**LEETCODE**

Q) 657 : There is a robot starting at the position (0, 0), the origin, on a 2D plane. Given a sequence of its moves, judge if this robot **ends up at**(0, 0) after it completes its moves.

You are given a string moves that represents the move sequence of the robot where moves[i] represents its ith move. Valid moves are 'R' (right), 'L' (left), 'U' (up), and 'D' (down).

Return true*if the robot returns to the origin after it finishes all of its moves, or*false*otherwise*.

**Note**: The way that the robot is "facing" is irrelevant. 'R' will always make the robot move to the right once, 'L' will always make it move left, etc. Also, assume that the magnitude of the robot's movement is the same for each move.

**Example 1:**

**Input:** moves = "UD"

**Output:** true

**Explanation**: The robot moves up once, and then down once. All moves have the same magnitude, so it ended up at the origin where it started. Therefore, we return true.

**CODE:--**

class Solution {

    public boolean judgeCircle(String moves) {

        int x = 0, y = 0;

        for(char move: moves.toCharArray()){

            if(move == 'U') y++;

            else if(move == 'D') y--;

            else if(move == 'L') x--;

            else if(move == 'R') x++;

        }

        return x == 0 && y == 0;

    }

}

Q) 3195 --You are given a 2D **binary** array grid. Find a rectangle with horizontal and vertical sides with the**smallest** area, such that all the 1's in grid lie inside this rectangle.

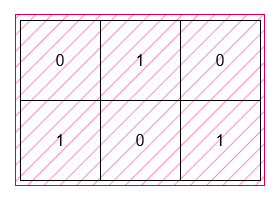
Return the **minimum** possible area of the rectangle.

**Example 1:**

**Input:** grid = [[0,1,0],[1,0,1]]

**Output:** 6

**Explanation:**



The smallest rectangle has a height of 2 and a width of 3, so it has an area of 2 \* 3 = 6.

CODE:--

class Solution {

    public int minimumArea(int[][] grid) {

        int n = grid.length;

        int m = grid[0].length;

        int min\_i = n;

        int max\_i = 0;

        int min\_j = m;

        int max\_j = 0;

        for (int i = 0; i < n; i++) {

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

                if (grid[i][j] == 1) {

                    min\_i = Math.min(min\_i, i);

                    max\_i = Math.max(max\_i, i);

                    min\_j = Math.min(min\_j, j);

                    max\_j = Math.max(max\_j, j);

                }

            }

        }

        return (max\_i - min\_i + 1) \* (max\_j - min\_j + 1);

    }

}

Q 26) -- Given an integer array nums sorted in **non-decreasing order**, remove the duplicates [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**. Then return *the number of unique elements in*nums.

Consider the number of unique elements of nums to be k, to get accepted, you need to do the following things:

* Change the array nums such that the first k elements of nums contain the unique elements in the order they were present in nums initially. The remaining elements of nums are not important as well as the size of nums.
* Return k.

**Custom Judge:**

The judge will test your solution with the following code:

int[] nums = [...]; // Input array

int[] expectedNums = [...]; // The expected answer with correct length

int k = removeDuplicates(nums); // Calls your implementation

assert k == expectedNums.length;

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

assert nums[i] == expectedNums[i];

}

If all assertions pass, then your solution will be **accepted**.

**Example 1:**

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

**Output:** 2, nums = [1,2,\_]

**Explanation:** Your function should return k = 2, with the first two elements of nums being 1 and 2 respectively.

It does not matter what you leave beyond the returned k (hence they are underscores).

CODE:--

class Solution {

    public int removeDuplicates(int[] nums) {

        int i=0;

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

            if(nums[i]!=nums[j]){

                i++;

                nums[i]=nums[j];

            }

        }

        return i+1;

    }

}

Q1) TWO SUM

Given an array of integers nums and an integer target, return *indices of the two numbers such that they add up to target*.

You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

**Example 1:**

**Input:** nums = [2,7,11,15], target = 9

**Output:** [0,1]

**Explanation:** Because nums[0] + nums[1] == 9, we return [0, 1].

**Example 2:**

**Input:** nums = [3,2,4], target = 6

**Output:** [1,2]

CODE:--

class Solution {

    public int[] twoSum(int[] nums, int target) {

        int n = nums.length;

        int[] ans = new int[2];

        ans[0] = ans[1] = -1;

        HashMap<Integer, Integer> mpp = new HashMap<>();

        for (int i = 0; i < n; i++) {

            int num = nums[i];

            int moreNeeded = target - num;

            if (mpp.containsKey(moreNeeded)) {

                ans[0] = mpp.get(moreNeeded);

                ans[1] = i;

                return ans;

            }

            mpp.put(nums[i], i);

        }

        return ans;

    }

}

Q15) 3 SUM

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

Notice that the solution set must not contain duplicate triplets.

**Example 1:**

**Input:** nums = [-1,0,1,2,-1,-4]

**Output:** [[-1,-1,2],[-1,0,1]]

**Explanation:**

nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.

nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.

nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.

The distinct triplets are [-1,0,1] and [-1,-1,2].

Notice that the order of the output and the order of the triplets does not matter.

**Example 2:**

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

**Output:** []

**Explanation:** The only possible triplet does not sum up to 0.

CODE:--

class Solution {

    public List<List<Integer>> threeSum(int[] nums) {

        int target = 0;

        Arrays.sort(nums);

        Set<List<Integer>> s = new HashSet<>();

        List<List<Integer>> output = new ArrayList<>();

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

            int j = i + 1;

            int k = nums.length - 1;

            while (j < k) {

                int sum = nums[i] + nums[j] + nums[k];

                if (sum == target) {

                    s.add(Arrays.asList(nums[i], nums[j], nums[k]));

                    j++;

                    k--;

                } else if (sum < target) {

                    j++;

                } else {

                    k--;

                }

            }

        }

        output.addAll(s);

        return output;

    }

}

Q) 164--- MAXIMUM GAPS

Given an integer array nums, return *the maximum difference between two successive elements in its sorted form*. If the array contains less than two elements, return 0.

You must write an algorithm that runs in linear time and uses linear extra space.

**Example 1:**

**Input:** nums = [3,6,9,1]

**Output:** 3

**Explanation:** The sorted form of the array is [1,3,6,9], either (3,6) or (6,9) has the maximum difference 3.

**Example 2:**

**Input:** nums = [10]

**Output:** 0

**Explanation:** The array contains less than 2 elements, therefore return 0.

CODE:--

class Solution {

    public int maximumGap(int[] nums) {

        if (nums == null || nums.length < 2){

            return 0;

        }

        Arrays.sort(nums);

        int maxDiff = 0;

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

            int diff = nums[i] - nums[i - 1];

            if (diff > maxDiff) {

                maxDiff = diff;

            }

        }

        return maxDiff;

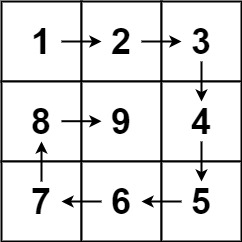
    }

}

Q 159) –SPIRAL MATRIX-II

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]]

**CODE:--**

class Solution {

    public int[][] generateMatrix(int n) {

        int[][] result = new int[n][n];

        int cnt = 1;

        int dir[][] = { { 0, 1 }, { 1, 0 }, { 0, -1 }, { -1, 0 } };

        int d = 0;

        int row = 0;

        int col = 0;

        while (cnt <= n \* n) {

            result[row][col] = cnt++;

            int r = Math.floorMod(row + dir[d][0], n);

            int c = Math.floorMod(col + dir[d][1], n);

            // change direction if next cell is non zero

            if (result[r][c] != 0) d = (d + 1) % 4;

            row += dir[d][0];

            col += dir[d][1];

        }

        return result;

    }

}

Q80) Given an integer array nums sorted in **non-decreasing order**, remove some duplicates [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) such that each unique element appears **at most twice**. The **relative order** of the elements should be kept the **same**.

Since it is impossible to change the length of the array in some languages, you must instead have the result be placed in the **first part** of the array nums. More formally, if there are k elements after removing the duplicates, then the first k elements of nums should hold the final result. It does not matter what you leave beyond the first k elements.

Return k*after placing the final result in the first*k*slots of*nums.

Do **not** allocate extra space for another array. You must do this by **modifying the input array**[**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

**Custom Judge:**

The judge will test your solution with the following code:

int[] nums = [...]; // Input array

int[] expectedNums = [...]; // The expected answer with correct length

int k = removeDuplicates(nums); // Calls your implementation

assert k == expectedNums.length;

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

assert nums[i] == expectedNums[i];

}

If all assertions pass, then your solution will be **accepted**.

**Example 1:**

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

**Output:** 5, nums = [1,1,2,2,3,\_]

**Explanation:** Your function should return k = 5, with the first five elements of nums being 1, 1, 2, 2 and 3 respectively.

It does not matter what you leave beyond the returned k (hence they are underscores).

CODE:--

class Solution {

    public int removeDuplicates(int[] nums) {

        int writeIndex = 1;

        int duplicateCount = 0;

        int prevElement = nums[0];

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

            if (nums[i] != prevElement) {

                duplicateCount = 0;

            } else {

                duplicateCount++;

            }

            if (duplicateCount <= 1) {

                nums[writeIndex++] = nums[i];

                prevElement = nums[i];

            }

        }

        return writeIndex;

    }

}

Q 299) You are playing the [**Bulls and Cows**](https://en.wikipedia.org/wiki/Bulls_and_Cows) game with your friend.

You write down a secret number and ask your friend to guess what the number is. When your friend makes a guess, you provide a hint with the following info:

* The number of "bulls", which are digits in the guess that are in the correct position.
* The number of "cows", which are digits in the guess that are in your secret number but are located in the wrong position. Specifically, the non-bull digits in the guess that could be rearranged such that they become bulls.

Given the secret number secret and your friend's guess guess, return *the hint for your friend's guess*.

The hint should be formatted as "xAyB", where x is the number of bulls and y is the number of cows. Note that both secret and guess may contain duplicate digits.

**Example 1:**

**Input:** secret = "1807", guess = "7810"

**Output:** "1A3B"

**Explanation:** Bulls are connected with a '|' and cows are underlined:

"1807"

|

"7810"

CODE:--

class Solution {

    public String getHint(String secret, String guess) {

        int bulls = 0;

        int cows = 0;

        int[] numbers = new int[10];

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

            int s = Character.getNumericValue(secret.charAt(i));

            int g = Character.getNumericValue(guess.charAt(i));

            if (s == g) bulls++;

            else {

                if (numbers[s] < 0) cows++;

                if (numbers[g] > 0) cows++;

                numbers[s] ++;

                numbers[g] --;

            }

        }

        return bulls + "A" + cows + "B";

        }

}

Q53)-- 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.

CODE:--

class Solution(object):

    def maxSubArray(self, nums):

        max\_sum = current\_sum = nums[0]

        for num in nums[1:]:

            current\_sum = max(num, current\_sum + num)

            max\_sum = max(max\_sum, current\_sum)

        return max\_sum

Q441) A binary watch has 4 LEDs on the top to represent the hours (0-11), and 6 LEDs on the bottom to represent the minutes (0-59). Each LED represents a zero or one, with the least significant bit on the right.

* For example, the below binary watch reads "4:51".

Given an integer turnedOn which represents the number of LEDs that are currently on (ignoring the PM), return *all possible times the watch could represent*. You may return the answer in **any order**.

The hour must not contain a leading zero.

* For example, "01:00" is not valid. It should be "1:00".

The minute must consist of two digits and may contain a leading zero.

* For example, "10:2" is not valid. It should be "10:02".

**Example 1:**

**Input:** turnedOn = 1

**Output:** ["0:01","0:02","0:04","0:08","0:16","0:32","1:00","2:00","4:00","8:00"]

CODE:--

class Solution(object):

    def readBinaryWatch(self, turnedOn):

        if turnedOn < 0 or turnedOn > 10:

            return []

        result = []

        for h in range(12):

            for m in range(60):

                if bin(h).count('1') + bin(m).count('1') == turnedOn:

                    result.append("{}:{:02d}".format(h, m))

        return result

Q2210) You are given a **0-indexed** integer array nums. An index i is part of a **hill** in nums if the closest non-equal neighbors of i are smaller than nums[i]. Similarly, an index i is part of a **valley** in nums if the closest non-equal neighbors of i are larger than nums[i]. Adjacent indices i and j are part of the **same** hill or valley if nums[i] == nums[j].

Note that for an index to be part of a hill or valley, it must have a non-equal neighbor on **both** the left and right of the index.

Return *the number of hills and valleys in*nums.

**Example 1:**

**Input:** nums = [2,4,1,1,6,5]

**Output:** 3

**Explanation:**

At index 0: There is no non-equal neighbor of 2 on the left, so index 0 is neither a hill nor a valley.

At index 1: The closest non-equal neighbors of 4 are 2 and 1. Since 4 > 2 and 4 > 1, index 1 is a hill.

At index 2: The closest non-equal neighbors of 1 are 4 and 6. Since 1 < 4 and 1 < 6, index 2 is a valley.

At index 3: The closest non-equal neighbors of 1 are 4 and 6. Since 1 < 4 and 1 < 6, index 3 is a valley, but note that it is part of the same valley as index 2.

At index 4: The closest non-equal neighbors of 6 are 1 and 5. Since 6 > 1 and 6 > 5, index 4 is a hill.

At index 5: There is no non-equal neighbor of 5 on the right, so index 5 is neither a hill nor a valley.

There are 3 hills and valleys so we return 3.

CODE:--

class Solution {

    public int countHillValley(int[] nums) {

       List<Integer> clean = new ArrayList<>();

        clean.add(nums[0]);

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

            if (nums[i] != nums[i - 1]) {

                clean.add(nums[i]);

            }

        }

        // Step 2: Count hills and valleys

        int count = 0;

        for (int i = 1; i < clean.size() - 1; i++) {

            int prev = clean.get(i - 1);

            int curr = clean.get(i);

            int next = clean.get(i + 1);

            if ((curr > prev && curr > next) || (curr < prev && curr < next)) {

                count++;

            }

        }

        return count;

    }

}

Q69) Given an array nums of size n, return *the majority element*.

The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

**Example 1:**

**Input:** nums = [3,2,3]

**Output:** 3

CODE:--

class Solution(object):

    def majorityElement(self, nums):

        count = 0

        candidate = None

        for num in nums:

            if count == 0:

                candidate = num

            count += (1 if num == candidate else -1)

        return candidate

Q229) Given an integer array of size n, find all elements that appear more than

 ⌊ n/3 ⌋ times.

**Example 1:**

**Input:** nums = [3,2,3]

**Output:** [3]

**Example 2:**

**Input:** nums = [1]

**Output:** [1]

**Example 3:**

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

**Output:** [1,2]

CODE:- class Solution(object):

    def majorityElement(self, nums):

        if not nums:

            return []

    # Step 1: Find potential candidates

        candidate1 = candidate2 = None

        count1 = count2 = 0

        for num in nums:

            if num == candidate1:

                count1 += 1

            elif num == candidate2:

                count2 += 1

            elif count1 == 0:

                candidate1 = num

                count1 = 1

            elif count2 == 0:

                candidate2 = num

                count2 = 1

            else:

                count1 -= 1

                count2 -= 1

    # Step 2: Verify counts of the candidates

        result = []

        count1 = count2 = 0

        for num in nums:

            if num == candidate1:

                count1 += 1

            elif num == candidate2:

                count2 += 1

        if count1 > len(nums) // 3:

            result.append(candidate1)

        if count2 > len(nums) // 3:

            result.append(candidate2)

        return result

      Q643)-- You are given an integer array nums consisting of n elements, and an integer k.

Find a contiguous subarray whose **length is equal to** k that has the maximum average value and return *this value*. Any answer with a calculation error less than 10-5 will be accepted.

**Example 1:**

**Input:** nums = [1,12,-5,-6,50,3], k = 4

**Output:** 12.75000

**Explanation:** Maximum average is (12 - 5 - 6 + 50) / 4 = 51 / 4 = 12.75

**Example 2:**

**Input:** nums = [5], k = 1

**Output:** 5.00000

CODE:- class Solution(object):

    def findMaxAverage(self, nums, k):

         # Step 1: Calculate the sum of the first 'k' elements

        window\_sum = sum(nums[:k])

        max\_sum = window\_sum

        # Step 2: Slide the window across the array

        for i in range(k, len(nums)):

            window\_sum += nums[i] - nums[i - k]

            max\_sum = max(max\_sum, window\_sum)

        # Step 3: Return the maximum average with float precision

        return max\_sum / k

Q209) Given an array of positive integers nums and a positive integer target, return *the****minimal length****of a subarray whose sum is greater than or equal to* target. If there is no such subarray, return 0 instead.

**Example 1:**

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

**Output:** 2

**Explanation:** The subarray [4,3] has the minimal length under the problem constraint.

**Example 2:**

**Input:** target = 4, nums = [1,4,4]

**Output:** 1

**Example 3:**

**Input:** target = 11, nums = [1,1,1,1,1,1,1,1]

**Output:** 0

CODE:--  class Solution {

    public int minSubArrayLen(int target, int[] nums) {

        int i=0,j=0,n=nums.length,sum=0,min=Integer.MAX\_VALUE;

        while(j<n)

        {

            sum+=nums[j];

            while(sum>=target)

            {

                min=Math.min(min,j-i+1);

                sum-=nums[i];

                i++;

            }

            j++;

        }

        return min==Integer.MAX\_VALUE?0:min;

    }

}

Q1004) Given a binary array nums and an integer k, return *the maximum number of consecutive*1*'s in the array if you can flip at most* k 0's.

**Example 1:**

**Input:** nums = [1,1,1,0,0,0,1,1,1,1,0], k = 2

**Output:** 6

**Explanation:** [1,1,1,0,0,**1**,1,1,1,1,**1**]

Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.

**Example 2:**

**Input:** nums = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], k = 3

**Output:** 10

**Explanation:** [0,0,1,1,**1**,**1**,1,1,1,**1**,1,1,0,0,0,1,1,1,1]

Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.

CODE:--

class Solution {

    public int longestOnes(int[] nums, int k) {

        int i=0,j=0;

        while(j<nums.length){

            if(nums[j]==0){

                k--;

            }

            if(k<0){

                if(nums[i]==0){

                    k++;

                }

                i++;

            }

            j++;

        }

        return j-I;

    }

}

Q3)-- Given a string s, find the length of the **longest** **substring** without duplicate characters.

**Example 1:**

**Input:** s = "abcabcbb"

**Output:** 3

**Explanation:** The answer is "abc", with the length of 3.

**Example 2:**

**Input:** s = "bbbbb"

**Output:** 1

**Explanation:** The answer is "b", with the length of 1.

CODE:--

class Solution {

    public int lengthOfLongestSubstring(String s) {

        Set<Character> set = new HashSet<>();

        int maxLength = 0;

        int left = 0;

        for (int right = 0; right < s.length(); right++) {

            char currentChar = s.charAt(right);

            // Remove characters from the left until we remove the duplicate

            while (set.contains(currentChar)) {

                set.remove(s.charAt(left));

                left++;

            }

            set.add(currentChar);

            maxLength = Math.max(maxLength, right - left + 1);

        }

        return maxLength;

    }

}

Q904) You are visiting a farm that has a single row of fruit trees arranged from left to right. The trees are represented by an integer array fruits where fruits[i] is the **type** of fruit the ith tree produces.

You want to collect as much fruit as possible. However, the owner has some strict rules that you must follow:

* You only have **two** baskets, and each basket can only hold a **single type** of fruit. There is no limit on the amount of fruit each basket can hold.
* Starting from any tree of your choice, you must pick **exactly one fruit** from **every** tree (including the start tree) while moving to the right. The picked fruits must fit in one of your baskets.
* Once you reach a tree with fruit that cannot fit in your baskets, you must stop.

Given the integer array fruits, return *the****maximum****number of fruits you can pick*.

**Example 1:**

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

**Output:** 3

**Explanation:** We can pick from all 3 trees.

**Example 2:**

**Input:** fruits = [0,1,2,2]

**Output:** 3

**Explanation:** We can pick from trees [1,2,2].

If we had started at the first tree, we would only pick from trees [0,1].

CODE:--

class Solution {

    public int totalFruit(int[] fruits) {

        int start = 0,end = 0;

        int n = fruits.length,maxLen = 0;

        Map<Integer,Integer> map = new HashMap<>();

        while(end<n)

        {

            map.put(fruits[end],map.getOrDefault(fruits[end],0)+1);

            while(map.size()>=3)

            {

                map.put(fruits[start],map.get(fruits[start])-1);

                if(map.get(fruits[start]) == 0) map.remove(fruits[start]);

                start++;

            }

            int currLen = end-start+1;

            maxLen = Math.max(maxLen,currLen);

            end++;

        }

        return maxLen;

    }

}