4. Median of Two Sorted Arrays

Solve using binary search approach

50. Pow(x, n)

There are basically three approaches to solve this particular problem 1.Brute force approach Logic: multiply x till n number of times to get the ans if(n<0) {x = 1/x; n = -n;} 2.Fast power calculation algorithm using recursion Logic: class Solution { public double myPow(double x, long n) { if(n < 0) { x = 1/x; n = -n; } if(n == 0) { return 1; } if(x == 1) { return 1; } if(n == 1) { return x; } double ans = 1; if(n % 2 == 0) { ans = *myPow(x*x, n/2); } else { ans = *x*myPow(x*x,n/2); } return ans; } } 3.Fast power calculation algorithm using iteration class Solution { public double myPow(double x, long n) { // iterative approach if(n < 0) { x = 1/x; n = -n; } double result = 1; double currentResult = x; for(long i=n; i >0; i/=2) { if(i % 2 == 1) { result = currentResult*result; } currentResult = currentResult*currentResult; } return result; } }

74. Search a 2D Matrix

try using another approach

88. Merge Sorted Array

try without using extra space

442. Find All Duplicates in an Array

solve using cyclic sort approach

832. Flipping an Image

learn from the solution :- Best solution/Accepted Solution

867. Transpose Matrix

Reflect rectangular matrix about primary/main diagonal. This operation is called transpose where rows and columns get swapped Extra space can't be avoided because if the matrix is not square, then dimensions of the result matrix swap, so the original matrix can't be used as the result. e.g. 4 x 3 matrix becomes 3 x 4. It can be avoided when the matrix is square. T/S: O(mn)/O(mn), where m x n are the dimensions of the matrix and m may or may not be equal to n.

public int[][] transpose(int[][] matrix) { var row = matrix.length; var col = matrix[0].length; var trans = new int[col][row]; for (var i = 0; i < row; i++) for (var j = 0; j < col; j++) trans[j][i] = matrix[i][j]; return trans; } Related questions Reflect square matrix about primary/main diagonal

T/S: O(n^2)/O(1), where n x n are the dimensions of the matrix

public int[][] reflectSquareMatrixAboutPrimaryDiagonal(int[][] matrix) { for (var i = 0; i < matrix.length; i++) for (var j = i + 1; j < matrix.length; j++) { var temp = matrix[i][j]; matrix[i][j] = matrix[j][i]; matrix[j][i] = temp; } return matrix; } Reflect square matrix about secondary diagonal

T/S: O(n^2)/O(1)

public int[][] reflectSquareMatrixAboutSecondaryDiagonal(int[][] matrix) { int n = matrix.length; for (int i = 0; i < n - 1; i++) for (int j = 0; j < n - 1 - i; j++) { int temp = matrix[i][j]; matrix[i][j] = matrix[n - 1 - j][n - 1 - i]; matrix[n - 1 - j][n - 1 - i] = temp; } return matrix; }

509. Fibonacci Number

try using dynamic programming for better optimization

989. Add to Array-Form of Integer

try it again

1252. Cells with Odd Values in a Matrix

try to do it in O(N+M+indices.length)

1572. Matrix Diagonal Sum

Approach:- My intutuion behind this approach is, the primary diagonals are the elements in the form of arr[index][index]. For example: for a 3*3 array the the primary diagonals are arr[0][0], arr[1][1], arr[2][2]. Now for the secondary diagonal: The secondary diagonals will be elements like arr[index][matrixLength-1] for first row, arr[index][matrixLength-2] and so on. For example: for a 3*3 matrix the secondary diagonal elements will be arr[0][2], arr[1][1], arr[2][0].

Now, as you can see above the element arr[1][1] is repeating two times, hence after doing sum for the primary diagonal elements we will assign them 0. So, the benefit of doing this is, while doing sum for secondary diagonal elements, we will not add it again.

So, in simple steps: declare a sum variable,declare another variable to contain matrixLength-1 then inside a for loop of iterations less than matrix length , calculate sum+=mat[i][i] then assign it 0, then sum+=mat[i][j] and then decrease j by 1. In the end return sum. Code is given below:

1588. Sum of All Odd Length Subarrays

This question can be solved with two approaches Approach1: Brute force approach public int sumOddLengthSubarrays(int[] arr) { int sum = 0; int arrSize = 1; // repeating loop odd number of times. while (arrSize <= arr.length) { for (int i = 0; i < arr.length - (arrSize - 1); i++) { //Below loop gives us sum of array elements. for (int j = i; j < i + arrSize; j++) { sum += arr[j]; } } arrSize = arrSize + 2; } return sum; } Approach2: O(N) complexity The basic idea behind the approach is to compute the sum, but not in the order intended. For example, take a look at the array [1, 2, 3, 4]. The subarrays are [1] [2] [3] [4] [1, 2] [2, 3] [3, 4] [1, 2, 3] [2, 3, 4] [1, 2, 3, 4] Notice how many copies of each element there are. There are four 1's, six 2's, six 3's, and four 4's. If we could efficiently compute how many copies of each element there are across all the different subarrays, we could directly compute the sum by multiply each element in the array by the number of times it appears across all the subarrays and then adding them up. You can find a bunch of interesting patterns with how many times each number shows up. Here's one useful one. We can count the number of subarrays that overlap a particular element at index i by counting those subarrays and focusing on the index at which those subarrays start. The first element of the array will appear in n different subarrays – each of them starts at the first position. The second element of the array will appear in n1 subarrays that begin at its position, plus n1 subarrays from the previous position (there are n total intervals, of which one has length one and therefore won't reach the second element). The third element of the array will appear in n2 subarrays that begin in its position, plus n2 subarrays beginning at the first element (all n arrays, minus the one of length two and the one of length one) and n2 subarrays beginning at the second element (all n1 of them except for the one of length one). More generally, the ith element will open n – i new intervals (one for each length stretching out to the end) and, for each preceding element, will overlap n – i of the intervals starting there. This means that the total number of intervals overlapping element i is given by (n – i)i + (n – i) = (n – i)(i + 1). class Solution { public int sumOddLengthSubarrays(int[] arr) { int sum = 0; for(int i = 0; i<arr.length; i++) { int eachElementContribution = (int)Math.ceil(((i+1)*(arr.length-i))/2.0); sum += eachElementContribution * arr[i]; } return sum; } }