Problem 1:

There are n kids with candies. You are given an integer array candies, where each candies[i] represents the number of candies the i th kid has, and an integer extraCandies, denoting the number of extra candies that you have. Return a boolean array result of length n, where result[i] is true if, after giving the i th kid all the extraCandies, they will have the greatest number of candies among all the kids, or false otherwise. Note that multiple kids can have the greatest number of candies.

class Solution {

public List<Boolean> kidsWithCandies(int[] candies, int extraCandies) {

List<Boolean>arr1=new ArrayList<>();

int k=0;

int max=candies[0];

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

{

if(candies[i]>max)

max=candies[i];

}

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

{

int sum=0;

sum=sum+candies[i]+extraCandies;

if(sum>=max)

arr1.add(true);

else

arr1.add(false);

}

return arr1;

}

}

Problem 2 :

You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the i th line are (i, 0) and (i, height[i]). Find two lines that together with the x-axis form a container, such that the container contains the most water. Return the maximum amount of water a container can store. Notice that you may not slant the container.

class Solution {

public int maxArea(int[] height) {

int ans = 0;

int l = 0;

int r = height.length - 1;

while (l < r) {

int minHeight = Math.min(height[l], height[r]);

ans = Math.max(ans, minHeight \* (r - l));

if (height[l] < height[r])

++l;

else

--r;

}

return ans;

}

}

Problem 3:

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.

class Solution {

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

Arrays.sort(nums);

int closestSum = nums[0] + nums[1] + nums[2];

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

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

while (left < right) {

int sum = nums[i] + nums[left] + nums[right];

if (Math.abs(sum - target) < Math.abs(closestSum - target)) {

closestSum = sum;

}

if (sum < target) {

left++;

} else {

right--;

}

}

}

return closestSum;

}

}

Problem 4:

A permutation of an array of integers is an arrangement of its members into a sequence or linear order. ● For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1]. The next permutation of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the next permutation of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order). ● ● ● For example, the next permutation of arr = [1,2,3] is [1,3,2]. Similarly, the next permutation of arr = [2,3,1] is [3,1,2]. While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement. Given an array of integers nums, find the next permutation of nums. The replacement must be in place and use only constant extra memory.

class Solution {

public void nextPermutation(int[] nums) {

int i = nums.length - 2;

while (i >= 0 && nums[i] >= nums[i+1]) {

i--;

}

if (i >= 0) {

int j = nums.length - 1;

while (j >= 0 && nums[j] <= nums[i]) {

j--;

}

swap(nums, i, j);

}

reverse(nums, i+1);

}

private void swap(int[] nums, int i, int j) {

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

private void reverse(int[] nums, int start) {

int i = start, j = nums.length - 1;

while (i < j) {

swap(nums, i, j);

i++;

j--;

}

}

}

Problem 5:

There is an integer array nums sorted in ascending order (with distinct values). Prior to being passed to your function, nums is possibly rotated at an unknown pivot index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k1]] (0-indexed). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2]. Given the array nums after the possible rotation and an integer target, return the index of target if it is in nums, or -1 if it is not in nums. You must write an algorithm with O(log n) runtime complexity.

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

int left = 0;

int right = nums.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

if (nums[mid] == target) {

return mid;

}

if (nums[mid] >= nums[left]) {

if (target >= nums[left] && target < nums[mid]) {

right = mid - 1;

} else {

left = mid + 1;

}

} else {

if (target > nums[mid] && target <= nums[right]) {

left = mid + 1;

} else {

right = mid - 1;

}

}

}

return -1;

}

Problem 6:

Given a collection of numbers, nums, that might contain duplicates, return all possible unique permutations in any order.

class Solution {

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

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

helper(nums,0,ans);

return new ArrayList(ans);

}

public void helper(int[] nums, int index, Set<List<Integer>> ans)

{

if(index==nums.length)

{

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

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

list.add(nums[i]);

}

ans.add(list);

return;

}

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

{

swap(i,index,nums);

helper(nums, index+1, ans);

swap(i,index,nums);

}

}

public static void swap(int i , int j, int[] nums){

int t=nums[i];

nums[i]=nums[j];

nums[j]=t;

}

}

Problem 7:

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

class Solution {

public int maxSubArray(int[] nums) {

int max = nums[0];

int curr = nums[0];

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

curr = Math.max(curr + nums[i], nums[i]);

if (curr > max) {

max = curr;

}

}

return max;

}

}

Problem 8:

Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

class Solution {

public int[][] merge(int[][] intervals) {

List<int[]> res = new ArrayList<>();

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

return res.toArray(new int[0][]);

}

Arrays.sort(intervals,(a,b)-> a[0]-b[0]);

int start = intervals[0][0];

int end = intervals[0][1];

for(int[] i : intervals){

if(i[0] <= end){

end = Math.max(end,i[1]);

}else{

res.add(new int[]{start,end});

start = i[0];

end = i[1];

}

}

res.add(new int[]{start,end});

return res.toArray(new int[0][]);

}

}

Problem 9:

Given an array nums with n objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue. We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively. You must solve this problem without using the library's sort function.

import java.util.\*;

public class Test{

public static void main (String[] args) {

Scanner sc=new Scanner(System.in);

int n=sc.nextInt();

int arr[]=new int[n];

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

{

arr[i]=sc.nextInt();

}

int j=0;

int k=n-1;

int l=0;

int m=0;

int p=0;

int temp=0;

while(j<=k)

{

if(arr[j]==0)

{

swap(temp,arr,l,j);

l++;

j++;

}

else if(arr[j]==2)

{

swap(temp,arr,j,k);

k--;

}

else

{

j++;

}

}

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

{

System.out.println(arr[i]);

}

}

public static void swap(int temp,int arr[],int j,int k)

{

temp=arr[j];

arr[j]=arr[k];

arr[k]=temp;

}

}

Problem 10:

You are given an integer array prices where prices[i] is the price of a given stock on the i th day. On each day, you may decide to buy and/or sell the stock. You can only hold at most one share of the stock at any time. However, you can buy it then immediately sell it on the same day. Find and return the maximum profit you can achieve.

class Solution {

public int maxProfit(int[] prices) {

int l = 0;

int r = 1;

int max = 0;

while(r < prices.length)

{

if(prices[l] < prices[r])

{

max += prices[r] - prices[l];

l++;

}

else{

l = r;

}

r++;

}

return max;

}

}

Problem 11:

Given a 0-indexed integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks.

class Solution {

public int findPeakElement(int[] nums) {

int max=nums[0];

int res=0;

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

{

if(nums[i]>max)

{

max=nums[i];

res=i;

}

}

return res;

}

}

Problem 12:

Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.

class Solution {

public double findMedianSortedArrays(int[] nums1, int[] nums2) {

int arr[]=new int[nums1.length+nums2.length];

int p=0;

int q=0;

int k=0;

while(p<nums1.length && q<nums2.length)

{

if(nums1[p]<nums2[q])

{

arr[k++]=nums1[p];

p++;

}

else

{

arr[k++]=nums2[q];

q++;

}

}

while(p==nums1.length && q!=nums2.length)

{

arr[k++]=nums2[q++];

}

while(p!=nums1.length && q==nums2.length)

{

arr[k++]=nums1[p++];

}

if(arr.length%2!=0)

return arr[arr.length/2];

else

{

return ((double)(arr[(arr.length/2)-1])+(arr[arr.length/2]))/2;

}

}

}

Problem 14:

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

class Solution {

public int findMaxConsecutiveOnes(int[] nums) {

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

int max=Integer.MIN\_VALUE;

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

{

if(nums[i]==1)

al.add(nums[i]);

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

if(nums[i]!=1)

{

al.clear();

}

}

return max;

}

}

Problem 15:

Given an integer array nums and an integer k, return the kth largest element in the array. Note that it is the kth largest element in the sorted order, not the kth distinct element. You must solve it in O(n) time complexity

class Solution {

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

PriorityQueue<Integer>pq=new PriorityQueue<>(Collections.reverseOrder());

for(int i: nums){

pq.add(i);

}

while(k>1){

pq.poll();

k--;

}

return pq.poll();

}

}