### 311. Sparse Matrix Multiplication Medium Array Hash Table Matrix

## Leetcode Link

## Problem Description The problem at hand requires us to perform the multiplication of two sparse matrices. A sparse matrix is a matrix that is comprised

mostly of zero values. The input consists of two matrices mat1 and mat2 where mat1 is of size m x k and mat2 is of size k x n. The task is to compute the matrix product of mat1 and mat2 and return the resulting matrix. It's assumed that the multiplication of the given matrices is always valid, which means that the number of columns in mat1 is equal to the number of rows in mat2. The output should also be in the form of a sparse matrix where the spaces are optimized to store only non-zero values as much as possible. Intuition

computation time and space. The idea behind the solution approach is to preprocess each input matrix to filter out all the zero values and keep track of the nonzero values along with their positions. This preprocessing step helps to perform multiplication operations only when it is needed (i.e., only with the non-zero values).

The intuitive approach to matrix multiplication involves three nested loops, iterating through each row of the first matrix and each

column of the second matrix to calculate the values of the resulting matrix. However, since we are given a condition that these are

sparse matrices, many of these calculations will involve multiplication by zero, which is unnecessary and can be skipped to save

The function f(mat) in the provided solution converts a matrix into a list of lists, where each sublist contains tuples. Each tuple has two elements: the column index and the value of the non-zero element in the matrix. This effectively compresses the matrix to store only the necessary information needed for the multiplication. Once we have these compressed representations g1 and g2 of mat1 and mat2, respectively, we create an answer matrix ans initialized

with zeros. For each row of mat1, we look at the non-zero values and their column indexes. For each of these non-zero values, we find the respective row (with matching column index) in g2 and multiply the corresponding elements. The product is then added to the correct position in the answer matrix.

This approach significantly reduces the number of multiplication operations, particularly when the matrices have a lot of zeros, which

is very common in applications that deal with sparse data. Solution Approach

The solution follows a two-step approach: preprocessing the input matrices to extract the non-zero values with their positions, and

then performing a modified multiplication that only operates with these non-zero values.

multiplication - that is, the column position and value of non-zero elements.

Preprocessing Step

index and the value of all non-zero elements in a new data structure which we refer to as g. For every row i in mat, a list g[i] is created. For every non-zero element x in row, a tuple (j, x) is appended to g[i], where j is the

column index of x. This results in a list of lists where each sublist represents a row and contains only the relevant data for

Once we have applied f to both mat1 and mat2, obtaining g1 and g2, we proceed to the actual multiplication.

The preprocessing function f(mat) is a decisive optimization in the algorithm. It takes advantage of the matrix's sparsity to reduce

the complexity of the multiplication step. The function goes through each element of the given matrix mat and records the column

## Matrix Multiplication Step

perform the next steps:

Example Walkthrough

mat2.

mat1 (2×3 matrix):

1 [1, 0, 0] 2 [0, 0, 3]

3 [0, 4]

sublists each containing n zeros.  $\{[0] * n \text{ for } \_ \text{ in } range(m)\}$  creates a list with m elements, each of which is a list of n zeros. 2. We then iterate through each row i of mat1. For each non-zero element x in this row (represented as a tuple (k, x) in g1[i]), we

3. This accumulator step, ans [i] [j] += x \* y, is the core of matrix multiplication. It adds up the product of corresponding

This algorithm maintains an efficient computation by only considering the relevant (non-zero) elements in the multiplication,

1. We initialize the answer matrix ans with the correct dimensions m x n and fill it with zeros. To do this, we build a list of lists with m

For every tuple (j, y) in g2[k], which represents the non-zero elements of the k-th row in mat2, we multiply the value x from

elements from pairwise aligned rows of mat1 and columns of mat2. 4. The nested loops over g1[i] and g2[k] ensure that we only consider terms that contribute to the final answer, avoiding the needless multiplication by zero which is the central inefficiency in dense matrix multiplication.

mat1 with the value y from mat2, and accumulate the product into the appropriate cell in the answer matrix ans [i] [j].

readability and conciseness of the code, while the use of a list of lists as a data structure allows for fast access and update of individual elements.

Let's walk through a simple example to illustrate the solution approach. Suppose we have the following two sparse matrices mat1 and

significantly speeding it up for sparse matrices. The use of list comprehensions and tuple unpacking in Python contributes to the

mat2 (3×2 matrix): 1 [1, 2] 2 [0, 0]

The expected result of the multiplication will be a 2×2 matrix. Now, let's use the solution approach to calculate this.

### First, we preprocess mat1 and mat2 into their sparse representations g1 and g2 using function f(mat). Non-zero values and their column indices are kept.

Result of f(mat1) is g1 (list of lists of tuples):

Result of f(mat2) is g2 (list of lists of tuples):

Next, we perform the matrix multiplication:

[(0, 1)], // Only element in first row is `1` at column `0`

non-zero elements in row 0 of g2 and multiply:

non-zero elements in row 2 of g2 and multiply:

The ans matrix now contains the result of the multiplication:

1 # Import type hints for better code readability

return sparse\_matrix

m, n = len(mat1), len(mat2[0])

for i in range(m):

return result\_matrix

# Iterate through each row of mat1

sparse\_mat1 = create\_sparse\_matrix(mat1)

sparse\_mat2 = create\_sparse\_matrix(mat2)

# Get the dimensions for the resulting matrix

# Initialize the resulting matrix with zeros

result\_matrix = [[0] \* n for \_ in range(m)]

[(2, 3)] // Only element in second row is '3' at column '2'

**Preprocessing Step** 

[(0, 1), (1, 2)], // Elements are `1` at column `0` and `2` at column `1` 

1. Initialize the answer matrix ans filled with zeros. In this case, an 2x2 matrix would look like: 1 ans = [

**Matrix Multiplication Step** 

[0, 0],

[0, 0]

1 ans = [

[1, 2],

[0, 12]

Python Solution

class Solution:

9

10

16

17

18

19

20

21

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

6

8

9

10

11

12

13

14

15

16

17

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

};

};

from typing import List

```
2. For each row i in g1, and for each tuple (k, x) in g1[i], we do the following:
    o If i is 0 (first row in mat1), we process g1[0], which is [(0, 1)]. There's only one non-zero value 1 at column 0. We find all
```

∘ If i is 1 (second row in mat1), we process g1[1], which is [(2, 3)]. There's only one non-zero value 3 at column 2. We find all

■ We multiply 1 (from g1[0][0]) \* 1 (from g2[0][0]) and add it to ans[0][0].

■ We multiply 1 (from g1[0][0]) \* 2 (from g2[0][1]) and add it to ans[0][1].

We multiply 3 (from g1[1][0]) \* 4 (from g2[2][1]) and add it to ans [1][1].

def multiply(self, mat1: List[List[int]], mat2: List[List[int]]) -> List[List[int]]:

def create\_sparse\_matrix(matrix: List[List[int]]) -> List[List[tuple]]:

# Helper function to create a sparse matrix representation

sparse\_matrix = [[] for \_ in range(len(matrix))]

# Initialize a list to store the sparse representation

# Create the sparse matrix representations for both input matrices

# Iterate through the sparse representation of the row from mat1

# Multiply and add to the resulting matrix

int[][] result = new int[m][n]; // initialize the resulting matrix

// Convert the matrices to lists containing only non-zero elements

List<int[]>[] nonZeroValuesMat1 = convertToNonZeroList(mat1);

List<int[]>[] nonZeroValuesMat2 = convertToNonZeroList(mat2);

// For each non-zero pair (column index, value) in row i

int columnIndexMat1 = pair1[0], valueMat1 = pair1[1];

for (int[] pair2 : nonZeroValuesMat2[columnIndexMat1])

for col\_index\_mat2, value\_mat2 in sparse\_mat2[col\_index\_mat1]:

result\_matrix[i][col\_index\_mat2] += value\_mat1 \* value\_mat2

int m = mat1.length, n = mat2[0].length; // determine the resulting matrix dimensions

// Multiply the non-zero elements with corresponding elements in second matrix

for col\_index\_mat1, value\_mat1 in sparse\_mat1[i]:

# Return the resulting matrix after multiplication

// The main method to perform matrix multiplication

public int[][] multiply(int[][] mat1, int[][] mat2) {

// Iterate through each row of the first matrix

for (int[] pair1 : nonZeroValuesMat1[i]) {

// Perform multiplication using the sparse representation

result[i][j] += value1 \* value2;

for (auto& [j, value2] : sparseMatrix2[k]) {

int rows = matrix.size(); // Number of rows in the matrix

int cols = matrix[0].size(); // Number of columns in the matrix

std::vector<std::vector<std::pair<int, int>>> sparseRepresentation(rows);

sparseRepresentation[i].emplace\_back(j, matrix[i][j]);

// Converts a matrix to a sparse representation to optimize multiplication of sparse matrices.

std::vector<std::vector<std::pair<int, int>>> convertToSparse(std::vector<std::vector<int>>& matrix) {

if (matrix[i][j] != 0) { // Filter out zero values for sparse representation

for (auto& [k, value1] : sparseMatrix1[i]) {

for (int i = 0; i < rows; ++i) {

for (int i = 0; i < rows; ++i) {

return sparseRepresentation;

for (int j = 0; j < cols; ++j) {

function multiply(mat1: number[][], mat2: number[][]): number[][] {

const numRowsOfMat1: number = mat1.length; // Number of rows in mat1

const filterZeros = (matrix: number[][]): [number, number][][] => {

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

const numColsOfMat2: number = mat2[0].length; // Number of columns in mat2

// Function to filter out all zero values and keep only non-zero values with their column index

const nonZeroValues: [number, number][][] = Array.from({ length: numRows }, () => []);

for (const [colIndexOfMat2, valueOfMat2] of filteredMat2[colIndexOfMat1]) {

resultMatrix[i][colIndexOfMat2] += valueOfMat1 \* valueOfMat2;

const numRows: number = matrix.length; // Number of rows in the given matrix

nonZeroValues[row].push([col, matrix[row][col]]);

// Get the dimensions required for the resulting matrix

for (let row = 0; row < numRows; ++row) {</pre>

return nonZeroValues;

// Get non-zero values for both matrices

for (let i = 0; i < numRowsOfMat1; ++i) {

if (matrix[row][col] !== 0) {

const filteredMat1: [number, number][][] = filterZeros(mat1);

const filteredMat2: [number, number][][] = filterZeros(mat2);

// Perform matrix multiplication using the sparse representations

for (const [colIndexOfMat1, valueOfMat1] of filteredMat1[i]) {

return result;

for (int i = 0; i < m; ++i) {

- This walk-through illustrates how the algorithm efficiently performs multiplication by directly accessing the non-zero elements and their positions, avoiding unnecessary multiplications with zero, which would not contribute to the result.
- for row\_index, row in enumerate(matrix): 11 12 for col\_index, value in enumerate(row): 13 if value: # Append a tuple containing the column index and the value if non-zero 14 sparse\_matrix[row\_index].append((col\_index, value)) 15

# For non-zero elements in mat1's row, iterate through the corresponding row in mat2

# Iterate through the matrix to record non-zero values along with their column indexes

```
Java Solution
```

class Solution {

```
int columnIndexMat2 = pair2[0], valueMat2 = pair2[1];
 18
 19
                         result[i][columnIndexMat2] += valueMat1 * valueMat2; // update the result matrix
 20
 21
 22
 23
             return result;
 24
 25
 26
         // Convert matrix to list of non-zero elements
 27
         private List<int[]>[] convertToNonZeroList(int[][] matrix) {
 28
             int numRows = matrix.length, numColumns = matrix[0].length;
 29
             List<int[]>[] nonZeroList = new List[numRows];
 30
 31
             // Initialize the list of arrays for each row
 32
             Arrays.setAll(nonZeroList, i -> new ArrayList<>());
 33
 34
             // Collect non-zero elements in the matrix
 35
             for (int i = 0; i < numRows; ++i) {</pre>
 36
                 for (int j = 0; j < numColumns; ++j) {</pre>
 37
                     if (matrix[i][j] != 0) {
 38
                         // For each non-zero element, add an array containing the column index and the value
 39
                         nonZeroList[i].add(new int[] {j, matrix[i][j]});
 40
 41
 42
 43
             return nonZeroList;
 44
 45 }
 46
C++ Solution
  1 #include <vector>
    #include <utility> // Include for std::pair
  4 class Solution {
    public:
         // Multiplies two matrices represented as 2D vectors.
         std::vector<std::vector<int>> multiply(std::vector<std::vector<int>>& matrix1, std::vector<std::vector<int>>& matrix2) {
             int rows = matrix1.size();
                                         // Number of rows in matrix1
  8
             int cols = matrix2[0].size(); // Number of columns in matrix2
  9
 10
             std::vector<std::vector<int>> result(rows, std::vector<int>(cols));
 11
             // Convert the matrices to a list of pairs (column index, value) for non-zero elements
 12
             auto sparseMatrix1 = convertToSparse(matrix1);
 13
 14
             auto sparseMatrix2 = convertToSparse(matrix2);
 15
```

### 5 6 // Initialize the resulting matrix with zeros const resultMatrix: number[][] = Array.from( { length: numRowsOfMat1 }, 8 () => Array.from({ length: numColsOfMat2 }, () => 0) 9 10 );

Typescript Solution

```
38
 39
 40
        // Return the resulting matrix after multiplication
 41
        return resultMatrix;
 42 }
 43
Time and Space Complexity
Time Complexity
To analyze time complexity, let's examine the code step by step.
 1. Function f(mat): Sparse Matrix Representation - This function converts a regular matrix to a sparse representation by
   recording non-zero elements' coordinates and their values. If mat has dimensions r x c with t non-zero elements, then this
   function would take 0(r * c) time, assuming the worst case where every element needs to be checked.
 2. Conversion to Sparse Representation - g1 = f(mat1) is called for the first matrix having dimensions m \times k (assuming m rows
   and k columns), and g^2 = f(mat^2) for the second matrix having dimensions k \times n. The total time for these calls is 0(m * k) + 0(k)
```

3. Matrix Multiplication - The nested loops for matrix multiplication iterate over m rows of mat1, k sparse columns of mat1, and n

sparse columns of mat2. The number of iterations would be dependent on the number of non-zero elements in the sparse

representations of mat1 and mat2. If nz1 and nz2 are the number of non-zero entries in mat1 and mat2 respectively, the time

complexity for the multiplication part is 0(nz1 \* n) + 0(nz2) because for every non-zero element in g1[i], we need to iterate

over potentially up to n elements in g2[k]. However, in a truly sparse scenario, not every element in g2[k] correlates to a non-

The overall time complexity would be 0(m \* k + k \* n + nz1 \* n + nz2) which is typically represented as 0(m \* k + k \* n) if we

# are looking at worst-case scenario considering dense matrices, where nz1 approaches m \* k and nz2 approaches k \* n. **Space Complexity**

For space complexity, we have:

\* n) = 0(m \* k + k \* n).

1. Sparse Representation - Storing the sparse representation of both matrices, which might take up to 0(m \* k + k \* n) space in the worst case (each original matrix is dense), but in practice would be 0(nz1 + nz2) for sparse matrices with nz1 and nz2 nonzero entries respectively.

zero product, so this is an upper bound.

- 2. Output Matrix ans Initializing a result matrix of dimensions  $m \times n$ , thus requiring space complexity of  $0(m \times n)$ .
- The space complexity is dominated by the larger of the two factors: output matrix and sparse representations. Therefore, the space complexity is 0(m \* n + nz1 + nz2). In worst-case scenario with dense matrices, this would simplify to 0(m \* n).