Problem 311: Sparse Matrix Multiplication

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

Given two

sparse matrices

mat1

of size

m x k

and

mat2

of size

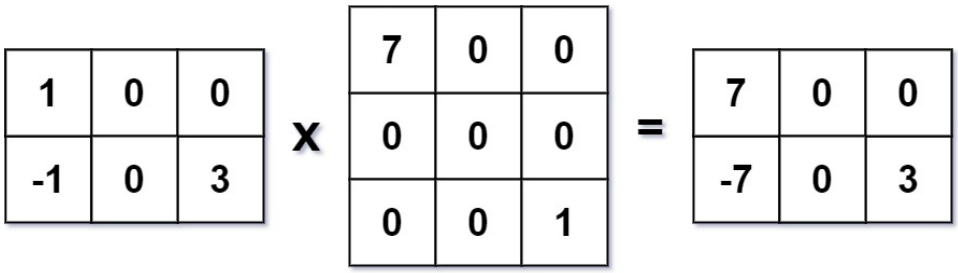
k x n

, return the result of

mat1 x mat2

. You may assume that multiplication is always possible.

Example 1:



Input:

mat1 = [[1,0,0],[-1,0,3]], mat2 = [[7,0,0],[0,0,0],[0,0,1]]

Output:

[[7,0,0],[-7,0,3]]

Example 2:

Input:

mat1 = [[0]], mat2 = [[0]]

Output:

[[0]]

Constraints:

m == mat1.length

k == mat1[i].length == mat2.length

n == mat2[i].length

1 <= m, n, k <= 100

-100 <= mat1[i][j], mat2[i][j] <= 100

Code Snippets

C++:

```
class Solution {
public:
vector<vector<int>> multiply(vector<vector<int>>& mat1, vector<vector<int>>&
mat2) {

}
};
```

Java:

```
class Solution {
public int[][] multiply(int[][] mat1, int[][] mat2) {

}
}
```

Python3:

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

Python:

```
class Solution(object):
def multiply(self, mat1, mat2):
"""
:type mat1: List[List[int]]
:type mat2: List[List[int]]
:rtype: List[List[int]]
"""
```

JavaScript:

```
/**
* @param {number[][]} mat1
* @param {number[][]} mat2
* @return {number[][]}
*/
var multiply = function(mat1, mat2) {
```

```
};
```

TypeScript:

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

};
```

C#:

```
public class Solution {
public int[][] Multiply(int[][] mat1, int[][] mat2) {

}
}
```

C:

```
/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** multiply(int** mat1, int mat1Size, int* mat1ColSize, int** mat2, int
mat2Size, int* mat2ColSize, int* returnSize, int** returnColumnSizes) {

}
```

Go:

```
func multiply(mat1 [][]int, mat2 [][]int) [][]int {

}
```

Kotlin:

```
class Solution {
fun multiply(mat1: Array<IntArray>, mat2: Array<IntArray>): Array<IntArray> {

}
}
```

Swift:

```
class Solution {
func multiply(_ mat1: [[Int]], _ mat2: [[Int]]) -> [[Int]] {

}
}
```

Rust:

```
impl Solution {
pub fn multiply(mat1: Vec<Vec<i32>>, mat2: Vec<Vec<i32>>) -> Vec<Vec<i32>> {

}
}
```

Ruby:

```
# @param {Integer[][]} mat1
# @param {Integer[][]} mat2
# @return {Integer[][]}
def multiply(mat1, mat2)

end
```

PHP:

```
class Solution {

/**
* @param Integer[][] $mat1
* @param Integer[][] $mat2
* @return Integer[][]
*/
function multiply($mat1, $mat2) {

}
}
```

Dart:

```
class Solution {
List<List<int>> multiply(List<List<int>> mat1, List<List<int>> mat2) {
```

```
}
}
```

Scala:

```
object Solution {
def multiply(mat1: Array[Array[Int]], mat2: Array[Array[Int]]):
Array[Array[Int]] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec multiply(mat1 :: [[integer]], mat2 :: [[integer]]) :: [[integer]]
def multiply(mat1, mat2) do

end
end
```

Erlang:

```
-spec multiply(Mat1 :: [[integer()]], Mat2 :: [[integer()]]) ->
[[integer()]].
multiply(Mat1, Mat2) ->
.
```

Racket:

```
(define/contract (multiply mat1 mat2)
(-> (listof (listof exact-integer?)) (listof (listof exact-integer?)) (listof
(listof exact-integer?)))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Sparse Matrix Multiplication
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public:
vector<vector<int>> multiply(vector<vector<int>>& mat1, vector<vector<int>>&
mat2) {

}
};
```

Java Solution:

```
/**
* Problem: Sparse Matrix Multiplication
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

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

}
}
```

Python3 Solution:

```
"""
Problem: Sparse Matrix Multiplication
Difficulty: Medium
Tags: array, hash
```

```
Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

class Solution:
def multiply(self, mat1: List[List[int]], mat2: List[List[int]]) ->
List[List[int]]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def multiply(self, mat1, mat2):
"""
:type mat1: List[List[int]]
:type mat2: List[List[int]]
:rtype: List[List[int]]
"""
```

JavaScript Solution:

```
/**
* Problem: Sparse Matrix Multiplication
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* @param {number[][]} mat1
* @param {number[][]} mat2
* @return {number[][]}
*/
var multiply = function(mat1, mat2) {
```

```
};
```

TypeScript Solution:

```
/**
* Problem: Sparse Matrix Multiplication
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

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

};
```

C# Solution:

```
/*
* Problem: Sparse Matrix Multiplication
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

public class Solution {
public int[][] Multiply(int[][] mat1, int[][] mat2) {

}
}
```

C Solution:

```
/*
* Problem: Sparse Matrix Multiplication
* Difficulty: Medium
```

```
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** multiply(int** mat1, int mat1Size, int* mat1ColSize, int** mat2, int
mat2Size, int* mat2ColSize, int* returnSize, int** returnColumnSizes) {

}
```

Go Solution:

```
// Problem: Sparse Matrix Multiplication
// Difficulty: Medium
// Tags: array, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

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

}
```

Kotlin Solution:

```
class Solution {
fun multiply(mat1: Array<IntArray>, mat2: Array<IntArray>): Array<IntArray> {

}
}
```

Swift Solution:

```
class Solution {
func multiply(_ mat1: [[Int]], _ mat2: [[Int]]) -> [[Int]] {

}
}
```

Rust Solution:

```
// Problem: Sparse Matrix Multiplication
// Difficulty: Medium
// Tags: array, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

impl Solution {
pub fn multiply(mat1: Vec<Vec<i32>>, mat2: Vec<Vec<i32>>) -> Vec<Vec<i32>> {

}
}
```

Ruby Solution:

```
# @param {Integer[][]} mat1
# @param {Integer[][]} mat2
# @return {Integer[][]}
def multiply(mat1, mat2)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[][] $mat1
* @param Integer[][] $mat2
* @return Integer[][]
*/
function multiply($mat1, $mat2) {
```

```
}
}
```

Dart Solution:

```
class Solution {
List<List<int>> multiply(List<List<int>> mat1, List<List<int>> mat2) {

}
}
```

Scala Solution:

```
object Solution {
def multiply(mat1: Array[Array[Int]], mat2: Array[Array[Int]]):
Array[Array[Int]] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec multiply(mat1 :: [[integer]], mat2 :: [[integer]]) :: [[integer]]
def multiply(mat1, mat2) do

end
end
```

Erlang Solution:

```
-spec multiply(Mat1 :: [[integer()]], Mat2 :: [[integer()]]) ->
[[integer()]].
multiply(Mat1, Mat2) ->
.
```

Racket Solution:

```
(define/contract (multiply mat1 mat2)
(-> (listof (listof exact-integer?)) (listof (listof exact-integer?)) (listof
(listof exact-integer?)))
```

```
)
```