2718. Sum of Matrix After Queries

Medium Array **Hash Table**

Problem Description In this problem, you are given two parameters. The first is an integer n, which indicates the size of a square matrix that you are to

consider. This matrix is initially filled with zeroes and has dimensions n x n. The second parameter is queries, a 2D array where each element represents a specific action to be performed on the matrix. Each query has three elements: type_i: indicates whether the action is to be performed on a row (0) or on a column (1). • index_i: the zero-based index of the row or column that the action is to be applied to.

Leetcode Link

- val_i: the value that will replace the current contents of the specified row or column. The goal of the problem is to apply each query to the matrix in the given order and then return the sum of all values in the matrix
- after all queries have been applied.

To find the solution efficiently, we have to take into account that applying a query to a row or column may overwrite the values previously set by another query. One intuitive approach to avoid unnecessary computations is to start from the last query and move

to the first, because once a whole row or column is overwritten, further modifications to it won't affect our final sum.

Intuition

The algorithm uses two sets, row and col, to keep track of all the rows and columns that have already been affected by a query. This is important because if a row or column is modified multiple times, only the last modification (the one encountered first in our reverse iteration) will stick by the end.

• If the query is of type 0 (row modification) and the row hasn't been affected by any previous query, we add to our running sum ans the value v multiplied by the number of columns n minus the number of columns already affected. Then the row index i is added to the set row.

• If the query is of type 1 (column modification) and the column hasn't been affected by any previous query, we similarly add to

ans the value v multiplied by the number of rows n minus the number of rows already affected. The column index i is then added

to the set col.

For each query, starting from the last one towards the first, the algorithm does the following:

- This ensures that each part of the matrix is counted exactly once, leading to an efficient computation of the total sum after applying all queries.
- The solution utilizes a set data structure for both rows and columns to keep track of which ones have already been updated. A set is a good choice here because it allows for constant time complexity 0(1) operations for adding elements and checking membership. Here's the algorithm in a detailed walk-through:

1. Initialize two sets, row and col, and a variable ans to store the sum of the matrix's values. 2. Iterate through the queries array in reverse order using for t, i, v in queries[::-1]:. This is crucial, as we only want to

count the last change made to a row or column; reverse iteration ensures that we encounter the last update first for each row or

column update.

column.

Solution Approach

 Check if the current row index i is not in the row set, which means it hasn't been updated before in this reverse iteration. ∘ If the row hasn't been updated before, multiply the new value v by the number of columns n minus the number of columns that have already been updated (len(col)). This calculation only accounts for the cells that have not been modified by a

 Add the current row index i to the set row. 4. If the query is to update a column (t == 1):

Add the result to ans to increment the matrix sum.

• Check if the current column index i is not in the col set.

Let us consider the following example to illustrate the solution approach:

Suppose our matrix dimensions n is 3, so we have a 3×3 matrix:

[0, 0, 5], // Query 1: Update row 0 with value 5

1. Initialize two sets row and col and a variable ans to 0.

Add 12 to ans, which becomes 12.

Add row index 1 to the row set: row = {1}.

Query 2 is [1, 2, 3] (update column 2 with 3).

Query 1 is [0, 0, 5] (update row 0 with 5).

 \circ Add row index 0 to the row set: row = {0, 1}.

The matrix after all queries have been applied looks like this:

5. Now that all queries are processed, ans is 28. This is our final sum.

it doesn't account for overlapping modifications in the same row or column.

3. If the query is to update a row (t == 0):

 Add the result to ans. • Add the current column index i to the set col. 5. After the loop concludes, ans will hold the final sum of all values in the matrix after applying all queries. 6. Return the variable ans.

In summary, this approach effectively skips over any redundant modifications to rows and columns by keeping track of whether they

have been touched by a query already or not. This Algorithm has a time complexity of roughly 0(0), where 0 is the number of

have already been updated (len(row)), accounting only for the cells not modified by a row update.

If the column hasn't been updated before, multiply the new value v by the number of rows n minus the number of rows that

Example Walkthrough

1 0 0 0

3 0 0 0

queries, since each query is processed in constant time.

[1, 2, 3], // Query 2: Update column 2 with value 3 [0, 1, 4], // Query 3: Update row 1 with value 4

And we have the following queries array where each query is type_i index_i val_i:

 Query 3 is [0, 1, 4] (update row 1 with 4). \circ Since row 1 is not in the row set, compute 4 * (3 - len(col)) = 4 * 3 = 12 because no columns have been updated yet.

```
rows.

    Add 6 to ans, which becomes 18.

 Add column index 2 to the col set: col = {2}.
```

4. Lastly, process Query 1:

3. Next, we have Query 2:

Following the solution algorithm:

- \circ Since row 0 is not in the row set, compute 5 * (3 len(col)) = 5 * (3 1) = 10 because column 2 is already updated. Add 10 to ans, which becomes 28.
- 6. Return the final sum ans, which is 28.

2. Start iterating from the last query. For this example, Query 3 is the first one to process in reverse order:

 \circ Since column 2 is not in the col set, compute 3 * (3 - len(row)) = 3 * (3 - 1) = 6 to account only for the non-modified

The sum of all values in the matrix is 5 + 5 + 3 + 4 + 4 + 3 + 0 + 0 + 3 = 27. However, the algorithm calculated a sum of 28 since

If this discrepancy is unexpected based on the problem statement, we would need to adjust the solution approach to ensure that the

value of cells that are at the intersection of the updated row and updated column is not counted twice. The presence of the

discrepancy suggests there is a nuance in the problem statement or example that needs to be reconciled with the proposed

Add the value to the sum for all columns that haven't been updated

- class Solution: def matrixSumQueries(self, n: int, queries: List[List[int]]) -> int: # Initialize sets to track unique rows and columns that have been updated

updated_rows = set()

total_sum = 0

return total_sum

Java Solution

updated_columns = set()

Initialize the answer to 0

algorithm.

10

11

12

13

14

16

17

18

19

20

27

28

29

15

16

17

18

19

20

21

22

23

24

25

26

28

29

30

31

33

34

35

36

37

38

39

41

42

43

45

46

47

48

49

50

52

51 }

Python Solution

Add the value to the sum for all rows that haven't been updated 23 total_sum += value * (n - len(updated_rows)) 24 # Mark the column as updated 25 updated_columns.add(index) 26

// The total number of queries.

long totalSum = 0;

} else {

return totalSum;

// Current query

int totalQueries = queries.length;

// Iterate through the queries in reverse order.

// Index of the row or column to update.

if (updatedRows.add(index)) {

if (updatedColumns.add(index)) {

// Value to be added to the row or column.

// Query type: 0 for row update, 1 for column update.

for (int $k = totalQueries - 1; k >= 0; --k) {$

int[] currentQuery = queries[k];

int queryType = currentQuery[0];

int index = currentQuery[1];

int value = currentQuery[2];

if (queryType == 0) {

// Return the computed sum.

Process the queries in reverse order

for query_type, index, value in reversed(queries):

Mark the row as updated

updated_rows.add(index)

if index not in updated_columns:

Return the total sum after processing all queries

else: # Query to update a column

if index not in updated_rows:

if query_type == 0: # Query to update a row

If the row has not been updated before

If the column has not been updated before

total_sum += value * (n - len(updated_columns))

class Solution { /** * Calculate the sum of all elements in a matrix after applying queries to increment rows/columns. The size of the matrix $(n \times n)$. * @param n * @param queries An array of queries where each query contains three integers: [type, index, value]. Type 0 means add 'value' to a row at 'index', type 1 means add 'value' to a column at 'index'. * @return The sum of all the elements in the matrix after applying all the queries. 9 10 public long matrixSumQueries(int n, int[][] queries) { 11 // Use hash sets to keep track of updated rows and columns to avoid repetitive additions. 12 Set<Integer> updatedRows = new HashSet<>(); 13 Set<Integer> updatedColumns = new HashSet<>(); 14

// Add value to each column in the row except those columns that have already been updated.

// Add value to each row in the column except those rows that have already been updated.

This function calculates the sum of matrix elements after applying a series of increment operations to all elements of specific

'queries' is a vector of vector<int> containing the increment operations in the form of [type, index, value].

reverse(queries.begin(), queries.end()); // Reverse the order of queries to process from last to first

The function returns the resulting sum of the matrix after all increment operations have been applied.

unordered_set<int> processedRows, processedCols; // Sets to track processed rows and columns

// This will hold the final sum of the matrix elements after applying the queries.

// If it's a row update and the row hasn't been updated before

totalSum += 1L * (n - updatedColumns.size()) * value;

totalSum += 1L * (n - updatedRows.size()) * value;

'n' is the size of the $n \times n$ matrix, where each element starts at 0.

long long matrixSumQueries(int n, vector<vector<int>>& queries) {

} else { // If the query type is '1', it targets a column.

answer += value * (n - processedRows.size);

processedCols.add(index);

// Return the final calculated answer.

Time and Space Complexity

if (!processedCols.has(index)) { // Check if the column hasn't been processed.

'value' is the amount by which to increment the specified row or column.

'index' specifies the row or column index to increment.

// Loop over each query in the reversed queries

for (const auto& query : queries) {

'type' specifies the operation type: 0 for row increment, 1 for column increment.

long long sum = 0; // Initialize the sum as a long long to avoid overflow

// If it's a column update and the column hasn't been updated before

```
C++ Solution
  1 #include <vector>
  2 #include <unordered_set>
```

public:

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

26

27

28

29

30

31

32

33

34

35

counting.

using namespace std;

class Solution {

```
23
                 int type = query[0]; // Type of operation (row or col increment)
                 int index = query[1]; // Index of row or column
 24
 25
                 int value = query[2]; // Value to add
 26
                 // Process row increment if type is 0
 27
 28
                 if (type == 0) {
 29
                     // Check if the row hasn't been processed already
 30
                     if (processedRows.count(index) == 0) {
 31
                         // Add to sum the value times the number of non-processed columns
                         sum += static_cast<long long>(n - processedCols.size()) * value;
 32
 33
                         // Mark the row as processed
 34
                         processedRows.insert(index);
 35
 36
                 } else { // Process column increment otherwise
 37
                     // Check if the column hasn't been processed already
 38
                     if (processedCols.count(index) == 0) {
 39
                         // Add to sum the value times the number of non-processed rows
 40
                         sum += static_cast<long long>(n - processedRows.size()) * value;
 41
                         // Mark the column as processed
 42
                         processedCols.insert(index);
 43
 44
 45
 46
 47
             return sum; // Return the calculated sum
 48
 49
    };
 50
Typescript Solution
1 // Declares a function to calculate the sum of matrix elements based on provided queries.
2 // n: the size of the square matrix.
  // queries: an array of queries where each query is an array containing a type (0 or 1),
4 // an index i, and a value v.
   function matrixSumQueries(n: number, queries: number[][]): number {
       // Initialize sets to keep track of processed rows and columns.
       const processedRows: Set<number> = new Set();
       const processedCols: Set<number> = new Set();
       // Initialize the answer to store the sum of query results.
       let answer = 0;
10
11
12
       // Reverse the array of queries to process them in the last-in-first-out order.
13
       queries.reverse();
14
15
       // Iterate over each query in the reversed array.
       for (const [type, index, value] of queries) {
16
           if (type === 0) { // If the query type is '0', it targets a row.
               if (!processedRows.has(index)) { // Check if the row hasn't been processed.
                   // Add the value times the number of unprocessed columns to the answer and mark the row as processed.
20
                   answer += value * (n - processedCols.size);
                   processedRows.add(index);
21
22
```

// Add the value times the number of unprocessed rows to the answer and mark the column as processed.

Time Complexity The time complexity of the code can be determined by looking at the code within the for loop, which is executed once for each

Space Complexity

In summary:

return answer;

The operations inside the loop, such as if i not in row and if i not in col, have an average-case time complexity of 0(1) due to the constant-time complexity of set operations in Python for average cases. However, in the worst-case scenario (e.g. if hash collisions become frequent), these operations could degenerate to O(n) per operation. When a row or column has not already been included in the row or col sets, the code performs a multiplication and an addition (v * (n - len(col)) or v * (n - len(row)), respectively). The multiplication and addition operations are 0(1).

Since each query is only processed once, and the operations within the for loop are (on average) constant time, the average case

time complexity of the entire code is 0(m), where m is the number of queries. However, considering the worst-case scenario for the

query. The for loop iterates over the queries in reverse order. For each query, the code checks whether the indexed row or column

(indicated by i) has been previously updated by checking membership in the row and col sets.

The provided code snippet appears to calculate the sum for a set of matrix sum queries. Each query consists of a type (t), an index

(i), and a value (v). Based on the type of query, either an entire row or an entire column is updated (conceptually) with the value v,

and the subsequent query computations take into account whether a row or column has already been updated to avoid double

The space complexity consists of the space required to store the intermediate data structures row and col, in addition to the input size (the queries list). • The row and col sets collectively will store at most n elements each, assuming that all rows and all columns are eventually added

set operations, the time complexity could potentially be 0(m * n).

to the sets. The space complexity is therefore O(n) due to the sets row and col.

- Average-case time complexity: O(m), where m is the number of queries
- Worst-case time complexity (considering potential set operation degradation): 0(m * n), where m is the number of queries and n is the dimension of the square matrix (row/column size) Space complexity: O(n), where n is the length or width of the square matrix