations, some elements in the matrix might be incremented multiple times.

Our task is to count the number of times the maximum value in the matrix occurs after carrying out all the given operations.

Intuition

To find the solution to this problem effectively, we notice that any overlap between operations will result in those specific cells being incremented multiple times. Therefore, the cells that are incremented the most are the ones that fall within the area of overlap of all the operations.

Instead of executing all operations which can lead to a time-consuming process for large matrices, we can find the intersection of all operation ranges. This intersection will give us the smallest submatrix that each operation influences.

We can find the minimum `a_i` and `b_i` across all operations since the operations are inclusive from index 0 up to but not including `a_i` for rows and `b_i` for columns. The intersection of these minimum values gives us the range of rows and columns that every operation will affect, and hence, these cells will have the maximum value.

Finally, we calculate the size of the range by multiplying the minimum `a_i` by the minimum `b_i`, and this gives us the count of maximum integers in the matrix after performing all operations. This is the key insight that allows us to arrive at an efficient solution.

Solution Approach

The solution approach for this problem is straightforward once the intuition is understood. We use a simple greedy algorithm to find the minimum values of `a_i` and `b_i`. A greedy algorithm is an approach for solving problems by choosing the best option available at the moment, without considering the bigger picture. In this case, we are selecting the minimum `a_i` and `b_i` because we are certain that the best option to find the maximum overlapped area is to find the smallest range that is affected by all operations.

Here are the steps in the algorithm using Python code as an example:

1. Initialize two variables `min_row` and `min_col` with the maximum possible values of `m` and `n` respectively, which represent the dimensions of the matrix.

2. Iterate over each operation (`ops`) in the list of operations:

   - For each operation, take the pair `[a_i, b_i]`.
   - Update `min_row` to be the minimum of the current `min_row` and `a_i`.
   - Update `min_col` to be the minimum of the current `min_col` and `b_i`. By doing this, we find the intersection of all the operations which is the smallest submatrix affected by all operations.

3. After finding the minimum values for the rows (`min_row`) and columns (`min_col`), the maximum number of times an element has been incremented is exactly the region defined by `min_row` x `min_col`.

We do not use any complex data structures for this problem as the Python solution demonstrates efficiency through avoiding explicit matrix construction and operation simulation, which could be costly in terms of both time and space for large matrices. The problem is reduced to simple value comparisons and multiplication which makes it an O(k) solution, where k is the number of operations, irrespective of the actual dimensions of the matrix.

The Python solution uses this approach to provide a concise and efficient solution. Here's a brief walk-through of the Python code:

```python
class Solution:
    def maxCount(self, m: int, n: int, ops: List[List[int]]) -> int:
        # Initialize `m` and `n` to represent the max possible numbers of rows and columns.
        for a, b in ops:  # Loop through each operation.
            m = min(m, a)  # Find the smallest value of `a_i` from all operations.
            n = min(n, b)  # Find the smallest value of `b_i` from all operations.
        return m * n  # The size of the overlapped area is the result we want.
```

The `min` functions are used to find the smallest values from the provided `a` and `b` because every `a` affects rows `0` to `a-1` and every `b` affects columns `0` to `b-1`. By multiplying `m` and `n`, which are now the sizes of the maximum overlapped area, we get the count of elements in the matrix with the maximum values after all operations.

Example Walkthrough

Let's consider a small example to illustrate the solution approach.

Suppose we have a matrix of size `4 x 5` (`m = 4`, `n = 5`), and we are given a list of three operations: `ops = [[2,3], [3,3], [3,2]]`. To visualize the initial state of the matrix filled with zeros, it would look something like this:

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

Now, let's perform these operations according to the described algorithm without actually incrementing each cell:

1. The first operation is `[2, 3]`, which means we would increment the submatrix from rows `0 to 1` and columns `0 to 2`. The affected area, for now, is `2 x 3`.

2. The second operation is `[3, 3]`, which would cover rows `0 to 2` and columns `0 to 2`, but since we are looking for the overlap, our affected area is still `2 x 3` (as the first operation only covers two rows).

3. The third operation is `[3, 2]`, which has a coverage from rows `0 to 2` and columns `0 to 1`. When looking for the overlapping area, we observe that the rows overlap with the first and second operations, but the columns now only overlap within the first two columns. Therefore, the final smallest submatrix that all operations influence is now `2 x 2`.

Given that we want the count of maximum values after all these operations, we don't need to perform the increment. We simply need the size of the range defined by the minimum `a_i` and `b_i` for rows and columns respectively.

For our example, the smallest `a_i` from the operations is `2` (from the first operation), and the smallest `b_i` is `2` as well (from the third operation).

Now we multiply these values: `min_row = 2` and `min_col = 2`, therefore, the count is `min_row * min_col = 2 * 2 = 4`.

So, after all the operations, there are four elements in the matrix that have the maximum value, and that's the result. No element will be incremented more times than these four elements that lie in the intersection of all operation ranges.

Python Solution

```python
class Solution:
    def maxCount(self, rows: int, cols: int, operations: List[List[int]]) -> int:
        # Iterate through each operation in the list of operations
        for operation in operations:
            # The operation contains two elements which are the new limits for rows (a)
            # and columns (b) for the increment operations on the matrix
            rows = min(rows, operation[0])  # Update rows to the minimum of the current rows and operation rows limit
            cols = min(cols, operation[1])  # Update columns to the minimum of the current columns and operation columns limit

        # Return the area of the top-left sub-matrix which would have the maximum count
        # This is because all increment operations would have affected this sub-matrix
        return rows * cols

```

Java Solution

```java
class Solution {
    /**
     * Finds the maximum count of the most frequent integer after performing operations on a matrix.
     *
     * @param rows The number of rows of the matrix.
     * @param columns The number of columns of the matrix.
     * @param operations An array of operations where each operation consists of 2 integers a, b,
     *                   meaning that all the elements of the submatrix top-left (0,0) to bottom-right (a-1,b-1)
     *                   will be incremented by 1.
     * @return The maximum count of the most frequent integer in the matrix after performing the given operations.
     */
    public int maxCount(int rows, int columns, int[][] operations) {
        // Iterate over each operation
        for (int[] operation : operations) {
            // Find the minimum row index affected by the current operation
            rows = Math.min(rows, operation[0]);
            // Find the minimum column index affected by the current operation
            columns = Math.min(columns, operation[1]);
        }
        // Return the area of the top-left submatrix that receives the maximum increments
        return rows * columns;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the maximum count of integers in the matrix that have the max value after performing operations
    int maxCount(int m, int n, vector<vector<int>>& ops) {
        // Iterate through each operation
        for (auto op : ops) {
            // Find and assign the minimum of "m" or the first value of the operation pair
            m = min(m, op[0]);
            // Find and assign the minimum of "n" or the second value of the operation pair
            n = min(n, op[1]);
        }
        // Since the operations are incrementing from the top-left corner, the minimum values of m and n
        // will give the dimensions of the rectangle that got incremented the most.
        // The product of m and n will give the total count of integers with maximum value.
        return m * n;
    }
};

```

Typescript Solution

```typescript
// Function to find the maximum count of integers in the matrix that have the max value after performing operations
function maxCount(m: number, n: number, ops: number[][]): number {
    // Iterate through each operation
    for (let op of ops) {
        // Find and assign the minimum of "m" or the first value of the operation pair
        m = Math.min(m, op[0]);
        // Find and assign the minimum of "n" or the second value of the operation pair
        n = Math.min(n, op[1]);
    }
    // Since the operations are incrementing starting from the top-left corner, the minimum values of m and n
    // will give the dimensions of the subset of the matrix that is incremented the most.
    // Multiplying m and n yields the total count of integers with the maximum value in the matrix.
    return m * n;
}

```

Time and Space Complexity

Time Complexity

The time complexity of this function is `O(k)`, where `k` is the number of operations in the `ops` list. The reason for this is that the function loops through each operation once and performs a constant-time operation of updating `m` and `n` to the minimum of their current values and the values in the operation. Since looping through the list of operations is the only step that depends on the size of the input, and each iteration involves constant time work, the overall time complexity is linear with respect to the number of operations in `ops`.

Space Complexity

The space complexity of this function is `O(1)` because it only uses a fixed amount of extra space. No additional space is allocated that is dependent on the input size of the problem; the variables `m` and `n` are updated in place. As the amount of space used does not change with varying input sizes (as `ops` are merely iterated over and not stored), the space complexity is constant.