**Note: All the Code is in JavaScript Programming Language**

**Question 1**

A permutation perm of n + 1 integers of all the integers in the range [0, n] can be represented as a string s of length n where:

* s[i] == 'I' if perm[i] < perm[i + 1], and
* s[i] == 'D' if perm[i] > perm[i + 1].

Given a string s, reconstruct the permutation perm and return it. If there are multiple valid permutations perm, return **any of them**.

**Example 1:**

**Input:** s = "IDID"

**Output:**

[0,4,1,3,2]

**Answer:** The problem says that we have an array that takes sting ‘s’ that consists of only ‘I’ and ‘D’ and should follow the conditions

* arr should contain integers from 0 to s.length(inclusive)
* For each character ‘I’, the corresponding number in ‘arr’ is the smallest available number; for ‘D’, it should be the largest available number.

The approach for this would be if we encounter ‘I’ in the array we would be pushing it to first with the increment in first and for ‘D’, we push to the last with the decrement in last. At last, we should be returning arr.

The time complexity would be O(n) where n is the length of the string s and space complexity would be O(n).

| **var diStringMatch = function(s) {  let first=0,last=s.length,arr=[]  for(let i=0;i<s.length;i++){  if(s[i]==='I'){  arr.push(first)  first++  }  else if(s[i]==='D'){  arr.push(last)  last--  }  }  arr.push(first++)  return arr };** |
| --- |

**Question 2**

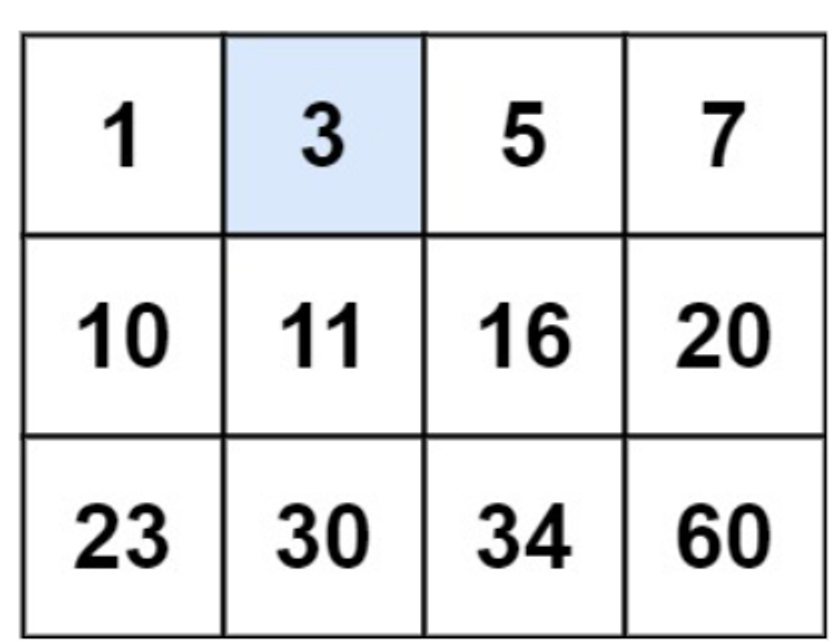
You are given an m x n integer matrix matrix with the following two properties:

* Each row is sorted in non-decreasing order.
* The first integer of each row is greater than the last integer of the previous row.

Given an integer target, return true *if* target *is in* matrix *or* false *otherwise*.

You must write a solution in O(log(m \* n)) time complexity.

**Example 1:**



**Input:** matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3

**Output:** true

**Answer:** The best approach would be to do a binary search of the complete matrix and find the target.

| var searchMatrix = function(matrix, target) {  let start = 0, end = (matrix.length \* matrix[0].length) - 1    while(start <= end) {  let mid = Math.floor((start + end) / 2)  let midNum =   matrix[Math.floor(mid / matrix[0].length)][mid % matrix[0].length]    if(midNum === target) return true   else if(midNum < target) start = mid + 1  else end = mid - 1  }  return false } |
| --- |

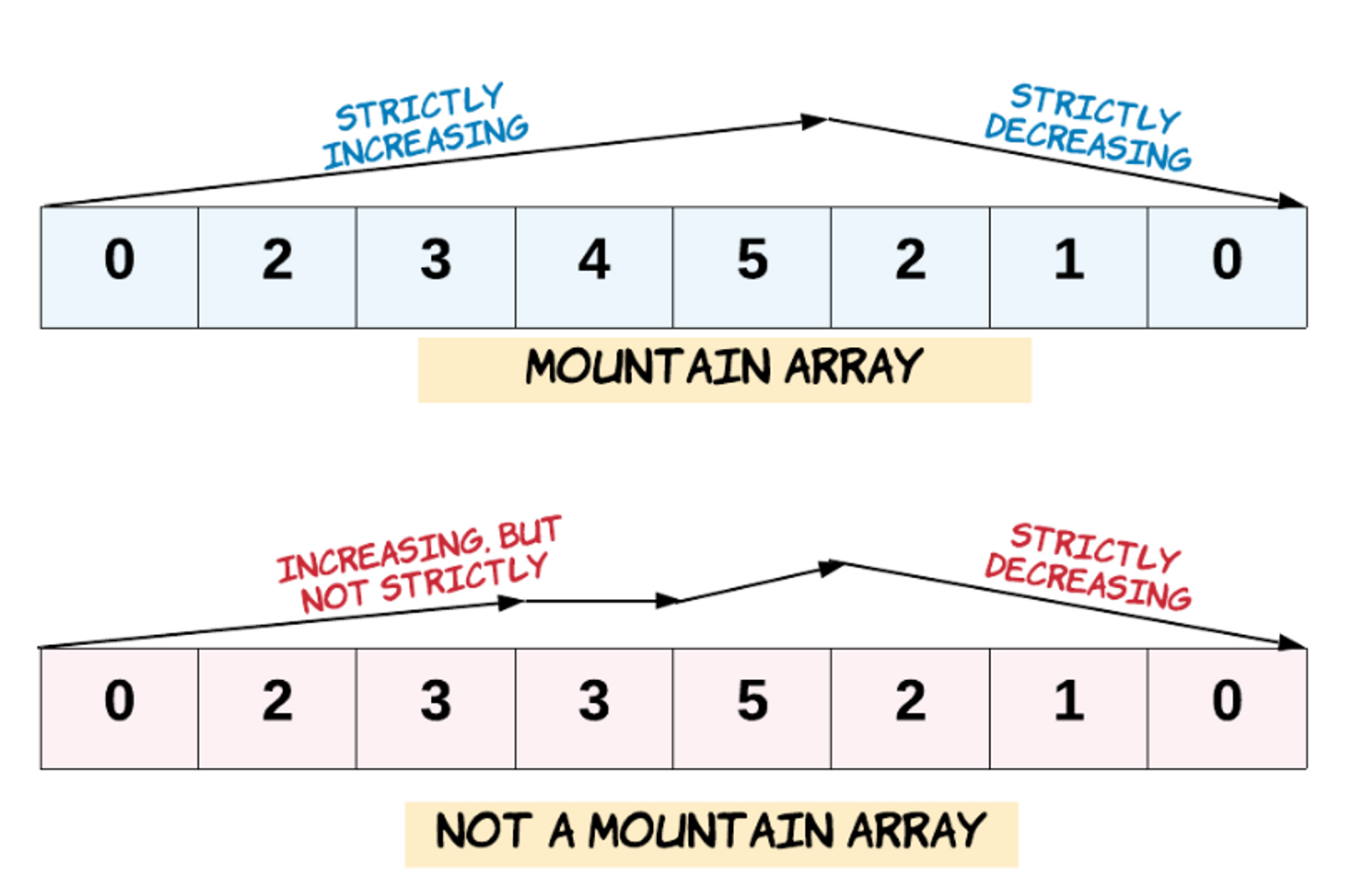
The time complexity would be O(log m\*n) and the space complexity would be O(1) as no extra space is taken in the process.

**Question 3**

Given an array of integers arr, return *true if and only if it is a valid mountain array*.

Recall that arr is a mountain array if and only if:

* arr.length >= 3
* There exists some i with 0 < i < arr.length - 1 such that:
  + arr[0] < arr[1] < ... < arr[i - 1] < arr[i]
  + arr[i] > arr[i + 1] > ... > arr[arr.length - 1] </aside>



**Example 1:**

**Input:** arr = [2,1]

**Output:**

false

**Answer:**

| /\* Test cases: 1. all increasing 2. all decreasing 3. arr length < 3 4. arr containing duplicates 5. arr not containing duplicates 6. zig-zag mountain 7. all equal \*/  // TC = O(n) var validMountainArray = function(arr) {  // 2 pointers - one will move from left to right and other will move from right to left  // so that both meet at the peak point  let left = 0,  right = arr.length - 1;    // condition: while either of the pointer is able to move  while(arr[left] < arr[left + 1] || arr[right] < arr[right - 1]) {  if(arr[left] < arr[left + 1]) {  ++left;  }  if(arr[right] < arr[right - 1]) {  --right;  }  }  // if left and right pointer does not meet at the peak - array contains equal integers or zig-zag pattern  // OR all integers are in ascending order  // OR all integers are in descending order  if(left !== right || left === arr.length - 1 || right === 0) {  return false;  }  return true; }; |
| --- |

The time complexity would be O(n) where n is the length of the input array ‘arr’. The space complexity would be O(1) as it doesn’t need extra space.

**Question 4**

Given a binary array num, return *the maximum length of a contiguous subarray with an equal number of* 0 *and* 1.

**Example 1:**

**Input:** nums = [0,1]

**Output:** 2

**Explanation:**

[0, 1] is the longest contiguous subarray with an equal number of 0 and 1.

**Answer:**

| const findMaxLength = (nums) => {  const hash = {};  let max\_length = 0;  let count = 0;  for (let i = 0; i < nums.length; i++) {  const current = nums[i];  if (current === 0) {  // if the current element is 0, then we decrement the count  count--;  } else if (current === 1) {  // if the current element is 1, then we increment the count  count++;  }   if (count === 0) {  // if the count is equal to o then we have a contiguous subarray of length equal to i+1  max\_length = i + 1;  }  if (count in hash) {    max\_length = Math.max(max\_length, i - hash[count]); // update our max length  } else {  hash[count] = i;  }   }  return max\_length; }; |
| --- |

The time complexity would be O(n) where n is the length of array and space complexity would be O(n) as n space is required to store the value in the hash table.

**Question 5**

The **product sum** of two equal-length arrays a and b is equal to the sum of a[i] \* b[i] for all 0 <= i < a.length (**0-indexed**).

* For example, if a = [1,2,3,4] and b = [5,2,3,1], the **product sum** would be 1*5 + 2*2 + 3*3 + 4*1 = 22.

Given two arrays nums1 and nums2 of length n, return *the* ***minimum product sum*** *if you are allowed to* ***rearrange*** *the* ***order*** *of the elements in* nums1.

**Example 1:**

**Input:** nums1 = [5,3,4,2], nums2 = [4,2,2,5]

**Output:** 40

**Explanation:**

We can rearrange nums1 to become [3,5,4,2]. The product sum of [3,5,4,2] and [4,2,2,5] is 3*4 + 5*2 + 4*2 + 2*5 = 40.

**Answer:**

In this problem, we will sort both nums1 and nums2 and then multiply nums1[i] with nums2[n-1-i]. This is how the product sum is minimized.

| class Solution {  minProductSum(nums1, nums2) {  nums1.sort((a, b) => a - b);  nums2.sort((a, b) => a - b);  const n = nums1.length;  let res = 0;  for (let i = 0; i < n; i++) {  res += nums1[i] \* nums2[n - i - 1];  }  return res;  } } |
| --- |

The time complexity of the above code be O(n log n) as string sorting is in the picture. The space complexity would be O(1) as no extra space is required.

**Question 6**

An integer array original is transformed into a **doubled** array changed by appending **twice the value** of every element in original, and then randomly **shuffling** the resulting array.

Given an array changed, return original *if* changed *is a* ***doubled*** *array. If* changed *is not a* ***doubled*** *array, return an empty array. The elements in* original *may be returned in* ***any*** *order*.

**Example 1:**

**Input:** changed = [1,3,4,2,6,8]

**Output:** [1,3,4]

**Explanation:** One possible original array could be [1,3,4]:

* Twice the value of 1 is 1 \* 2 = 2.
* Twice the value of 3 is 3 \* 2 = 6.
* Twice the value of 4 is 4 \* 2 = 8.

Other original arrays could be [4,3,1] or [3,1,4].

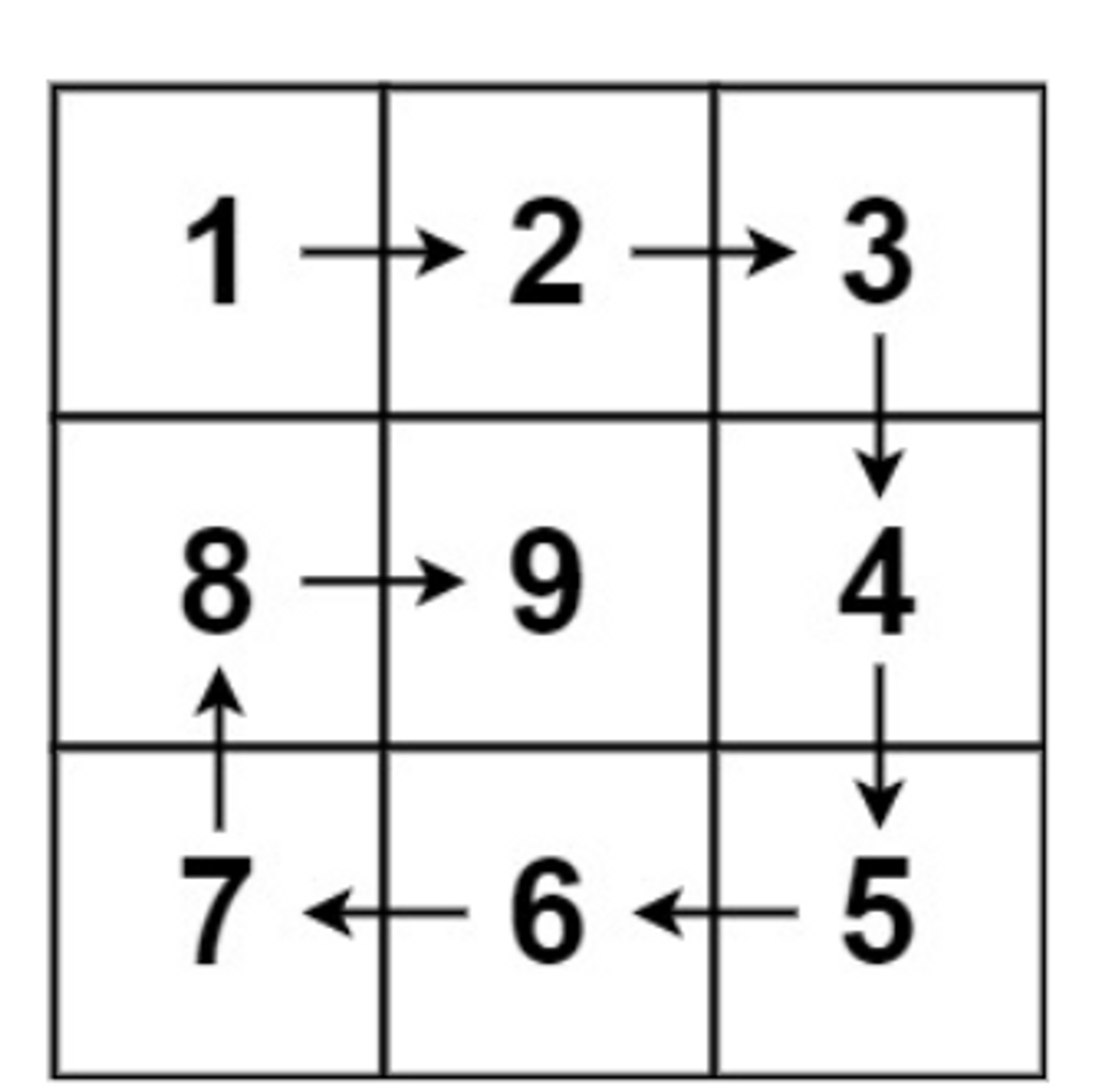
**Answer:** First and foremost, we must sort the modified array. This will allow us to find the best original, doubled pairs in a single iteration over the modified. Because js sort does in-place sorting, I make a copy of the changed argument and sort it by ascending value. To keep track of found pairs, we'll use HashMap, where keys are original array values and values are frequency of occurrence, because the altered array may contain non-unique values. Next, we iterate over the input array to see if the split by 2 value is in our HashMap, and if so, we add the value to the result and lower the HashMap counter. Finally, return the result, but first ensure that our HashMap is empty so that there are no leftovers.

The time complexity would be O(n logn) and the space complexity would be O(n) where n is the length of the array.

**Question 7**

Given a positive integer n, generate an n x n matrix filled with elements from 1 to n2 in spiral order.

**Example 1:**

****

**Input:** n = 3

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

**Answer:** When traversing the matrix in the spiral order, at any time we follow one out of the following four directions: RIGHT DOWN LEFT UP.

Suppose we are working on a 5 x 3 matrix as such:

0 1 2 3 4 5

6 7 8 9 10

11 12 13 14 15

Imagine a cursor starts off at (0, -1), i.e. the position at '0', then we can achieve the spiral order by doing the following:

1. Go right 5 times

2. Go down 2 times

3. Go left 4 times

4. Go up 1 times.

5. Go right 3 times

6. Go down 0 times -> quit

Using this concept we will find the spiral of the matrix.

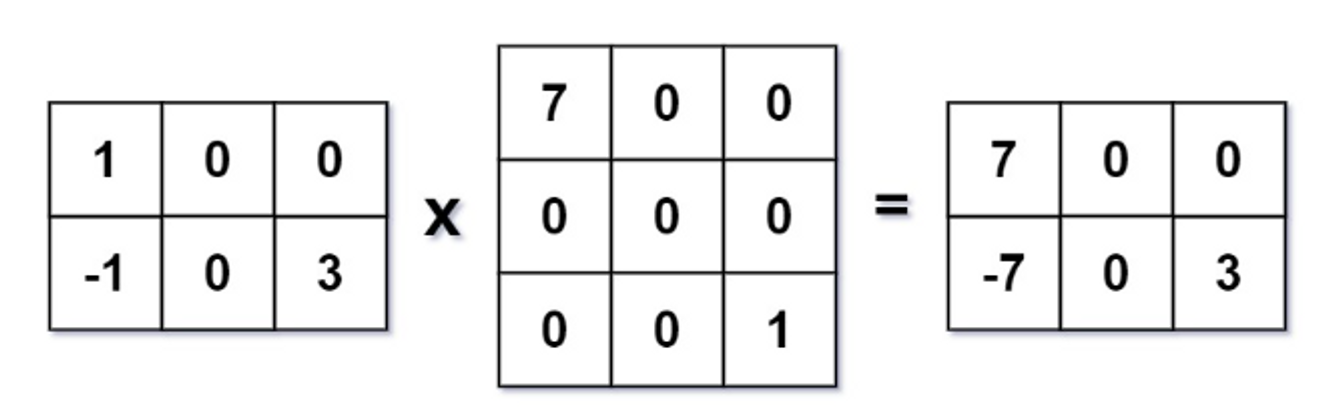
| const generateMatrix = (n) => {  const matrix = [...Array(n)].map(() => Array(n).fill(null));  const dirs = [[0, 1], [1, 0], [0, -1], [-1, 0]]; // right, down, left, up  const steps = [n, n - 1];   let num = 1;  let dir = 0;  let x = 0;  let y = -1;   while (steps[dir % 2] > 0) {  for (let i = 0; i < steps[dir % 2]; i++) {  x += dirs[dir][0];  y += dirs[dir][1];  matrix[x][y] = num++;  }   steps[dir % 2]--;  dir = (dir + 1) % 4;  }  return matrix; }; |
| --- |

The time and space complexity would be O(n ^2) where n is the input size.

**Question 8**

Given two [sparse matrices](https://en.wikipedia.org/wiki/Sparse_matrix) 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]]

**Answer:** Before continuing, ensure that A[i][k] is not 0, and then traverse the kth row of matrix B. If B[K][J] is not 0, accumulate the result matrix res[i][j] += A[i][k] \* B[k][j], so that we may efficiently compute the sparse matrix multiplication.

| class Solution {  multiply(mat1, mat2) {  const r1 = mat1.length;  const c1 = mat1[0].length;  const c2 = mat2[0].length;  const res = Array.from({ length: r1 }, () => Array(c2).fill(0));  const mp = new Map();   for (let i = 0; i < r1; i++) {  for (let j = 0; j < c1; j++) {  if (mat1[i][j] !== 0) {  if (!mp.has(i)) {  mp.set(i, []);  }  mp.get(i).push(j);  }  }  }   for (let i = 0; i < r1; i++) {  for (let j = 0; j < c2; j++) {  for (let k of mp.get(i)) {  res[i][j] += mat1[i][k] \* mat2[k][j];  }  }  }   return res;  } } |
| --- |

The time complexity would be O(r1\*c1\*c2) and space complexity would beO(r1\*c2+r1\*c1)