2033. Minimum Operations to Make a Uni-Value Grid

Medium Array Math Matrix Sorting Leetcode Link

Problem Description

The problem presents us with a two-dimensional grid of integers and a single integer `x`. The operation that can be executed involves adding or subtracting `x` to any element in the grid. A `uni-value grid` is defined as a grid where all elements are equal. The goal is to determine the minimum number of such operations required to turn the input grid into a uni-value grid. If this objective is unattainable, the function should return `-1`.

The critical constraint to keep in mind is that all elements in the final uni-value grid must be equal, which implies that the difference between any two given elements in the original grid must be a multiple of `x`. Otherwise, it would be impossible to reach a common value through the given operation.

Intuition

To arrive at the minimum number of operations needed to achieve a uni-value grid, we need to find a target value that all elements in the grid will be equal to after the operations. Since adding or subtracting `x` continues to maintain the same remainder when any element is divided by `x`, all elements must share the same remainder `mod` when divided by `x`; otherwise, there's no legal operation that will create a uni-value grid.

Given that all elements can be modified to share the same remainder when divided by `x`, the next step is to decide on which value to set all elements to. A natural choice is the median of all values, as it minimizes the absolute deviation sum for a set of numbers. In other words, using the median value as a target implies the least total difference between each number and the median, hence requiring the fewest operations.

The solution approach follows these steps:

1. Flatten the grid into a list `nums`, to work with a single-dimensional array of all the values.
2. Check if all elements have the same remainder when divided by `x`. If not, return `-1`.
3. Sort `nums` and find the median value.
4. Sum the absolute division (//) of differences between each element in `nums` and the median by `x`. This represents the minimum number of operations to adjust each element to make the entire grid uni-value.

This solution is efficient as it minimally processes the elements by using their inherent properties (specifically, the remainder upon division) and statistical measures (median) to determine feasibility and achieve optimality for the problem.

Solution Approach

The solution approach for converting the grid to a uni-value grid with a minimum number of operations follows a simple but powerful statistical concept. Here is the breakdown of the implementation steps, referencing the provided Python code:

1. **Flatten the Grid**: The first step involves transforming the two-dimensional grid into a one-dimensional array (list). This is achieved by defining an empty list `nums` and iterating over each row and within that, each value `v`, to append it to `nums`.

2. **Check Modulo Condition**: Before proceeding, we need to guarantee that all elements can be adjusted to the same value using the given operation. For this purpose, we store the remainder of the first grid element (e.g., `grid[0][0] % x`) in `mod` and then check if `v % x == mod` for all elements in the grid. If any element doesn't satisfy this, it's impossible to reach a uni-value grid, and the function immediately returns `-1`.

3. **Sort and Find Median**: Once we have a flat list `nums` of all values, we sort it to arrange the numbers in ascending order. Finding the median is straightforward from this sorted list; it's the middle value, which minimizes the total number of operations needed. If the list's length is even, either of the two middle values will work due to the way the median minimizes the sum of absolute deviations. The median is found using `nums[len(nums) >> 1]`, which is an efficient way to calculate the median index (the same as `len(nums) // 2`).

4. **Calculate Total Operations**: Finally, we calculate the total number of operations needed by summing the integer division of absolute differences between each element and the median, divided by `x` (i.e., `sum(abs(v - mid) // x for v in nums)`). This represents how many times we'd need to add or subtract `x` to/from each element to convert it into the median value.

5. **Return Result**: The result of the sum is the minimum number of operations required to make the grid uni-value, which is what the function returns.

In terms of algorithms and patterns, the problem mainly relies on the properties of numbers (divisibility and modulo operation), and the use of sorting and median as a means to minimize absolute differences. No complex data structures are used beyond the one-dimensional list for sorting and iteration, and the algorithm overall exhibits a time complexity effectively determined by the sorting step, which is typically O(N log N) where N is the number of elements in the grid.

Example Walkthrough

Let's consider a two-dimensional grid and an integer `x` with the following values:

```
Grid: [[2, 4], [6, 8]]
x: 2
```

We want to convert the given grid into a uni-value grid using the minimum number of operations where we can add or subtract `x` to any element.

Following the solution approach:

1. **Flatten the Grid**: We transform our two-dimensional grid into a one-dimensional list `nums`. After flattening, `nums` will be `[2, 4, 6, 8]`.

2. **Check Modulo Condition**: We check if all elements have the same remainder when divided by `x`. For our grid, all elements (`2 % 2`, `4 % 2`, `6 % 2`, `8 % 2`) have a remainder of `0` when divided by `2`. Since they all have the same remainder, we can proceed.

3. **Sort and Find Median**: We sort the list `nums` to get `[2, 4, 6, 8]`. The length of the list is 4 (even), so the median could be either of the two middle values, 4 or 6. Here we'll choose 4 as the median (`nums[4 // 2]`).

4. **Calculate Total Operations**: We calculate the number of operations needed to transform each element into the median. For every element `v` in `nums`, we calculate `abs(v - median) // x` and sum these values up:

   - For `2`: `abs(2 - 4) // 2` equals `1` operation.
   - For `4`: `abs(4 - 4) // 2` equals `0` operations.
   - For `6`: `abs(6 - 4) // 2` equals `1` operation.
   - For `8`: `abs(8 - 4) // 2` equals `2` operations.

   Summing these up gives us `1 + 0 + 1 + 2 = 4` operations.

5. **Return Result**: The minimum number of operations required to make the grid uni-value is `4`. This is the final result we return.

Through these steps, we've used a statistical approach (median) alongside modulo operations to efficiently determine the required operations to reach a uni-value grid, ensuring that the chosen operations are valid and minimal.

Python Solution

```python
class Solution:
    def minOperations(self, grid, x):
        """
        Determine the minimum number of operations required to make all elements in the grid equal,
        where in one operation you can add or subtract `x` to any element in the grid.

        :param grid: List[List[int]]
        :param x: int
        :return: int, the minimum number of operations or -1 if it's not possible
        """

        # Flatten the grid into a single sorted list to make it easier to handle
        flattened_grid = []

        # Start by storing the remainder of the first element (for modulo comparison)
        required_modulo = grid[0][0] % x

        # Iterate through the grid
        for row in grid:
            for value in row:

                # If the current value cannot be achieved by any number of operations,
                # we return -1 because the whole grid cannot be equalized
                if value % x != required_modulo:
                    return -1

                # Otherwise, store the value in our flattened grid
                flattened_grid.append(value)

        # Sort all values in our list to easily find the median
        flattened_grid.sort()

        # Find the median, which will be our target value
        median = flattened_grid[len(flattened_grid) // 2]

        # Calculate the sum of operations required to make all values equal to the median
        operations = sum(abs(value - median) // x for value in flattened_grid)

        # Return the total number of operations
        return operations

# Example usage:
# solution = Solution()
# grid = [[1, 5], [2, 3]]
# x = 1
# print(solution.minOperations(grid, x)) # Outputs the minimum number of operations required

```

Java Solution

```java
class Solution {
    public int minOperations(int[][] grid, int x) {
        // Get the number of rows
        int rows = grid.length;
        // Get the number of columns
        int cols = grid[0].length;
        // Initialize an array to store all elements in the grid
        int[] flattenedGrid = new int[rows * cols];
        // Modulo of the first element in the grid. All other elements should have the same modulo if a solution is possible
        int initialMod = grid[0][0] % x;

        // Flatten the 2D grid into a 1D array while checking if operation is not possible
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // If any element has a different modulo than the first element, return -1 as it's not possible to make them all equa
                if (grid[i][j] % x != initialMod) {
                    return -1;
                }
                // Store elements of the grid in the flattened 1D array
                flattenedGrid[i * cols + j] = grid[i][j];
            }
        }

        // Sort the flattened 1D array
        Arrays.sort(flattenedGrid);
        // Find the median element
        int median = flattenedGrid[flattenedGrid.length / 2];
        // Initialize operation count to 0
        int operationCount = 0;

        // Calculate the total number of operations required to make all elements equal to the median
        for (int value : flattenedGrid) {
            // Increment the operation count by the number of operations needed for each element.
            // The distance between the element and the median is divided by x
            // as that's the number of operations required to make them equal
            operationCount += Math.abs(value - median) / x;
        }

        // Return the total operation count
        return operationCount;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int minOperations(vector<vector<int>>& grid, int x) {
        // Calculate the dimensions of the grid
        int rows = grid.size();
        int cols = grid[0].size();

        // Determine the modulo of the first element as a reference
        int referenceModulo = grid[0][0] % x;

        // Flatten the grid into a single vector for ease of manipulation
        vector<int> flattenedGrid(rows * cols);

        // Process the grid while checking if it's possible to make all elements equal
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // If an element's modulo isn't equal to the reference, it's not possible to make all elements equal
                if (grid[i][j] % x != referenceModulo) {
                    return -1;
                }
                // Populate the flattened grid
                flattenedGrid[i * cols + j] = grid[i][j];
            }
        }

        // Sort the flattened grid to find the median
        sort(flattenedGrid.begin(), flattenedGrid.end());

        // The median value is the target value for all elements to make them equal
        int targetValue = flattenedGrid[(rows * cols) / 2];

        // Calculate the total number of operations needed
        int totalOps = 0;
        for (int value : flattenedGrid) {
            // Add the required number of operations per element
            totalOps += abs(value - targetValue) / x;
        }

        // Return the total number of operations
        return totalOps;
    }
};

```

Typescript Solution

```typescript
// Function to find the minimum number of operations needed to make all elements of a grid equal.
function minOperations(grid: number[][], x: number): number {
    // Calculate the dimensions of the grid
    let rows = grid.length;
    let cols = grid[0].length;

    // Determine the modulo of the first element as a reference
    let referenceModulo = grid[0][0] % x;

    // Flatten the grid into a single array for ease of manipulation
    let flattenedGrid: number[] = [];

    // Process the grid while checking if it's possible to make all elements equal
    for (let i = 0; i < rows; i++) {
        for (let j = 0; j < cols; j++) {
            // If an element's modulo isn't equal to the reference, it's not possible to make all elements equal
            if (grid[i][j] % x !== referenceModulo) {
                return -1;
            }
            // Populate the flattened grid
            flattenedGrid.push(grid[i][j]);
        }
    }

    // Sort the flattened grid to find the median
    flattenedGrid.sort((a, b) => a - b);

    // The median value is the target value for all elements to make them equal
    let targetValue = flattenedGrid[Math.floor((rows * cols) / 2)];

    // Calculate the total number of operations needed
    let totalOps = 0;
    for (let value of flattenedGrid) {
        // Add the required number of operations per element
        totalOps += Math.abs(value - targetValue) / x;
    }

    // Return the total number of operations
    return totalOps;
}

```

Time and Space Complexity

The given Python code is used to determine the minimum number of operations to make all elements in a grid equal if one can only add or subtract a value `x` from elements in the grid.

Time Complexity:

The time complexity of the code is analyzed as follows:

1. Creating the `nums` list with a nested loop through the grid: This takes `O(m*n)`, where `m` and `n` are the dimensions of the grid.
2. Sorting the `nums` list: The sort operation has a time complexity of `O(k*log(k))`, where `k` is the total number of elements in the grid (`k = m*n`).
3. Calculating the median and the number of operations: This involves a single pass through the sorted `nums` list, which takes `O(k)`.

Combining these, the dominant term is the sorting step, therefore the overall time complexity is `O(k*log(k))` which simplifies to `O(m*n*log(m*n))` since `k = m*n`.

Space Complexity:

1. Storing the `nums` list: Space complexity is `O(k)`, which is `O(m*n)` where `k` is the total number of elements in the grid.
2. Variables `mod` and `mid`: These are constant space and do not scale with the input, so they contribute `O(1)`.

Thus, the overall space complexity is `O(m*n)` since the storage for the `nums` list dominates the space usage.