**Sliding window**

**1) Longest Substring Without Repeating Characters (Leetcode - 3)**

Given a string s, find the length of the longest substring without repeating 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.

Example 3:

Input: s = "pwwkew"

Output: 3

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

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

Constraints:

0 <= s.length <= 5 \* 104

s consists of English letters, digits, symbols and spaces.

**Solution :**

class Solution {

    public int lengthOfLongestSubstring(String s) {

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

        int max=0,left=0;

        for(char c:s.toCharArray())

        {

            while(set.contains(c))

            {

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

                left++;

            }

            set.add(c);

            max=Math.max(max,set.size());

        }

        return max;

    }

}

**2) Maximum Points You Can Obtain from Cards (Leetcode - 1423)**

There are several cards arranged in a row, and each card has an associated number of points. The points are given in the integer array cardPoints.

In one step, you can take one card from the beginning or from the end of the row. You have to take exactly k cards.

Your score is the sum of the points of the cards you have taken.

Given the integer array cardPoints and the integer k, return the maximum score you can obtain.

Example 1:

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

Output: 12

Explanation: After the first step, your score will always be 1. However, choosing the rightmost card first will maximize your total score. The optimal strategy is to take the three cards on the right, giving a final score of 1 + 6 + 5 = 12.

Example 2:

Input: cardPoints = [2,2,2], k = 2

Output: 4

Explanation: Regardless of which two cards you take, your score will always be 4.

Example 3:

Input: cardPoints = [9,7,7,9,7,7,9], k = 7

Output: 55

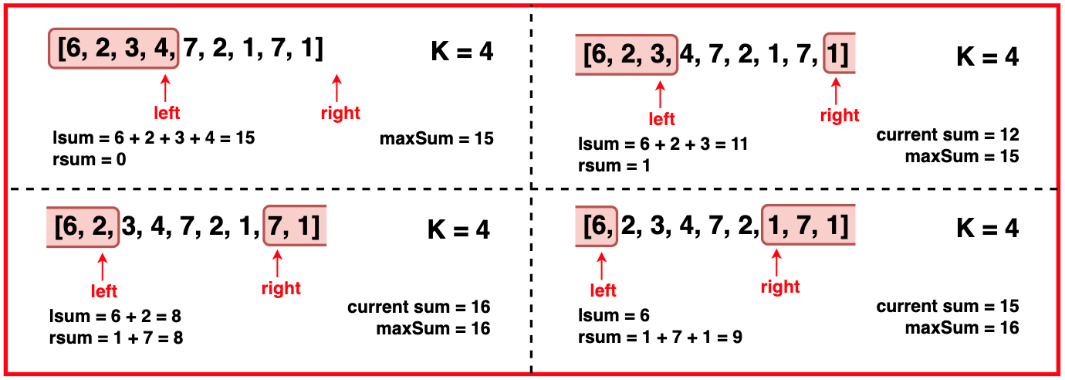
Explanation: You have to take all the cards. Your score is the sum of points of all cards.

Constraints:

1 <= cardPoints.length <= 105

1 <= cardPoints[i] <= 104

1 <= k <= cardPoints.length



**Solution :**

class Solution {

    public int maxScore(int[] cardPoints, int k) {

        int left=k-1;

        int right=cardPoints.length-1;

        int score=0;

        for(int i=0;i<k;i++)

        {

            score+=cardPoints[i];

        }

        int maxScore=score;

        while(k!=0)

        {

            score-=cardPoints[left--];

            score+=cardPoints[right--];

            maxScore=Math.max(score,maxScore);

            k--;

        }

        return maxScore;

    }

}

**3) Max Consecutive Ones (Leetcode - 485)**

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 left=-1;

        int max=0;

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

        {

            if(nums[i]==0)

            {

                left=i;

            }

            max=Math.max(max,i-left);

        }

        return max;

    }

}

**4) Max Consecutive Ones III (Leetcode - 1004)**

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.

Constraints:

1 <= nums.length <= 105

nums[i] is either 0 or 1.

0 <= k <= nums.length

**Solution :**

class Solution {

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

        int temp=k;

        int left=0;

        int max=0;

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

        {

            if(nums[i]==0)

            {

                temp--;

            }

            while(temp<0)

            {

                if(nums[left]==0)

                {

                    temp++;

                }

                left++;

            }

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

        }

        return max;

    }

}

**5) maximum length of consecutive 1s after flipping at most one zero**

**Solution :**

class Solution {

public int findMaxConsecutiveOnes(int[] nums) {

int maxConsecutiveOnes = 0;

int left = 0;

int zid = -1; // Initialize to -1 to track the last zero encountered

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

// If we encounter a zero

if (nums[right] == 0) {

// Move the left pointer to one position after the last zero

if (zid != -1) {

left = zid + 1;

}

zid = right; // Update zid to the current zero's index

}

// Calculate the maximum length of the window

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

}

return maxConsecutiveOnes;

}

}

**6) Longest Substring with At Most K Distinct Characters (coding ninjas)**

You are given a string 'str' and an integer ‘K’. Your task is to find the length of the largest substring with at most ‘K’ distinct characters.

For example:

You are given ‘str’ = ‘abbbbbbc’ and ‘K’ = 2, then the substrings that can be formed are [‘abbbbbb’, ‘bbbbbbc’]. Hence the answer is 7.

Detailed explanation ( Input/output format, Notes, Images )

Constraints:

1 <= T <= 10

1 <= K <= 26

1 <= |str| <= 10^6

The string str will contain only lowercase alphabets.

Time Limit: 1 sec

Note:

You do not need to print anything. It has already been taken care of. Just implement the function.

Sample Input 1:

2

2

abbbbbbc

3

abcddefg

Sample Output 1:

7

4

Explanation:

For the first test case, ‘str’ = ‘abbbbbbc’ and ‘K’ = 2, then the substrings that can be formed are [‘abbbbbb’, ‘bbbbbbc’]. Hence the answer is 7.

For the second test case, ‘str’ = ‘abcddefg’ and ‘K’ = 3, then the substrings that can be formed is [‘cdde’, ‘ddef’]. Hence the answer is 4.

Sample Input 2:

2

3

aaaaaaaa

1

abcefg

Sample Output 2:

8

1

**Solution :**

public static int kDistinctChars(int k, String str) {

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

        int left=0;

        int max=0;

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

        {

            char ch=str.charAt(i);

            map.put(ch,map.getOrDefault(ch, 0)+1);

            while(map.size()>k)

            {

                char c = str.charAt(left);

                map.put(c,map.get(c)-1);

                if(map.get(str.charAt(left))==0)

                {

                    map.remove(c);

                }

                left++;

            }

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

        }

        return max;

    }

**7) Number of Substrings Containing All Three Characters (Leetcode - 1358)**

Given a string s consisting only of characters a, b and c.

Return the number of substrings containing at least one occurrence of all these characters a, b and c.

Example 1:

Input: s = "abcabc"

Output: 10

Explanation: The substrings containing at least one occurrence of the characters a, b and c are "abc", "abca", "abcab", "abcabc", "bca", "bcab", "bcabc", "cab", "cabc" and "abc" (again).

Example 2:

Input: s = "aaacb"

Output: 3

Explanation: The substrings containing at least one occurrence of the characters a, b and c are "aaacb", "aacb" and "acb".

Example 3:

Input: s = "abc"

Output: 1

Constraints:

3 <= s.length <= 5 x 10^4

s only consists of a, b or c characters

**Solution :**

class Solution {

    public int numberOfSubstrings(String s) {

        int count=0;

        int hash[] = new int[3];

        Arrays.fill(hash,-1);

        for(int i=0;i<s.length();i++)

        {

            int idx=s.charAt(i)-'a';

            hash[idx]=i;

            if(hash[0]!=-1 && hash[1]!=-1 && hash[2]!=-1)

            {

                count+=1+Math.min(Math.min(hash[0],hash[1]),hash[2]);

            }

        }

        return count;

}

}

**8) Longest Repeating Character Replacement(Leetcode - 424)**

You are given a string s and an integer k. You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most k times.

Return the length of the longest substring containing the same letter you can get after performing the above operations.

Example 1:

Input: s = "ABAB", k = 2

Output: 4

Explanation: Replace the two 'A's with two 'B's or vice versa.

Example 2:

Input: s = "AABABBA", k = 1

Output: 4

Explanation: Replace the one 'A' in the middle with 'B' and form "AABBBBA".

The substring "BBBB" has the longest repeating letters, which is 4.

There may exists other ways to achieve this answer too.

Constraints:

1 <= s.length <= 105

s consists of only uppercase English letters.

0 <= k <= s.length

**Step:**

**WindowLegth-MaxFreq it is no of replacements**

**Solution :**

class Solution {

    public int characterReplacement(String s, int k) {

        int left = 0;

        int maxLen = 0;

        int maxFreq = 0;

        int[] hash = new int[26];

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

            int idx = s.charAt(right) - 'A';

            hash[idx]++;

            maxFreq = Math.max(maxFreq, hash[idx]);

            // If the current window size minus the frequency of the most common character exceeds k, shrink the window

            if ((right - left + 1) - maxFreq > k) {

                hash[s.charAt(left) - 'A']--;

                left++;

            }

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

        }

        return maxLen;

    }

}

**9) Binary Subarrays With Sum (Leetcode - 930)**

Given a binary array nums and an integer goal, return the number of non-empty subarrays with a sum goal.

A subarray is a contiguous part of the array.

Example 1:

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

Output: 4

Explanation: The 4 subarrays are bolded and underlined below:

[1,0,1,0,1]

[1,0,1,0,1]

[1,0,1,0,1]

[1,0,1,0,1]

Example 2:

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

Output: 15

Constraints:

1 <= nums.length <= 3 \* 104

nums[i] is either 0 or 1.

0 <= goal <= nums.length

**Solution :**

**//It can also solved using count subarrays sum equal to k i.e sum equal to goal**

**//but in this approach space is o(1)**

class Solution {

//this to find no of subarrays less than or equal to goal

    public int lessEqualGoal(int[] nums,int goal)

    {

        if(goal<0) return 0;

        int left=0,count=0,sum=0;

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

        {

            sum+=nums[right];

            while(sum>goal)

            {

                sum=sum-nums[left];

                left++;

            }

            count+=(right-left+1);

        }

        return count;

    }

    public int numSubarraysWithSum(int[] nums, int goal) {

//no of subarrays less than or equal to goal – no of subarrays less than or equal to goal-1

        return lessEqualGoal(nums,goal)-lessEqualGoal(nums,goal-1);

    }

}

**10) Count Number of Nice Subarrays (Leetcode - 1248)**

Given an array of integers nums and an integer k. A continuous subarray is called nice if there are k odd numbers on it.

Return the number of nice sub-arrays.

Example 1:

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

Output: 2

Explanation: The only sub-arrays with 3 odd numbers are [1,1,2,1] and [1,2,1,1].

Example 2:

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

Output: 0

Explanation: There are no odd numbers in the array.

Example 3:

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

Output: 16

Constraints:

1 <= nums.length <= 50000

1 <= nums[i] <= 10^5

1 <= k <= nums.length

**Solution :** ( same as above question )

//subarray sum equal to k

class Solution {

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

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

        map.put(0,1);

        int ans=0;

        int sum=0;

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

        {

            sum=sum+nums[i]%2;

            ans+=map.getOrDefault(sum-k,0);

            map.put(sum,map.getOrDefault(sum,0)+1);

        }

        return ans;

    }

}

**Solution 2:(previous question)**

**//same as previous convert odd nums to 1 and even to 0**

class Solution {

//no of subarrays atmost goal

    public int lessEqualGoal(int[] nums,int goal)

    {

        if(goal<0) return 0;

        int left=0,count=0,sum=0;

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

        {

            sum+=nums[right]%2;

            while(sum>goal)

            {

                sum=sum-nums[left]%2;

                left++;

            }

            count+=(right-left+1);

        }

        return count;

    }

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

        return lessEqualGoal(nums,k)-lessEqualGoal(nums,k-1);

    }

}

**11) Subarrays with K Different Integers (Leetcode - 992)**

Given an integer array nums and an integer k, return the number of good subarrays of nums.

A good array is an array where the number of different integers in that array is exactly k.

For example, [1,2,3,1,2] has 3 different integers: 1, 2, and 3.

A subarray is a contiguous part of an array.

Example 1:

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

Output: 7

Explanation: Subarrays formed with exactly 2 different integers: [1,2], [2,1], [1,2], [2,3], [1,2,1], [2,1,2], [1,2,1,2]

Example 2:

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

Output: 3

Explanation: Subarrays formed with exactly 3 different integers: [1,2,1,3], [2,1,3], [1,3,4].

Constraints:

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

1 <= nums[i], k <= nums.length

**Solution :**

class Solution {

    public int atmostK(int[] nums,int k)

    {

        int left=0;

        int count=0;

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

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

        {

            map.put(nums[right],map.getOrDefault(nums[right],0)+1);

            while(map.size()>k)

            {

                map.put(nums[left],map.getOrDefault(nums[left],0)-1);

                if(map.get(nums[left])==0)

                {

                    map.remove(nums[left]);

                }

                left++;

            }

            count+=right-left+1;

        }

        return count;

    }

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

       return atmostK(nums,k)-atmostK(nums,k-1);

    }

}

**12) Minimum Window Substring (Leetcode - 76)**

Given two strings s and t of lengths m and n respectively, return the minimum window

substring

of s such that every character in t (including duplicates) is included in the window. If there is no such substring, return the empty string "".

The testcases will be generated such that the answer is unique.

Example 1:

Input: s = "ADOBECODEBANC", t = "ABC"

Output: "BANC"

Explanation: The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t.

Example 2:

Input: s = "a", t = "a"

Output: "a"

Explanation: The entire string s is the minimum window.

Example 3:

Input: s = "a", t = "aa"

Output: ""

Explanation: Both 'a's from t must be included in the window.

Since the largest window of s only has one 'a', return empty string.

Constraints:

m == s.length

n == t.length

1 <= m, n <= 105

s and t consist of uppercase and lowercase English letters.

**Solution :**

class Solution {

    public String minWindow(String s, String t) {

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

        int sInd=-1,minLen=Integer.MAX\_VALUE,left=0,count=0;

        for(int i=0;i<t.length();i++)

        {

            map.put(t.charAt(i),map.getOrDefault(t.charAt(i),0)+1);

        }

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

        {

            if(map.getOrDefault(s.charAt(right),0)>0)

            {

                count++;

            }

            map.put(s.charAt(right),map.getOrDefault(s.charAt(right),0)-1);

            while(count==t.length())

            {

                if(right-left+1<minLen)

                {

                    minLen=right-left+1;

                    sInd=left;

                }

                map.put(s.charAt(left),map.getOrDefault(s.charAt(left),0)+1);

                if(map.getOrDefault(s.charAt(left),0)>0)

                {

                    count--;

                }

                left++;

            }

        }

        return (sInd==-1)?"":s.substring(sInd,sInd+minLen);

    }

}

**13) Minimum Size Subarray Sum (Leetcode - 209)**

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

Constraints:

1 <= target <= 109

1 <= nums.length <= 105

1 <= nums[i] <= 104

**Solution :**

class Solution {

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

        int left=0,len=Integer.MAX\_VALUE;

        int sum=0;

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

        {

            sum+=nums[right];

            while(sum>=target)

            {

                if(len>right-left+1)

                {

                    len=right-left+1;

                }

                sum-=nums[left];

                left++;

            }

        }

        return (len==Integer.MAX\_VALUE)?0:len;

    }

}

**14) Subarray Product Less Than K (Leetcode - 713)**

Given an array of integers nums and an integer k, return the number of contiguous subarrays where the product of all the elements in the subarray is strictly less than k.

Example 1:

Input: nums = [10,5,2,6], k = 100

Output: 8

Explanation: The 8 subarrays that have product less than 100 are:

[10], [5], [2], [6], [10, 5], [5, 2], [2, 6], [5, 2, 6]

Note that [10, 5, 2] is not included as the product of 100 is not strictly less than k.

Example 2:

Input: nums = [1,2,3], k = 0

Output: 0

Constraints:

1 <= nums.length <= 3 \* 104

1 <= nums[i] <= 1000

0 <= k <= 106

**Solutions:**

class Solution {

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

// Handle edge cases where k is 0 or 1 (no subarrays possible)

        if(k<=1) return 0;

        int count = 0;

        int left = 0;

        long prod = 1;

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

        {

            prod=prod\*nums[right];

            while(prod>=k)

            {

                prod=prod/nums[left];

                left++;

            }

            count += right - left+1;

        }

        return count;

    }

}

**Two pointers**

**1) 3Sum (Leetcode - 15)**

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.

Example 3:

Input: nums = [0,0,0]

Output: [[0,0,0]]

Explanation: The only possible triplet sums up to 0.

Constraints:

3 <= nums.length <= 3000

-105 <= nums[i] <= 105

**Solution :**

class Solution {

    /\*

    // 1st Approach: Brute Force Approach(Using 3 Loops) - Time Complexity = O(n^3), Space Complexity = O(n^2)

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

        // Create a set to store unique triplets that sum to zero.

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

        // Iterate over the array with the first index i.

        for (int i = 0; i < nums.length - 2; i++) {

            // Iterate over the array with the second index j, starting from i+1.

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

                // Iterate over the array with the third index k, starting from j+1.

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

                    // Check if the sum of the elements at indices i, j, and k is zero.

                    if (nums[i] + nums[j] + nums[k] == 0) {

                        // Create a list containing the triplet.

                        List<Integer> triplet = Arrays.asList(nums[i], nums[j], nums[k]);

                        // Sort the triplet to ensure uniqueness (i.e., avoid [a, b, c] and [c, b, a] being considered different).

                        triplet.sort(null);

                        // Add the sorted triplet to the set, ensuring no duplicates are added.

                        set.add(triplet);

                    }

                }

            }

        }

        // Convert the set of unique triplets into a list.

        List<List<Integer>> ans = new ArrayList<>(set);

        // Return the list of unique triplets.

        return ans;

    }

**Approach 2: (USING Two sum) (TC : o(n\*n))**

class Solution {

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

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

        for(int i=0;i<nums.length-2;i++)

        {

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

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

            {

                int c = -(nums[i]+nums[j]); //c=-(a+b)

                if(set.contains(c))

                {

                    List<Integer> l = Arrays.asList(nums[i],nums[j],c);

                    l.sort(null);

                    res.add(l);

                }

                set.add(nums[j]);

            }

        }

        return new ArrayList<>(res);

    }

}

**Approach 3: (Two sum approach – sorting and two pointers)**

class Solution {

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

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

        Arrays.sort(nums);

        for(int i=0;i<nums.length-2;i++)

        {

            if (i > 0 && nums[i] == nums[i - 1]) continue;

            int j=i+1,k=nums.length-1;

            int sum=-nums[i]; //(b+c)=-a we need to find (b+c)

            while(j<k)

            {

                if(nums[j]+nums[k]==sum)

                {

                    List<Integer> l = Arrays.asList(nums[i],nums[j],nums[k]);

                    res.add(l);

                    while(j<k && nums[j]==nums[j+1])

                    {

                        j++;

                    }

                    while(j<k && nums[k]==nums[k-1] )

                    {

                        k--;

                    }

                    j++;

                    k--;

                }

                else if(nums[j]+nums[k]<sum)

                {

                    j++;

                }

                else

                {

                    k--;

                }

            }

        }

        return res;

    }

}

**2) Trapping Rain Water (Leetcode - 42)**

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.



Example 1:

Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]

Output: 6

Explanation: The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

Example 2:

Input: height = [4,2,0,3,2,5]

Output: 9

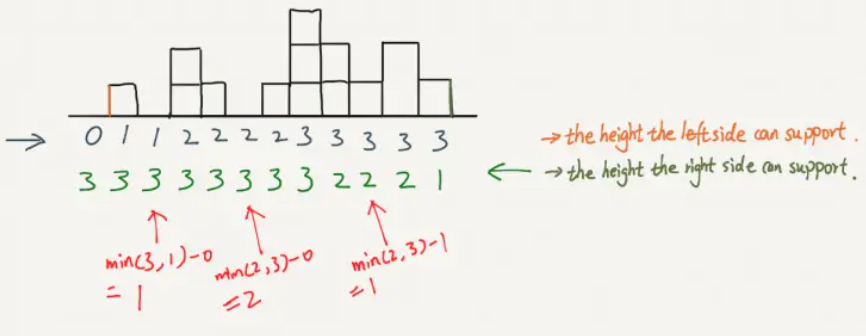
Constraints:

n == height.length

1 <= n <= 2 \* 104

0 <= height[i] <= 105

**Solution : (Using Prefix) (TC – o(n) ,SC-o(n))**



class Solution {

**//At EveryIndex the water trapped in min(max(left),max(right)) - height[i]; i.e including left and right**

    public int trap(int[] height) {

        int leftMax[] = new int[height.length];

        int rightMax[] = new int[height.length];

        int max=0;

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

        {

            max=Math.max(height[i],max);

            leftMax[i]=max;

        }

        max=0;

        for(int i=height.length-1;i>=0;i--)

        {

            max=Math.max(height[i],max);

            rightMax[i]=max;

        }

        int sum=0;

        for(int i=1;i<height.length-1;i++)

        {

            int level = Math.min(leftMax[i],rightMax[i]);

            sum+=level - height[i];

        }

        return sum;

    }

}

**Solution 2: (Two pointer) space – o(1)**

class Solution {

    public int trap(int[] height) {

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

        int leftMax=height[left],rightMax=height[right];

        int sum=0;

        while(left<=right)

        {

            if(height[left]<=height[right])

            {

                if(leftMax<height[left])

                {

                    leftMax=height[left];

                }

                else

                {

                    sum+=(leftMax-height[left]);

                }

                left++;

            }

            else

            {

                if(rightMax<height[right])

                {

                    rightMax=height[right];

                }

                else

                {

                    sum+=(rightMax-height[right]);

                }

                right--;

            }

        }

        return sum;

    }

}

**3) Remove Duplicates from Sorted Array (Leetcode - 26)**

Given an integer array nums sorted in non-decreasing order, remove the duplicates in-place 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).

Example 2:

Input: nums = [0,0,1,1,1,2,2,3,3,4]

Output: 5, nums = [0,1,2,3,4,\_,\_,\_,\_,\_]

Explanation: Your function should return k = 5, with the first five elements of nums being 0, 1, 2, 3, and 4 respectively.

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

Constraints:

1 <= nums.length <= 3 \* 104

-100 <= nums[i] <= 100

nums is sorted in non-decreasing order.

**Solution :**

class Solution {

    public int removeDuplicates(int[] nums) {

        int left = -101;

        int count = 0 ;

        int right=0;

        while(right<nums.length)

        {

            if(left!=nums[right])

            {

                nums[count]=nums[right]; **//updating the unique element at count**

                count++;

                left=nums[right];

            }

            right++;

        }

        return count;

    }

}

**4) 4Sum (Leetcode - 18)**

Given an array nums of n integers, return an array of all the unique quadruplets [nums[a], nums[b], nums[c], nums[d]] such that:

0 <= a, b, c, d < n

a, b, c, and d are distinct.

nums[a] + nums[b] + nums[c] + nums[d] == target

You may return the answer in any order.

Example 1:

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

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

Example 2:

Input: nums = [2,2,2,2,2], target = 8

Output: [[2,2,2,2]]

Constraints:

1 <= nums.length <= 200

-10^9 <= nums[i] <= 10^9

-10^9 <= target <= 10^9

**Solution : (TC – O(n^3), SC – O(1))**

class Solution {

    public List<List<Integer>> fourSum(int[] nums, int target) {

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

        Arrays.sort(nums);

        for(int i=0;i<nums.length-3;i++)

        {

            if(i>0 && nums[i]==nums[i-1]) continue; //avoid dublicates

            for(int j=i+1;j<nums.length-2;j++)

            {

                if(j>i+1 && nums[j]==nums[j-1]) continue;

                int left =j+1,right=nums.length-1;

                long sum = (long) target - (nums[i]+nums[j]); // c+d = target - (a+b)

                while(left<right)

                {

                    long currentSum =(long) nums[left]+nums[right];

                    if(currentSum==sum)

                    {

                        list.add(Arrays.asList(nums[i],nums[j],nums[left],nums[right]));

                        while(left<right && nums[left]==nums[left+1])

                        {

                            left++;

                        }

                        while(left<right && nums[right]==nums[right-1])

                        {

                            right--;

                        }

                        left++;

                        right--;

                    }

                    else if(currentSum<sum)

                    {

                        left++;

                    }

                    else

                    {

                        right--;

                    }

                }

            }

        }

        return list;

    }

}

**5) Count inversions in an array**

What is an inversion of an array?

Definition:

for all i & j < size of array, if i < j then you have to find pair (A[i],A[j]) such that A[j] < A[i].

**Example 1:**

**Input Format**: N = 5, array[] = {1,2,3,4,5}

**Result**: 0

**Explanation**: we have a sorted array and the sorted array has 0 inversions as for i < j you will never find a pair such that A[j] < A[i]. More clear example: 2 has index 1 and 5 has index 4 now 1 < 5 but 2 < 5 so this is not an inversion.

**Example 2:**

**Input Format**: N = 5, array[] = {5,4,3,2,1}

**Result**: 10

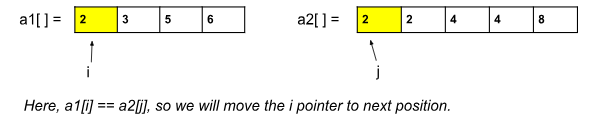
**Explanation**: we have a reverse sorted array and we will get the maximum inversions as for i < j we will always find a pair such that A[j] < A[i]. Example: 5 has index 0 and 3 has index 2 now (5,3) pair is inversion as 0 < 2 and 5 > 3 which will satisfy out conditions and for reverse sorted array we will get maximum inversions and that is (n)\*(n-1) / 2.For above given array there is 4 + 3 + 2 + 1 = 10 inversions.

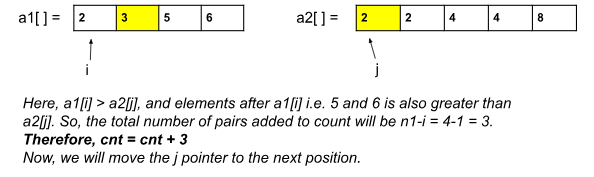
**Example 3:**

**Input Format**: N = 5, array[] = {5,3,2,1,4}

**Result**: 7

**Explanation**: There are 7 pairs (5,1), (5,3), (5,2), (5,4),(3,2), (3,1), (2,1) and we have left 2 pairs (2,4) and (1,4) as both are not satisfy our condition.





**Solution : (using merge sort)**

import java.util.\*;

public class tUf {

private static int merge(int[] arr, int low, int mid, int high) {

ArrayList<Integer> temp = new ArrayList<>(); *// temporary array*

int left = low; *// starting index of left half of arr*

int right = mid + 1; *// starting index of right half of arr*

*//Modification 1: cnt variable to count the pairs:*

int cnt = 0;

*//storing elements in the temporary array in a sorted manner//*

while (left <= mid && right <= high) {

if (arr[left] <= arr[right]) {

temp.add(arr[left]);

left++;

} else {

temp.add(arr[right]);

cnt += (mid - left + 1); *//Modification 2*

right++;

}

}

*// if elements on the left half are still left //*

while (left <= mid) {

temp.add(arr[left]);

left++;

}

*// if elements on the right half are still left //*

while (right <= high) {

temp.add(arr[right]);

right++;

}

*// transfering all elements from temporary to arr //*

for (int i = low; i <= high; i++) {

arr[i] = temp.get(i - low);

}

return cnt; *// Modification 3*

}

public static int mergeSort(int[] arr, int low, int high) {

int cnt = 0;

if (low >= high) return cnt;

int mid = (low + high) / 2 ;

cnt += mergeSort(arr, low, mid); *// left half*

cnt += mergeSort(arr, mid + 1, high); *// right half*

cnt += merge(arr, low, mid, high); *// merging sorted halves*

return cnt;

}

public static int numberOfInversions(int[] a, int n) {

*// Count the number of pairs:*

return mergeSort(a, 0, n - 1);

}

public static void main(String[] args) {

int[] a = {5, 4, 3, 2, 1};

int n = 5;

int cnt = numberOfInversions(a, n);

System.out.println("The number of inversions are: " + cnt);

}

}

**6) Reverse Pairs (Leetcode - 493)**

Given an integer array nums, return the number of reverse pairs in the array.

A reverse pair is a pair (i, j) where:

0 <= i < j < nums.length and

nums[i] > 2 \* nums[j].

Example 1:

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

Output: 2

Explanation: The reverse pairs are:

(1, 4) --> nums[1] = 3, nums[4] = 1, 3 > 2 \* 1

(3, 4) --> nums[3] = 3, nums[4] = 1, 3 > 2 \* 1

Example 2:

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

Output: 3

Explanation: The reverse pairs are:

(1, 4) --> nums[1] = 4, nums[4] = 1, 4 > 2 \* 1

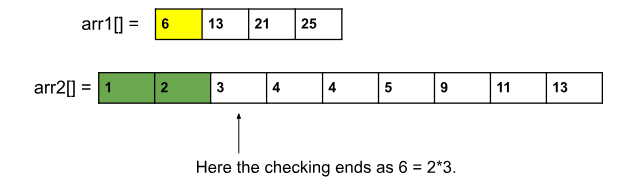
(2, 4) --> nums[2] = 3, nums[4] = 1, 3 > 2 \* 1

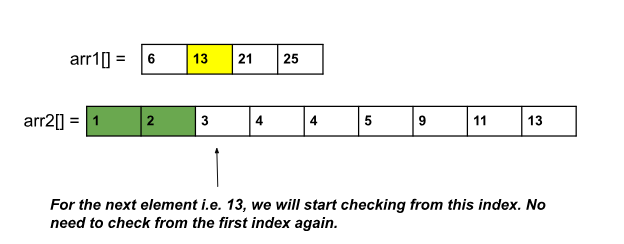
(3, 4) --> nums[3] = 5, nums[4] = 1, 5 > 2 \* 1

Constraints:

1 <= nums.length <= 5 \* 104

-2^31 <= nums[i] <= 2^31 – 1





**Solution :(using Mergesort)**

class Solution {

    public void merge(int low,int mid,int high,int nums[])

    {

        ArrayList<Integer> l = new ArrayList<>();

        int i = low,j=mid+1;

        while(i<=mid && j<=high)

        {

            if(nums[i]<=nums[j])

            {

                l.add(nums[i]);

                i++;

            }

            else

            {

                l.add(nums[j]);

                j++;

            }

        }

        while(i<=mid)

        {

            l.add(nums[i]);

            i++;

        }

        while(j<=high)

        {

            l.add(nums[j]);

            j++;

        }

        for(i=low;i<=high;i++)

        {

            nums[i]=l.get(i-low);

        }

    }

    public int mergeSort(int low,int high,int[] nums)

    {

        int count = 0;

        if(low<high)

        {

            int mid = low+(high-low)/2;

            count += mergeSort(low,mid,nums);

            count+= mergeSort(mid+1,high,nums);

            count+= countPairs(low,mid,high,nums); //above picture

            merge(low,mid,high,nums);

        }

        return count;

    }

    public int countPairs(int low,int mid,int high,int nums[])

    {

        int count =0;

        int right = mid+1;

        for(int i=low;i<=mid;i++)

        {

            while(right<=high && (long)nums[i] > 2L\*nums[right]) right++;

            count+=(right - (mid+1));

        }

        return count;

    }

    public int reversePairs(int[] nums) {

       return mergeSort(0,nums.length-1,nums);

    }

}

**Complexity Analysis**

**Time Complexity:** O(2N\*logN), where N = size of the given array.  
**Reason:**Inside the mergeSort() we call merge() and countPairs() except mergeSort() itself. Now, inside the function countPairs(), though we are running a nested loop, we are actually iterating the left half once and the right half once in total. That is why, the time complexity is O(N). And the merge() function also takes O(N). The mergeSort() takes O(logN) time complexity. Therefore, the overall time complexity will be O(logN \* (N+N)) = O(2N\*logN).

**Space Complexity:**O(N), as in the merge sort We use a temporary array to store elements in sorted order.

**7) Minimum Number of Platforms Required for a Railway/Bus Station**

***Input:****arr[] = {9:00, 9:40, 9:50, 11:00, 15:00, 18:00}, dep[] = {9:10, 12:00, 11:20, 11:30, 19:00, 20:00}****Output:****3****Explanation:****There are at-most three trains at a time (time between 9:40 to 12:00)*

***Input:****arr[] = {9:00, 9:40}, dep[] = {9:10, 12:00}****Output:****1****Explanation:****Only one platform is needed.*

**Solution : (brute force)**

**The maximum value of the overlap counter corresponds to the maximum number of trains that are at the station simultaneously, which directly determines the number of platforms required.**

public static int findPlatform(int arr[], int dep[],

int n)

{

*// plat\_needed indicates number of platforms*

*// needed at a time*

int plat\_needed = 1, result = 1;

*// run a nested loop to find overlap*

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

*// minimum platform*

plat\_needed = 1;

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

if (i != j)

*// check for overlap*

if (arr[i] >= arr[j]

&& dep[j] >= arr[i])

plat\_needed++;

}

*// update result*

result = Math.max(result, plat\_needed);

}

return result;

}

**Solution 2 (Two pointer+sorting):**

****

static int findPlatform(int arr[], int dep[], int n)

{

*// Sort arrival and departure arrays*

Arrays.sort(arr);

Arrays.sort(dep);

*// plat\_needed indicates number of platforms*

*// needed at a time*

int plat\_needed = 1, result = 1;

int i = 1, j = 0;

*// Similar to merge in merge sort to process*

*// all events in sorted order*

while (i < n && j < n) {

*// If next event in sorted order is arrival,*

*// increment count of platforms needed*

if (arr[i] <= dep[j]) {

plat\_needed++;

i++;

}

*// Else decrement count of platforms needed*

else if (arr[i] > dep[j]) {

plat\_needed--;

j++;

}

*// Update result if needed*

if (plat\_needed > result)

result = plat\_needed;

}

return result;

}

**8) 3Sum Closest (Leetcode - 16)**

Given an integer array nums of length n and an integer target, find three integers in nums such that the sum is closest to target.

Return the sum of the three integers.

You may assume that each input would have exactly one solution.

Example 1:

Input: nums = [-1,2,1,-4], target = 1

Output: 2

Explanation: The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

Example 2:

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

Output: 0

Explanation: The sum that is closest to the target is 0. (0 + 0 + 0 = 0).

Constraints:

3 <= nums.length <= 500

-1000 <= nums[i] <= 1000

-104 <= target <= 104

**Solution :**

class Solution {

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

        int close = Integer.MAX\_VALUE;

        Arrays.sort(nums);

        for(int i=0;i<nums.length-2;i++)

        {

            int sum = target - nums[i]; //(b+c) = target - a;

            int left = i+1,right = nums.length-1;

            while(left<right)

            {

                int s = nums[left]+nums[right];

                if(Math.abs(target-(s+nums[i]))<Math.abs(target-close))

                {

                    close=s+nums[i];

                }

                if(s<sum)

                {

                    left++;

                }

                else

                {

                    right--;

                }

            }

        }

        return close;

    }

}

**9) Rotate Array (Leetcode - 189)**

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 <= 10^5

**Solution :**

class Solution {

    void reverse(int[] nums, int left, int right) {

        while (left < right) {

            int temp = nums[left];

            nums[left] = nums[right];

            nums[right] = temp;

            left++;

            right--;

        }

    }

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

        k = k%nums.length;

        reverse(nums,0,nums.length-1);

        reverse(nums,0,k-1);

        reverse(nums,k,nums.length-1);

    }

}