**LC#1:TWO SUM**

**APPROACH 1:BRUTE(O(N^2))**

**->Iterate by two loops and check every combination**

**CODE:**

class Solution {

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

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

{

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

{

if(nums[i] +nums[j] == target)

return new int[]{i,j};

}

}

return new int[] {};

}

}

**APPROACH 2:USING HASHMAP**

**->put the value and index in the hashmap orderly. If the remaining required is previously stored in the hashmap then return both indices**

**CODE:**

class Solution {

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

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

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

{

int diff = target - nums[i];

if(hm.containsKey(diff))

{

return new int[] {i,hm.get(diff)};

}

hm.put(nums[i],i);

}

return new int [] {};

}

}

**APPROACH 3:IF SORTING ALLOWED ONLY CAN BE DONE**

->**sort the array and use two pointer technique (binary search)**

class Solution {

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

        int len = nums.length;

        int arr[][] = new int [len][2];

        for(int i = 0 ; i< len ;i++)

        {

            arr[i][0] = nums[i];

            arr[i][1] = i;

        }

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

        int left = 0 , right = len-1;

        while(left<right)

        {

            int sum = arr[left][0] + arr[right][0];

            if(sum==target)

                return new int[]{arr[left][1],arr[right][1]};

            else if(sum<target)

                left++;

            else

                right--;

        }

        return new int[]{};

    }

}

**LC#167 : TWO SUM – II :INPUT ARRAY IS SORTED**

**->We can use two pointers directly (binary search)**

**CODE:**

Class Solution {

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

        int n = nums.length-1;

        int arr[] = new int[2];

        int left = 0 , right = n;

        while(left <=right)

        {

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

            if(sum==target)

            {

                arr[0] = left+1;

                arr[1] = right+1;

                break;

            }

            else if(sum<target)

                left++;

            else

                right--;

        }

        return arr;

    }

}

**LC#15:3SUM**

**->find when sum of triplets =0;**

**APPROACH 1:BRUTE(TLE)**

**->use three loops find all the possible and if match add It to the array list**

import java.util.\*;

class Solution {

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

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

        int n = nums.length;

        for (int i = 0; i < n; i++) {

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

                for (int k = j + 1; k < n; k++) {

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

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

                        Collections.sort(temp);

                        if (!res.contains(temp)) { // avoid duplicates

                            res.add(temp);

                        }

                    }

                }

            }

        }

        return res;

    }

}

**APPROACH 2:HASHMAP WITHOUT SORTING**

**->outer loop creates target after that perform two sum inside using another for loop to find target matches -> sum=0**

class Solution {

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

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

        int n = nums.length;

        for(int i = 0 ; i< n ; i++)

        {

            int target = - nums[i];

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

            for(int j = i+1;j<n;j++)

            {

                int diff = target - nums[j];

                if(hm.containsKey(diff))

                {

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

                    Collections.sort(triplet);

                    res.add(triplet);

                }

                hm.put(nums[j],j);

            }

        }

**return new ArrayList(res);**

    }

}

**APPROACH 3:TWO POINTER**

**I)USING SET SPACE-> O(N^2) NOT OPTIMAL**

class Solution {

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

Arrays.sort(nums);

int n = nums.length;

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

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

{ int target = -nums[i];

int left = i+1, right = n-1;

while(left<right)

{

if(nums[left]+nums[right]==target)

{

List<Integer> temp = Arrays.asList(nums[left], nums[right],nums[i]);

res.add(temp);

left++;

right--;

}

else if(nums[left]+nums[right]<target)

left++;

else

right--;

}

}

return new ArrayList(res);

}

}

**II)OPTIMAL REDUCED THE SPACE USED BY SET**

class Solution {

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

        Arrays.sort(nums);

        int n = nums.length;

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

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

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

            int left = i + 1, right = n - 1;

            while (left < right) {

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

                if (sum == 0) {

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

                    //skipping duplicates wihout using set

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

                        left++;

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

                        right--;

                    left++;

                    right--;

                } else if (sum < 0) {

                    left++;

                } else {

                    right--;

                }

            }

        }

        return res;}}

**LC#11:CONTAINER WITH MOST WATER:**



**->FIND THE MAX AREA:**

**APPROACH 1:BRUTE FORCE**

* **Check every possible using nested for loops(n^2):OP:TLE as max inp size is 10^5 which reject the solution of N^2**

**CODE:**

class Solution {

    public int maxArea(int[] height) {

        int maxArea = 0 , n = height.length;

        for(int i = 0 ; i<n;i++)

        {

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

            {

                int breadth = j-i;

                int length = Math.min(height[i], height[j]);

                int currArea = length\*breadth;

                maxArea = Math.max(maxArea,currArea);

            }

        }

        return maxArea;

    }}

**APPROACH 2:TWO POINTER**

**->have two pointers “left” and “right”**

**->loop until both meet increase the pointer when whichever is lower**

**CODE:**

class Solution {

    public int maxArea(int[] height) {

        int max=0,n=height.length;

        int left = 0 , right = n-1;

        while(left <= right)

        {

            int breadth = right - left;

            int length = Math.min(height[left],height[right]);

            int currArea = length \* breadth;

            max=Math.max(max,currArea);

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

            {

                left++;

            }

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

            {

                left++;

                right--;

            }

            else

                right--;

        }

        return max;

    }}

**LC#349:INTERSECTION OF TWO ARRAYS:**

**RETURN THE INTERSECTION OF TWO ARRAYS BUT IN THE RESULT IT SHOULD NOT CONTAIN ANY DUPLICATES**

**APPROACH 1:**

**->Using ArrayList , two for loops**

**->add if a not in b**

**CODE:**

class Solution {

public int[] intersection(int[] nums1, int[] nums2) {

List<Integer> res = new ArrayList();

for(int num : nums1)

{

for(int num2:nums2)

{

if(num==num2)

{

if(!res.contains(num))

{

res.add(num);

break;

}

}

}

}

int[] ans = new int[res.size()];

for (int i = 0; i < res.size(); i++)

ans[i] = res.get(i);

return ans;}}

**APPROACH 2: USING HASHSET**

* **Put nums1 elements in set1 automaticaly not allows duplicates**
* **Put nums2 elements in set2 only if it is in set1**
* **Copy the elements in set2 to res array**

**CODE:**

class Solution {

public int[] intersection(int[] nums1, int[] nums2) {

Set<Integer> set1 = new HashSet();

Set<Integer> set2 = new HashSet();

for(int a:nums1)

{

set1.add(a);

}

for(int b :nums2)

{

if(set1.contains(b))

set2.add(b);

}

int l = set2.size();

int res[] = new int[l] , i = 0;

for(int val : set2)

res[i++] = val;

return res;

}

}

**APPROACH 3:TWO POINTER AND SORTING (TO REDUCE MEMORY)**

**CODE:**

public class Solution {

    public int[] intersection(int[] nums1, int[] nums2) {

        Arrays.sort(nums1);

        Arrays.sort(nums2);

        int i = 0, j = 0;

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

        Integer lastAdded = null;

        while (i < nums1.length && j < nums2.length) {

            if (nums1[i] == nums2[j]) {

                if (lastAdded == null || lastAdded != nums1[i]) {

                    res.add(nums1[i]);

                    lastAdded = nums1[i];

                }

                i++;

                j++;

            }

            else if (nums1[i] < nums2[j]) {

                i++;

            } else {

                j++;

            }

        }

        int[] ans = new int[res.size()];

        for (int k = 0; k < res.size(); k++)

            ans[k] = res.get(k);

        return ans;}}