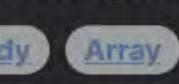
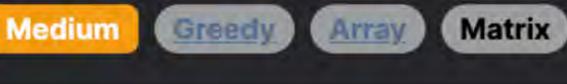




Problem Description





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

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:

The key to solving this problem is realizing that each operation can be used to negate the effect of negative numbers in the matrix.

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:

abs(v))).

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,
 - 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.
 - 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

3. After iterating through the matrix:

- 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. o 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:

Following the solution approach:

2. Iterate through the matrix and calculate:

1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45.

```
1. Initialize s as 0, cnt as 0, and mi as infinity (or a very large number).
```

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

• mi (the smallest absolute value), which is min(infinity, 1, 2, 3, 4, 5, 6, 7, 8, 9) = 1. \circ cnt (the count of negative numbers), there are 4 negatives: -1, -2, -6, -7 so cnt = 4.

by -1 to turn them positive.

3. After the iteration, check the conditions: o cnt is 4, which is even. Therefore, we don't need to subtract anything from s, because we can pair all negatives and multiply

os (the sum of absolute values), which is | −1 | + | −2 | + | 3 | + | 4 | + | 5 | + | −6 | + | −7 | + | 8 | + | 9 | =

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

Since cnt is even, we can return s which equals 45 as our answer.

Python Solution

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

According to the strategy, the operations performed would be:

class Solution: def maxMatrixSum(self, matrix: List[List[int]]) -> int: # Initialize total sum, negative count, and minimum positive value total_sum = negative_count = 0

Iterate over each row in the matrix

total_sum += abs(value)

negative_count += 1

if (negativeCount % 2 == 0 || minAbsValue == 0) {

* Calculates the maximum absolute sum of any submatrix of the given matrix.

for row in matrix: # Iterate over each value in the row for value in row: 10 11 # Add the absolute value of the current element to the total sum

minimum_positive_value = float('inf') # Represents infinity

Record the minimum positive value (smallest absolute value) 14 15 minimum_positive_value = min(minimum_positive_value, abs(value)) 16 17 # If the current value is negative, increment the negative count 18 if value < 0:

6

12

13

19

17

18

19

20

21

22

23

24

25

26

27

28

30

29 }

```
20
21
           # If there is an even number of negatives (or a zero element, which can flip sign without penalty),
22
           # the result is simply the total sum of absolute values
23
           if negative_count % 2 == 0 or minimum_positive_value == 0:
24
               return total_sum
25
26
           # If there's an odd number of negatives, subtract twice the minimum positive
           # value to account for the one value that will remain negative
27
28
            return total_sum - minimum_positive_value * 2
29
Java Solution
 1 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
 9
               for (int value : row) {
10
                   sum += Math.abs(value); // Add the absolute value of the element to the sum
11
                   // Find the smallest absolute value in the matrix
12
13
                   minAbsValue = Math.min(minAbsValue, Math.abs(value));
                   // If the element is negative, increment the negativeCount
14
                   if (value < 0) {
15
16
                       negativeCount++;
```

// If the count of negative numbers is even or there's at least one zero, return the sum of absolute values

// Since the negative count is odd, we subtract twice the smallest absolute value to maximize the matrix sum

return sum;

return sum - (minAbsValue * 2);

```
C++ Solution
 1 class Solution {
 2 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
 8
           for (auto& row : matrix) {
 9
               // Loop over each element in the row
10
               for (int& value : row) {
11
12
                   // Add the absolute value of the current element to the total sum
13
                   sumOfAbsoluteValues += abs(value);
                   // Update the smallest absolute value encountered if current absolute value is smaller
14
15
                   minAbsValue = min(minAbsValue, abs(value));
                   // If the current element is negative, increment the negative counter
16
                   negativeCount += value < 0;</pre>
17
18
19
20
21
           // If the number of negative values is even or the minimum absolute value is 0,
22
           // we can make all elements non-negative without decreasing the sum of absolute values.
23
           if (negativeCount % 2 == 0 || minAbsValue == 0) {
               return sumOfAbsoluteValues;
24
25
26
27
           // If the number of negative values is odd, we subtract twice the smallest
28
           // absolute value to compensate for the one element that will remain negative.
29
           return sumOfAbsoluteValues - minAbsValue * 2;
30
31 };
32
```

10 11 12 13

1 /**

Typescript Solution

```
* @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
9
       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
14
           for (const value of row) {
15
               sum += Math.abs(value); // Add the absolute value to the sum
16
17
               minAbsValue = Math.min(minAbsValue, Math.abs(value)); // Update min absolute value if necessary
18
               negativeCount += value < 0 ? 1 : 0; // Increment count if the value is negative
19
20
21
22
       // If the count of negative numbers is even, the sum is already maximized
23
       if (negativeCount % 2 == 0) {
24
           return sum;
25
26
       // Otherwise, subtract double the smallest absolute value to negate an odd count of negatives
27
28
       return sum - minAbsValue * 2;
29 }
30
31 // Example usage:
32 // const matrix = [[1, -1], [-1, 1]];
33 // const result = maxMatrixSum(matrix);
  // console.log(result); // Output should be 4
35
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

Iterating through all elements takes 0(n^2) time, where n is the dimension of the square matrix (since there are n rows and n columns).

Space Complexity

cnt if the value is negative.

 All the operations inside the nested loops are constant time operations. Hence, combining these, the overall time complexity of the code is $O(n^2)$.

- 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 0(1), which indicates constant space usage regardless of the input size.