**HAPPY CODING - 150**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1.Maximum Subarray

Easy

Given an integer array nums, find the subarray with the largest sum, and return its sum.

Example 1:

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]

Output: 6

Explanation: The subarray [4,-1,2,1] has the largest sum 6.

Example 2:

Input: nums = [1]

Output: 1

Explanation: The subarray [1] has the largest sum 1.

Example 3:

Input: nums = [5,4,-1,7,8]

Output: 23

Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.

Constraints:

1 <= nums.length <= 105

-104 <= nums[i] <= 104

Follow up: If you have figured out the O(n) solution, try coding another solution using the divide and conquer approach, which is more subtle.

Solution:

class Solution {

public int maxSubArray(int[] nums) {

int max = nums[0];

int currSum = nums[0];

for (int i = 1; i < nums.length; i++) { // 1 to (n-1)

if (currSum < 0) {

currSum = 0;

}

currSum = currSum + nums[i];

if (currSum > max) {

max = currSum;

}

}

return max;

}

}

2.Best Time to Buy and Sell Stock

Easy

You are given an array prices where prices[i] is the price of a given stock on the ith day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

Example 1:

Input: prices = [7,1,5,3,6,4]

Output: 5

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

Input: prices = [7,6,4,3,1]

Output: 0

Explanation: In this case, no transactions are done and the max profit = 0.

Constraints:

1 <= prices.length <= 105

0 <= prices[i] <= 104

Solution:

class Solution {

public int maxProfit(int[] prices) {

int n = prices.length;

int buyPrice = Integer.MAX\_VALUE;

int maxProfit = 0;

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

if (buyPrice < prices[i]) {

int profit = prices[i] - buyPrice;

maxProfit = Math.max(maxProfit, profit);

} else {

buyPrice = prices[i];

}

}

return maxProfit;

}

}

3.Maximum Product Subarray

Medium

Given an integer array nums, find a subarray that has the largest product, and return the product.

The test cases are generated so that the answer will fit in a 32-bit integer.

Example 1:

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

Output: 6

Explanation: [2,3] has the largest product 6.

Example 2:

Input: nums = [-2,0,-1]

Output: 0

Explanation: The result cannot be 2, because [-2,-1] is not a subarray.

Constraints:

1 <= nums.length <= 2 \* 104

-10 <= nums[i] <= 10

The product of any subarray of nums is guaranteed to fit in a 32-bit integer.

Solution:

class Solution {

public int maxProduct(int[] nums) {

if (nums == null || nums.length == 0) {

return 0;

}

int maxProduct = nums[0];

int minProduct = nums[0];

int result = nums[0];

for (int i = 1; i < nums.length; i++) {

if (nums[i] >= 0) {

maxProduct = Math.max(nums[i], maxProduct \* nums[i]);

minProduct = Math.min(nums[i], minProduct \* nums[i]);

} else {

int temp = maxProduct;

maxProduct = Math.max(nums[i], minProduct \* nums[i]);

minProduct = Math.min(nums[i], temp \* nums[i]);

}

result = Math.max(result, maxProduct);

}

return result;

}

}

4. Product of Array Except Self

Medium

Given an integer array nums, return an array answer such that answer[i] is equal to the product of all the elements of nums except nums[i].

The product of any prefix or suffix of nums is guaranteed to fit in a 32-bit integer.

You must write an algorithm that runs in O(n) time and without using the division operation.

Example 1:

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

Output: [24,12,8,6]

Example 2:

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

Output: [0,0,9,0,0]

Constraints:

2 <= nums.length <= 105

-30 <= nums[i] <= 30

The input is generated such that answer[i] is guaranteed to fit in a 32-bit integer.

Follow up: Can you solve the problem in O(1) extra space complexity? (The output array does not count as extra space for space complexity analysis.)

Solution:

class Solution {

public int[] productExceptSelf(int[] nums) {

int n = nums.length;

int right[] = new int[n];

int left[] = new int[n];

int output[] = new int[n];

left[0] = 1;

right[n - 1] = 1;

for (int i = 1; i < n; i++) {

left[i] = left[i - 1] \* nums[i - 1];

}

for (int i = n - 2; i >= 0; i--) {

right[i] = right[i + 1] \* nums[i + 1];

}

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

output[i] = left[i] \* right[i];

}

return output;

}

}

5.Rotate Array

Medium

Given an integer array nums, rotate the array to the right by k steps, where k is non-negative.

Example 1:

Input: nums = [1,2,3,4,5,6,7], k = 3

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

Explanation:

rotate 1 steps to the right: [7,1,2,3,4,5,6]

rotate 2 steps to the right: [6,7,1,2,3,4,5]

rotate 3 steps to the right: [5,6,7,1,2,3,4]

Example 2:

Input: nums = [-1,-100,3,99], k = 2

Output: [3,99,-1,-100]

Explanation:

rotate 1 steps to the right: [99,-1,-100,3]

rotate 2 steps to the right: [3,99,-1,-100]

Constraints:

1 <= nums.length <= 105

-231 <= nums[i] <= 231 - 1

0 <= k <= 105

Follow up:

Try to come up with as many solutions as you can. There are at least three different ways to solve this problem.

Could you do it in-place with O(1) extra space?

Solution:

class Solution {

public void rotate(int[] nums, int k) {

int n = nums.length;

k = k % n;

reverse(nums, 0, n - 1);

reverse(nums, 0, k - 1);

reverse(nums, k, n-1);

}

// helper function

void reverse(int arr[], int start, int end) {

int i = start;

int j = end;

while (i < j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

i++;

j--;

}

}

}

6. Max Consecutive Ones

Easy

Given a binary array nums, return the maximum number of consecutive 1's in the array.

Example 1:

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

Output: 3

Explanation: The first two digits or the last three digits are consecutive 1s. The maximum number of consecutive 1s is 3.

Example 2:

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

Output: 2

Constraints:

1 <= nums.length <= 105

nums[i] is either 0 or 1

Solution:

class Solution {

public int findMaxConsecutiveOnes(int[] nums) {

int ans = 0;

int count = 0;

for (int i = 0; i < nums.length; i++) {

if (nums[i] == 0) {

count = 0;

} else {

count++;

}

if (count > ans) {

ans = count;

}

}

return ans;

}

}

7.Valid Sudoku

Medium

Determine if a 9 x 9 Sudoku board is valid. Only the filled cells need to be validated according to the following rules:

Each row must contain the digits 1-9 without repetition.

Each column must contain the digits 1-9 without repetition.

Each of the nine 3 x 3 sub-boxes of the grid must contain the digits 1-9 without repetition.

Note:

A Sudoku board (partially filled) could be valid but is not necessarily solvable.

Only the filled cells need to be validated according to the mentioned rules.

Example 1:

Input: board =

[["5","3",".",".","7",".",".",".","."]

,["6",".",".","1","9","5",".",".","."]

,[".","9","8",".",".",".",".","6","."]

,["8",".",".",".","6",".",".",".","3"]

,["4",".",".","8",".","3",".",".","1"]

,["7",".",".",".","2",".",".",".","6"]

,[".","6",".",".",".",".","2","8","."]

,[".",".",".","4","1","9",".",".","5"]

,[".",".",".",".","8",".",".","7","9"]]

Output: true

Example 2:

Input: board =

[["8","3",".",".","7",".",".",".","."]

,["6",".",".","1","9","5",".",".","."]

,[".","9","8",".",".",".",".","6","."]

,["8",".",".",".","6",".",".",".","3"]

,["4",".",".","8",".","3",".",".","1"]

,["7",".",".",".","2",".",".",".","6"]

,[".","6",".",".",".",".","2","8","."]

,[".",".",".","4","1","9",".",".","5"]

,[".",".",".",".","8",".",".","7","9"]]

Output: false

Explanation: Same as Example 1, except with the 5 in the top left corner being modified to 8. Since there are two 8's in the top left 3x3 sub-box, it is invalid.

Constraints:

board.length == 9

board[i].length == 9

board[i][j] is a digit 1-9 or '.'.

Solution:

class Solution {

public boolean isValidSudoku(char[][] board) {

//here i created hashset for 2-D

Set<Character>[] rowSet = new HashSet[9];

Set<Character>[] colSet = new HashSet[9];

Set<Character>[] gridSet = new HashSet[9];

//initialization

for(int i=0; i<9; i++) {

rowSet[i] = new HashSet<>();

colSet[i] = new HashSet<>();

gridSet[i] = new HashSet<>();

}

//checking

for(int i=0; i<9; i++) {

for(int j=0; j<9; j++) {

int gridNo = (i/3)\*3 + (j/3);

if(board[i][j]!='.') {

boolean isPresentInRow = rowSet[i].contains(board[i][j]);

boolean isPresentInCol = colSet[j].contains(board[i][j]);

boolean isPresentInGrid = gridSet[gridNo].contains(board[i][j]);

if(isPresentInRow || isPresentInCol || isPresentInGrid) {

return false;

}

rowSet[i].add(board[i][j]);

colSet[j].add(board[i][j]);

gridSet[gridNo].add(board[i][j]);

}

}

}

return true;

}

}

8. K Closest Points to Origin

Medium

Given an array of points where points[i] = [xi, yi] represents a point on the X-Y plane and an integer k, return the k closest points to the origin (0, 0).

The distance between two points on the X-Y plane is the Euclidean distance (i.e., √(x1 - x2)2 + (y1 - y2)2).

You may return the answer in any order. The answer is guaranteed to be unique (except for the order that it is in).

Example 1:

Input: points = [[1,3],[-2,2]], k = 1

Output: [[-2,2]]

Explanation:

The distance between (1, 3) and the origin is sqrt(10).

The distance between (-2, 2) and the origin is sqrt(8).

Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.

We only want the closest k = 1 points from the origin, so the answer is just [[-2,2]].

Example 2:

Input: points = [[3,3],[5,-1],[-2,4]], k = 2

Output: [[3,3],[-2,4]]

Explanation: The answer [[-2,4],[3,3]] would also be accepted.

Constraints:

1 <= k <= points.length <= 104

-104 <= xi, yi <= 104

Solution:

import java.util.Random;

class Solution {

public int[][] kClosest(int[][] points, int k) {

int left = 0, right = points.length - 1;

while (left <= right) {

int pivotIndex = partition(points, left, right);

if (pivotIndex == k - 1) {

break;

} else if (pivotIndex < k - 1) {

left = pivotIndex + 1;

} else {

right = pivotIndex - 1;

}

}

int[][] result = new int[k][2];

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

result[i] = points[i];

}

return result;

}

private int partition(int[][] points, int left, int right) {

int pivotIndex = left + new Random().nextInt(right - left + 1);

int[] pivotPoint = points[pivotIndex];

swap(points, pivotIndex, right);

int i = left;

for (int j = left; j < right; j++) {

if (compare(points[j], pivotPoint) < 0) {

swap(points, i++, j);

}

}

swap(points, i, right);

return i;

}

private int compare(int[] point1, int[] point2) {

return (point1[0] \* point1[0] + point1[1] \* point1[1]) -

(point2[0] \* point2[0] + point2[1] \* point2[1]);

}

private void swap(int[][] points, int i, int j) { //swap function

int[] temp = points[i];

points[i] = points[j];

points[j] = temp;

}

}