1975. Maximum Matrix Sum

Medium Greedy Array Matrix Leetcode Link

Problem Description

Given an `n x n` matrix filled with integers, the task is to perform strategic operations to maximize the sum of all elements in the matrix. An operation consists of selecting any two adjacent elements (elements that share a border) and multiplying both by `-1`. This toggling between positive and negative can be done as many times as desired. The goal is to determine the highest possible sum of all elements in the matrix after performing zero or more of these operations.

Intuition

The key to solving this problem is realizing that each operation can be used to negate the effect of negative numbers in the matrix. It's generally advantageous to have all numbers positive for the highest sum. However, since operations are limited to pairs of adjacent elements, it may not always be possible to convert all negatives to positives.

The strategy is as follows:

1. Calculate the total sum of all the absolute values of the elements in the matrix. This is the maximum sum the matrix can possibly have if we could individually toggle each element to be positive.
2. Track the smallest absolute value of the elements in the matrix. This value is important because if there's an odd number of negative elements that can't be paired, the smallest element's negativity will have the smallest negative impact on the total sum.
3. Count the number of negative elements in the matrix. If this count is even, it means every negative element can be paired with another negative to turn both into positives through one operation.
4. If the count of negative elements is odd, there will remain one unpaired negative element affecting the total sum. In this case, subtract twice the smallest absolute value found in the matrix from the total sum to account for the impact of the remaining negative number after all possible pairs have been negated. If the smallest value is zero, it means an operation can be done without impacting the total sum, as multiplying zero by `-1` does not change the value.

The solution leverages these intuitions to determine the maximum sum.

Solution Approach

To implement the solution, we perform a series of steps that follow the intuition described earlier. Here's a breakdown of the algorithm based on the code provided:

1. Initialize a sum variable `s` to 0, which will store the total sum of absolute values of the matrix elements. Also, create a counter `cnt` to keep track of the number of negative elements and a variable `mi` to keep the minimum absolute value observed in the matrix.
2. Iterate through each element of the matrix using a nested loop (iterating first through rows, then through elements within each row). For every element `v` in the matrix:
   - Add the absolute value of the element to `s` (`s += abs(v)`), progressively building the total sum of absolute values.
   - Update the minimum absolute value `mi` if the absolute value of the current element is less than the current `mi` (`mi = min(mi, abs(v))`).
   - If the current element `v` is negative, increment the counter `cnt` (`if v < 0: cnt += 1`). This helps keep track of how many negative numbers are in the matrix, crucial for deciding the following steps.
3. After iterating through the matrix:
   - Check if the number of negative elements `cnt` is even or the smallest value `mi` is zero:
     - If `cnt` is even, it's possible to negate all negative elements by using the operation, and the highest sum can be obtained by simply summing all absolute values. Return the sum `s`.
     - If `mi` is zero, it means one of the elements is zero, and no subtraction is needed as multiplying zero by `-1` multiple times will not change the sum. So, return `s`.
   - If the number of negative elements `cnt` is odd and `mi` is not zero, it means that not all negative elements can be negated, and there will be one negative element affecting the total sum. Return `s - mi * 2`, subtracting twice the smallest absolute value to account for the remaining negative element's impact on the total sum.

This approach uses simple data structures (just integers and loops over the 2D matrix), with the main pattern being the calculation of sums, tracking the smallest value, and counting occurrences of a particular condition (negativity in this case), which are fairly common operations in matrix manipulation problems.

Example Walkthrough

Let's consider a small `3x3` matrix as an example:

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

Following the solution approach:

1. Initialize `s` as 0, `cnt` as 0, and `mi` as infinity (or a very large number).
2. Iterate through the matrix and calculate:
   - `s` (the sum of absolute values), which is `| -1 | + | -2 | + | 3 | + | 4 | + | 5 | + | -6 | + | -7 | + | 8 | + | 9 |` = `1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9` = `45`.
   - `mi` (the smallest absolute value), which is `min(infinity, 1, 2, 3, 4, 5, 6, 7, 8, 9)` = `1`.
   - `cnt` (the count of negative numbers), there are `4` negatives: `-1, -2, -6, -7` so `cnt = 4`.
3. After the iteration, check the conditions:
   - `cnt` is `4`, which is even. Therefore, we don't need to subtract anything from `s`, because we can pair all negatives and multiply by `-1` to turn them positive.
   - Since `cnt` is even, we can return `s` which equals `45` as our answer.

According to the strategy, the operations performed would be:

- Multiplying `-1` and `-2` by `-1` to turn them positive.
- Multiplying `-6` and `-7` by `-1` to turn them positive.

Therefore, the highest possible sum after applying the operations is `45`.

Python Solution

```python
class Solution:
    def maxMatrixSum(self, matrix: List[List[int]]) -> int:
        # Initialize total sum, negative count, and minimum positive value
        total_sum = negative_count = 0
        minimum_positive_value = float('inf')  # Represents infinity

        # Iterate over each row in the matrix
        for row in matrix:
            # Iterate over each value in the row
            for value in row:
                # Add the absolute value of the current element to the total sum
                total_sum += abs(value)

                # Record the minimum positive value (smallest absolute value)
                minimum_positive_value = min(minimum_positive_value, abs(value))

                # If the current value is negative, increment the negative count
                if value < 0:
                    negative_count += 1

        # If there is an even number of negatives (or a zero element, which can flip sign without penalty),
        # the result is simply the total sum of absolute values
        if negative_count % 2 == 0 or minimum_positive_value == 0:
            return total_sum

        # If there's an odd number of negatives, subtract twice the minimum positive
        # value to account for the one value that will remain negative
        return total_sum - minimum_positive_value * 2

```

Java Solution

```java
class Solution {
    public long maxMatrixSum(int[][] matrix) {
        long sum = 0; // Initialize a sum variable to hold the total sum of matrix elements
        int negativeCount = 0; // Counter for the number of negative elements in the matrix
        int minAbsValue = Integer.MAX_VALUE; // Initialize to the maximum possible value to track the smallest absolute value seen

        // Loop through each row of the matrix
        for (int[] row : matrix) {
            // Loop through each value in the row
            for (int value : row) {
                sum += Math.abs(value); // Add the absolute value of the element to the sum
                // Find the smallest absolute value in the matrix
                minAbsValue = Math.min(minAbsValue, Math.abs(value));
                // If the element is negative, increment the negativeCount
                if (value < 0) {
                    negativeCount++;
                }
            }
        }

        // If the count of negative numbers is even or there's at least one zero, return the sum of absolute values
        if (negativeCount % 2 == 0 || minAbsValue == 0) {
            return sum;
        }

        // Since the negative count is odd, we subtract twice the smallest absolute value to maximize the matrix sum
        return sum - (minAbsValue * 2);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    long long maxMatrixSum(vector<vector<int>>& matrix) {
        long long sumOfAbsoluteValues = 0; // This variable will store summation of absolute values of all elements in matrix
        int negativeCount = 0;             // Counter for the number of negative elements in the matrix
        int minAbsValue = INT_MAX;         // This variable will keep track of the smallest absolute value encountered

        // Loop over each row in the matrix
        for (auto& row : matrix) {
            // Loop over each element in the row
            for (int& value : row) {
                // Add the absolute value of the current element to the total sum
                sumOfAbsoluteValues += abs(value);
                // Update the smallest absolute value encountered if current absolute value is smaller
                minAbsValue = min(minAbsValue, abs(value));
                // If the current element is negative, increment the negative counter
                negativeCount += value < 0;
            }
        }

        // If the number of negative values is even or the minimum absolute value is 0,
        // we can make all elements non-negative without decreasing the sum of absolute values.
        if (negativeCount % 2 == 0 || minAbsValue == 0) {
            return sumOfAbsoluteValues;
        }

        // If the number of negative values is odd, we subtract twice the smallest
        // absolute value to compensate for the one element that will remain negative.
        return sumOfAbsoluteValues - minAbsValue * 2;
    }
};

```

Typescript Solution

```typescript
/**
 * Calculates the maximum absolute sum of any submatrix of the given matrix.
 *
 * @param {number[][]} matrix - The 2D array of numbers representing the matrix.
 * @return {number} - The maximum absolute sum possible by potentially negating any submatrix element.
 */
function maxMatrixSum(matrix: number[][]): number {
    let negativeCount = 0; // Count of negative numbers in the matrix
    let sum = 0; // Sum of the absolute values of the elements in the matrix
    let minAbsValue = Infinity; // Smallest absolute value found in the matrix

    // Iterate through each row of the matrix
    for (const row of matrix) {
        // Iterate through each value in the row
        for (const value of row) {
            sum += Math.abs(value); // Add the absolute value to the sum
            minAbsValue = Math.min(minAbsValue, Math.abs(value)); // Update min absolute value if necessary
            negativeCount += value < 0 ? 1 : 0; // Increment count if the value is negative
        }
    }

    // If the count of negative numbers is even, the sum is already maximized
    if (negativeCount % 2 == 0) {
        return sum;
    }

    // Otherwise, subtract double the smallest absolute value to negate an odd count of negatives
    return sum - minAbsValue * 2;
}

// Example usage:
// const matrix = [[1, -1], [-1, 1]];
// const result = maxMatrixSum(matrix);
// console.log(result); // Output should be 4

```

Time and Space Complexity

Time Complexity

The given code iterates through all elements of the `matrix` with dimension `n x n`. For each element, it performs a constant number of operations: calculating the absolute value, updating the sum `s`, comparing with the minimum value `mi`, and incrementing a counter `cnt` if the value is negative.

- Iterating through all elements takes `O(n^2)` time, where `n` is the dimension of the square matrix (since there are `n` rows and `n` columns).
- All the operations inside the nested loops are constant time operations.

Hence, combining these, the overall time complexity of the code is `O(n^2)`.

Space Complexity

The space complexity is determined by the additional space required by the code, not including the space taken by the inputs.

- The variables `s`, `cnt`, and `mi` use constant space.
- There are no additional data structures that grow with the size of the input.

Therefore, the space complexity of the code is `O(1)`, which indicates constant space usage regardless of the input size.