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

**Question 1**

Given three integer arrays arr1, arr2, and arr3 sorted in strictly increasing order, return a sorted array of only the integers that appeared in all three arrays.

Example 1:

Input: arr1 = [1,2,3,4,5], arr2 = [1,2,5,7,9], arr3 = [1,3,4,5,8]

Output: [1,5]

Explanation: Only 1 and 5 appeared in the three arrays.

**Solution:** The best way to solve this question is to find the intersection of the three arrays and store the intersection in the temp array variable.

The time complexity would be O(n1+n2+n3) where n is the sizes of arrays 1,2 and 3. The space complexity would be O(1).

| // This function prints common elements in ar1  function findCommon(ar1, ar2, ar3, n1, n2, n3)  {    // Initialize starting indexes  // for ar1[], ar2[] and ar3[]  var i = 0,  j = 0,  k = 0;    // Iterate through three arrays  // while all arrays have elements  while (i < n1 && j < n2 && k < n3)  {    // If x = y and y = z, print any of them and move ahead  // in all arrays  if (ar1[i] == ar2[j] && ar2[j] == ar3[k])  {  document.write(ar1[i] + " ");  i++;  j++;  k++;  }    // x < y  else if (ar1[i] < ar2[j]) i++;    // y < z  else if (ar2[j] < ar3[k]) j++;    // We reach here when x > y and z < y, i.e., z is smallest  else k++;  }  }    // Driver code  var ar1 = [1, 5, 10, 20, 40, 80];  var ar2 = [6, 7, 20, 80, 100];  var ar3 = [3, 4, 15, 20, 30, 70, 80, 120];  var n1 = ar1.length;  var n2 = ar2.length;  var n3 = ar3.length;    document.write("Common Elements are ");  findCommon(ar1, ar2, ar3, n1, n2, n3); |
| --- |

**Question 2**

Given two **0-indexed** integer arrays nums1 and nums2, return *a list* answer *of size* 2 *where:*

* answer[0] *is a list of all* ***distinct*** *integers in* nums1 *which are* ***not*** *present in* nums2\*.\*
* answer[1] *is a list of all* ***distinct*** *integers in* nums2 *which are* ***not*** *present in* nums1.

**Note** that the integers in the lists may be returned in **any** order.

**Example 1:**

**Input:** nums1 = [1,2,3], nums2 = [2,4,6]

**Output:** [[1,3],[4,6]]

**Explanation:**

For nums1, nums1[1] = 2 is present at index 0 of nums2, whereas nums1[0] = 1 and nums1[2] = 3 are not present in nums2. Therefore, answer[0] = [1,3].

For nums2, nums2[0] = 2 is present at index 1 of nums1, whereas nums2[1] = 4 and nums2[2] = 6 are not present in nums2. Therefore, answer[1] = [4,6].

**Answer:** The best way to do this is using ‘set’. Creating a set out of the 2 arrays and deleting the original arrays will get us the solutions.

| var findDifference = function(nums1, nums2) {  const set1 = new Set(nums1),  set2 = new Set(nums2);    nums1.forEach(set2.delete, set2);  nums2.forEach(set1.delete, set1);    return [[...set1], [...set2]] }; |
| --- |

The time complexity for this code is O(m+n) where m and n are lengths of nums1 and nums2. The space complexity would be O(n+m) as the overall required is the length m and n.

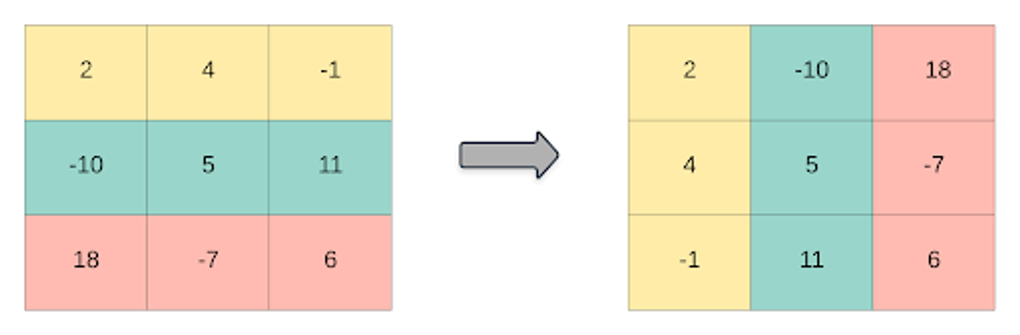
**Question 3** Given a 2D integer array matrix, return *the* ***transpose*** *of* matrix.

The **transpose** of a matrix is the matrix flipped over its main diagonal, switching the matrix's row and column indices.

**Example 1:**

Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]

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



**Answer:**

The best way to accomplish the transpose is to iterate over each array till the length and push the elements as soon as it appears row-wise.

| **var transpose = function(A) {  let result = [];    for(let i= 0; i<A[0].length; i++){  let currentCol = []  for(let j=0; j<A.length; j++){  currentCol.push(A[j][i])  }  result.push(currentCol);  }  return result };** |
| --- |

The time complexity would be O(n) where n is the length of the array. The time complexity would be O(n) as the same space is required to shift the array.

**Question 4** Given an integer array nums of 2n integers, group these integers into n pairs (a1, b1), (a2, b2), ..., (an, bn) such that the sum of min(ai, bi) for all i is **maximized**. Return *the maximized sum*.

**Example 1:**

Input: nums = [1,4,3,2]

Output: 4

**Explanation:** All possible pairings (ignoring the ordering of elements) are:

1. (1, 4), (2, 3) -> min(1, 4) + min(2, 3) = 1 + 2 = 3
2. (1, 3), (2, 4) -> min(1, 3) + min(2, 4) = 1 + 2 = 3
3. (1, 2), (3, 4) -> min(1, 2) + min(3, 4) = 1 + 3 = 4

So the maximum possible sum is 4.

**Answer:** The best approach is to find the sum of minimum elements in each pair of the sorted array. We need to sort the array first before finding the sum. The array should be in descending order and should start from the second element and iterate over the step of 2. At last the sum should be returned.

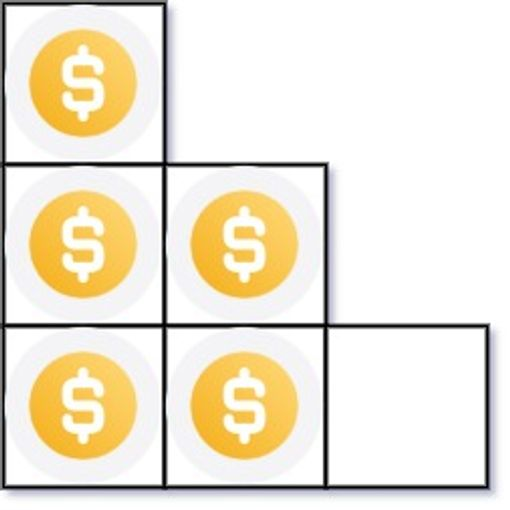
| var arrayPairSum = function(nums) {  let sumMin = 0  nums.sort((a, b) => b-a)  for (let i = 1; i < nums.length; i+=2) {  sumMin += nums[i]  }  return sumMin  }; |
| --- |

The time complexity would be O(n log n) as it sorts the array and runs the for loop.The space complexity would be O(1) as it doesn’t require any extra space.

**Question 5** You have n coins and you want to build a staircase with these coins. The staircase consists of k rows where the ith row has exactly i coins. The last row of the staircase **may be** incomplete.

Given the integer n, return *the number of* ***complete rows*** *of the staircase you will build*.

**Example 1:**

****

**Input:** n = 5

**Output:** 2

**Explanation:** Because the 3rd row is incomplete, we return 2.

**Answer:** The approach is pretty straightforward where we compare the stairs with the number of coins. If the stairs are less than the coins, then operation would be done to arrange the coin and return the respective rows built in the process.

| var arrangeCoins = function(n) {  let stairs = 0;  while(stairs <= n) { n -= stairs; stairs++; }  return stairs-1 }; |
| --- |

The time complexity would be O(n) where n is the length of stairs. The space complexity would be O(1) as it takes constant space.

**Question 6** Given an integer array nums sorted in **non-decreasing** order, return *an array of* ***the squares of each number*** *sorted in non-decreasing order*.

**Example 1:**

Input: nums = [-4,-1,0,3,10]

Output: [0,1,9,16,100]

**Explanation:** After squaring, the array becomes [16,1,0,9,100]. After sorting, it becomes [0,1,9,16,100]

**Answer:** We use a 2 pointer array here as we know the arrays are sorted. So max number will be either side of the array. If we start from opposite ends, we know that both ends will give us the respective highest squares, especially when working with negative numbers. Later we will compare the values and insert into the result array.

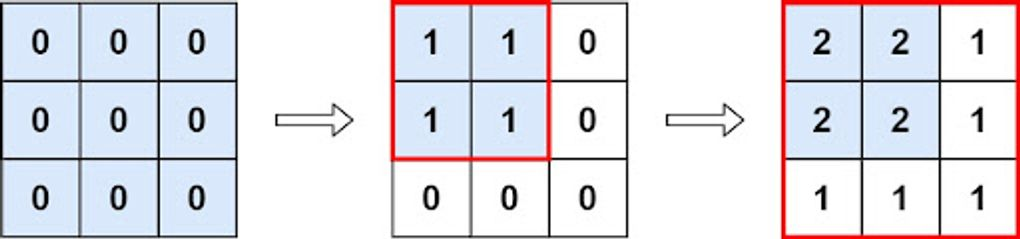
| var sortedSquares = function (nums) {  const results = new Array(nums.length);  let left\_idx = 0;  let right\_idx = nums.length-1;  let next\_highest\_square\_index = nums.length-1; // keep track where our next insert position will be in the results array   while(left\_idx <= right\_idx){  const left\_el = Math.pow(nums[left\_idx],2)  const right\_el = Math.pow(nums[right\_idx],2)  if(left\_el > right\_el){  results[next\_highest\_square\_index] = left\_el  next\_highest\_square\_index -=1;  left\_idx +=1  }  else {  results[next\_highest\_square\_index] = right\_el;  next\_highest\_square\_index -= 1;  right\_idx -= 1  }  }  return results }; |
| --- |

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

**Question 7** You are given an m x n matrix M initialized with all 0's and an array of operations ops, where ops[i] = [ai, bi] means M[x][y] should be incremented by one for all 0 <= x < ai and 0 <= y < bi.

Count and return *the number of maximum integers in the matrix after performing all the operations*

**Example 1:**

****

**Input: m = 3, n = 3, ops = [[2,2],[3,3]]**

**Output: 4**

**Explanation: The maximum integer in M is 2, and there are four of it in M. So return 4.**

**Answer:**

Finding the count of the largest integer after performing the specified actions is the task at hand. According to the question, a cell's value can only be increased if and only if it is one of the cells ranging from (0,0) to the inputs provided (p, q).

Therefore, when the value of a matrix cell is increased a greater number of times, the maximum integer will result. The most overlapping cell between the specified set of operations or matrix cell positions must be identified for a matrix cell to have the largest value since this range of cells will have more opportunities to increase in value.

In the first example, each cell value is increased from (0,0) to (2,2) in the first operation. In the second operation, each cell value is increased from (0,0) to (2,2) once more because it overlaps with (3,3). Therefore, the solution is as simple as calculating the product of the cell's minimum first and second coordinates. Because we must get the count of the largest integer in the matrix after carrying out the specified procedures, we are taking the product.

| var maxCount = function(m, n, ops) {  let length = ops.length;  if (length === 0) {  return m \* n;  }  let result = [ops[0][0], ops[0][1]];  for (let i = 1; i < length; i++) {  result[0] = Math.min(result[0], ops[i][0]);  result[1] = Math.min(result[1], ops[i][1]);  }  return result[0] \* result[1]; }; |
| --- |

The time complexity would be O(n) where n is the length and space complexity would be O(1).

**Question 8**

Given the array nums consisting of 2n elements in the form [x1,x2,...,xn,y1,y2,...,yn].

*Return the array in the form* [x1,y1,x2,y2,...,xn,yn].

**Example 1:**

**Input:** nums = [2,5,1,3,4,7], n = 3

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

**Explanation:** Since x1=2, x2=5, x3=1, y1=3, y2=4, y3=7 then the answer is [2,3,5,4,1,7].

**Answer:** Here, the middle of the array is given where we have to break. So simple we push the first element alongside the element next to the mid. The time complexity would be O(n) and space complexity would be O(n).

| var shuffle = function(nums, n) {  let res = [];  for (i = 0; i < n; i++) {  res.push(nums[i],nums[i+n]);  }  return res; }; |
| --- |