2679. Sum in a Matrix

Matrix

Sorting

Simulation

## **Problem Description**

Array

Medium

score based on specific operations performed on this matrix. We start with a score of 0, and we repeatedly perform the following two-part operation until the matrix is empty: 1. From each row in the matrix, select and remove the largest number. If any row has multiple largest numbers (i.e., a tie), any of them can be

In this problem, we are provided with a 0-indexed 2D integer array nums, which represents a matrix. The goal is to calculate a

- selected. 2. After removing these numbers from each row, identify the largest number among them and add that number to the score.

By sorting each row, we guarantee that the largest number in each row will be at the end.

Heap (Priority Queue)

The required output is the final score once there are no more numbers left in the matrix. It's important to keep track of the largest numbers being selected during each operation and ensuring that the correct value (the

maximum from these selections) is added appropriately to the score.

The intuition behind the solution comes from the way the operations on the matrix are defined. Since we always need to select

## the largest number from each row and find the overall maximum to add to the score, sorting the rows can simplify the problem.

This operation effectively "flips" the matrix so that each row becomes a column. Then, the algorithm iterates through the new "rows" (original columns) and computes the maximum value. This value is the one that should be added to the score since it represents the largest value that would be selected from the original rows during an iteration.

By continuing to sum these maxima for all "new rows," we accumulate the total score. The benefit of this approach is that the whole process takes place with a complexity close to that of the sorting operation itself, which is efficient compared to a naive

approach that might involve multiple iterations for each step. The use of Python's built-in functions like sort(), max(), and map() allows for a concise and efficient implementation of this intuition.

The implementation of the solution follows the steps that correspond to its intuitive approach described previously. Let's walk through each part of the implementation:

## in ascending order. By sorting each row, we ensure that the largest number in each row will be at the end.

return sum(map(max, zip(\*nums)))

Solution Approach

 This is done with the following line of code: for row in nums: row.sort()

Sort each row of the nums matrix. In Python, this can be achieved using the sort() method, which sorts the elements of a list

function. The zip(\*nums) effectively transposes the matrix, so the last elements of each original row (which are the largest due to the sort) become the elements of the new "rows." This "transposition" and finding the maximum occurs in the following line of code:

The map() function applies max to each new row (actually a column of the original matrix) and finds the maximum element,

The next step is to find the largest number that was removed from each row and add that to the score. Since the rows are

sorted, we can use the fact that in Python, the max() function can be applied to a list of lists by using the zip(\*iterables)

which corresponds to the maximum number we removed from the original rows. Finally, the sum() function takes the iterable produced by map() (which consists of the largest numbers from each

in a straightforward manner, and sum() to accumulate these values into the final score.

operation) and sums them up, thus calculating the final score.

- The combination of sort(), max(), zip(), and map() provides an elegant and efficient solution. It uses sorting to rearrange the elements, a matrix transpose operation to work across rows as if they were columns, the max() function to grab the largest values
- Consider the following small matrix for our example: nums = [

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

# Before sorting:

[4, 5, 6],

[1, 2, 9],

[4, 5, 6],

[1, 2, 9],

[4, 1, 3],

[6, 9, 8]

[3, 7, 8]

[7, 8, 3]

**Example Walkthrough** 

```
After sorting each row:
```

Largest numbers, which are the maxima of the transposed rows:

The max of the first transposed row is 4

The max of the third transposed row is 9

The max of the second transposed row is 7

def matrix\_sum(self, nums: List[List[int]]) -> int:

# Sort each row of the matrix in ascending order

# Transpose the matrix to access columns as rows

# And compute the sum of these maximum values

return sum(map(max, transposed\_matrix))

# Example of using the Solution class to find matrix sum

Let's walk through the steps of the solution approach using this matrix:

Sort each row of the nums matrix in ascending order:

```
For each new row (originally a column), we select the maximum value (which, in this case, would be the originally last item of
each row due to sorting).
```

Using zip (\*nums), we transpose the matrix so we can easily access the largest numbers (the last elements of each row):

As per the first step, rows are sorted, and now the largest number in each row is at the end.

Transposed and zipped matrix (effectively "flip" so that each row is now a column):

Finally, we sum up these maxima to get the final score, which is 4 + 7 + 9 = 20.

Solution Implementation **Python** 

# Find the max element in each column (since rows are sorted, it is the last element in each row after transposition)

The sum of these maxima gives us the solution to the problem, which is the final score. In this example, the score is 20.

```
[4, 5, 6],
#
      [7, 8, 9]
# sol = Solution()
# print(sol.matrix_sum(matrix)) # Output: 18
```

[1, 2, 3],

from typing import List

for row in nums:

row.sort()

transposed\_matrix = zip(\*nums)

class Solution:

# matrix = [

Java

class Solution {

int matrixSum(vector<vector<int>>& matrix) {

sort(row.begin(), row.end());

int totalSum = 0; // Initialize sum of max elements to 0

for (int col = 0; col < matrix[0].size(); ++col) {</pre>

maxElem = max(maxElem, row[col]);

// Return the total sum of max elements of all columns

// Sort each row in ascending order

for (auto& row : matrix) {

for (auto& row : matrix) {

// Loop through each column

totalSum += maxElem;

function matrixSum(matrix: number[][]): number {

// Iterate over the columns of the matrix

for (const row of matrix) {

totalSum += maxInColumn;

return totalSum;

for (const row of matrix) {

**TypeScript** 

public:

```
class Solution {
   // Method to calculate the sum of the maximum elements in each column of the matrix.
    public int matrixSum(int[][] matrix) {
       // Sort each row of the matrix to ensure elements are in non-decreasing order.
        for (int[] row : matrix) {
           Arrays.sort(row);
       // Initialize the sum that will eventually store the answer.
       int sum = 0;
       // Traverse each column of the sorted matrix to find the maximum element.
        for (int col = 0; col < matrix[0].length; ++col) {</pre>
            int maxInColumn = 0; // Variable to keep track of the max element in the current column.
           // Iterate through each row to find the maximum element for the current column.
            for (int[] row : matrix) {
                maxInColumn = Math.max(maxInColumn, row[col]); // Update the max for the column if a larger element is found.
           // After finding the maximum element in the column, add it to the sum.
            sum += maxInColumn;
       // Return the final sum, which is the total of all maximum elements in each column.
        return sum;
#include <vector>
#include <algorithm> // Include algorithm library for sort and max functions
```

// Function that calculates the sum of maximum elements in each column after sorting each row

int maxElem = 0; // Variable to store the max element in the current column

// Loop through each row to find the max element in the current column

// Update maxElem if we find a larger element in current column

// Add the max element of the current column to the total sum

```
row.sort((a, b) => a - b);
let totalSum = 0; // Initialize a variable to store the sum of maximums from each column
```

// Sort each row in the matrix to have numbers in ascending order

for (let columnIndex = 0; columnIndex < matrix[0].length; ++columnIndex) {</pre>

maxInColumn = Math.max(maxInColumn, row[columnIndex]);

// Add the maximum value of the current column to the total sum

// Iterate over each row to find the maximum value in the current column

```
return totalSum; // Return the computed sum
from typing import List
class Solution:
   def matrix_sum(self, nums: List[List[int]]) -> int:
       # Sort each row of the matrix in ascending order
        for row in nums:
            row.sort()
       # Transpose the matrix to access columns as rows
        transposed_matrix = zip(*nums)
       # Find the max element in each column (since rows are sorted, it is the last element in each row after transposition)
        # And compute the sum of these maximum values
        return sum(map(max, transposed_matrix))
# Example of using the Solution class to find matrix sum
# matrix = [
     [1, 2, 3],
     [4, 5, 6],
      [7, 8, 9]
```

let maxInColumn = 0; // Initialize a variable to store the maximum value in the current column

### element of each column after transposition. Sorting: Each row is sorted individually using row.sort(), which typically uses Tim Sort, an algorithm with a worst-case time

because 0(m \* n log n) is the dominating term.

for storing the maximums of each column.

# print(sol.matrix\_sum(matrix)) # Output: 18

Time and Space Complexity

# sol = Solution()

**Time Complexity:** 

columns in nums, then the sorting step has a time complexity of  $0(m * n \log n)$ . **Finding max and summing**: The zip(\*nums) function is used to transpose the matrix, and map(max, ...) is used to find the

complexity of O(n log n) for sorting a list of n elements. If m represents the number of rows and n represents the number of

maximum element in each column. Since there are n columns, and finding the max takes 0(m) time for each column, this step

The time complexity of the matrixSum method involves two steps: sorting the rows of the matrix and finding the maximum

- has a time complexity of 0(m \* n). The final summing of these values is done in 0(n). Combining the two steps, the overall time complexity is  $0(m * n \log n) + 0(m * n) + 0(n)$ , which simplifies to  $0(m * n \log n)$
- **Space Complexity:**
- **Sorting:** Sorting is done in-place for each row, so no additional space is proportional to the size of the input matrix is used. Therefore, it does not increase the asymptotic space complexity.
- Transposing and finding max: The zip function returns an iterator of tuples, which, when combined with map, doesn't create a list of the entire transposed matrix but rather creates one tuple at a time. Thus, the space required for this operation is O(n)
  - Thus, the space complexity of the matrixSum method is O(n).