**Program**

**1.Contains Duplicate** (**Input:** nums = [1,2,3,1],**Output:** true)

***Hash Set***

**Logic c#**

int n = arr.Length;

if (n == 0)

return false;

var Dup = new HashSet<int>();

foreach (var item in arr) {

if (Dup.Contains(item)) {

true;}

else{

Dup.Add(item);}}

return false;

**Logic JS**

var containsDuplicate = function ( nums ) {

const set = new Set();

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

if ( set.has( nums[i] ) ) return true;

else set.add( nums[i] );

}

return false;

};

**2.Missing Number (Input:** nums = [3,0,1],**Output:** 2 **)**

***Arrays***

**Logic c#**

public int MissingNumber(int[] nums) {

int n = nums.Length;

int sum = 0;

int sumN = n \* (n + 1) / 2;

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

{

sum += nums[i];

}

return sumN - sum;

}

**Logic JS**

var missingNumber = function(nums) {

let n = nums.length;

let sumN = n\*(n+1)/2;

let sum = 0;

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

sum += nums[i];

}

return sumN-sum;

};

**3.Numbers Disappeared in array (Input:** nums = [4,3,2,7,8,2,3,1],**Output:** [5,6]**)**

***Hashset & List***

**Logic c#**

public IList<int> FindDisappearedNumbers(int[] arr) {

int n = arr.Length;

var numbers = new HashSet<int>();

List<int> missingNumbers = new List<int>();

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

numbers.Add(arr[i]);

}

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

if(!numbers.Contains(j)){

missingNumbers.Add(j);

}

}

return missingNumbers;

}

**Logic JS**

var findDisappearedNumbers = function(arr) {

let data =new Set();

let missingNumbers = [];

for(let i = 0; i<arr.length; i++){

data.add(arr[i]);

}

for(let j = 1; j<arr.length+1; j++){

if(!data.has(j)){

missingNumbers.push(j);

}

}

return missingNumbers;

};

**4.Single Number ( Input:** nums = [2,2,1],**Output:** 1 **)**

***HashSet***

**Logic c#**

public int SingleNumber(int[] nums) {

var data = new HashSet<int>();

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

{

if (data.Contains(nums[i]))

data.Remove(nums[i]);

else

data.Add(nums[i]);

}

return data.ElementAt(0);

}

**Logic JS**

var singleNumber = function(nums) {

let data =new Set();

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

if(data.has(nums[i])){

data.delete((nums[i]));

}

else

data.add(nums[i]);

}

let it = data.values();

let first = it.next();

let value = data.values().next().value;

return value;

};

**5.Climbing Steps (Input:** n = 2,**Output:** 2

**Explanation:** There are two ways to climb to the top.

1. 1 step + 1 step

2. 2 steps**)**

***Dynamic programming Fibonacci Series***

**Logic c#**

public int ClimbStairs(int n) {

if (n <= 2)

return n;

int[] res = new int[n + 1];

res[1] = 1;

res[2] = 2;

for (int i = 3; i <= n; i++)

res[i] = res[i - 1] + res[i - 2];

return res[n];

}

**Logic JS**

var climbStairs = function(n) {

if(n<=2)

return n;

let res = [];

res[1]= 1;

res[2] = 2;

for(let i = 3; i<=n; i++){

res[i] = res[i-1] + res[i-2];

}

return res[n];

};

**6.Best time to buy and Sell stocks(Input:** prices = [7,1,5,3,6,4],**Output:** 5**)**

***Greedy Approach***

**Logic c#**

public int MaxProfit(int[] prices) {

int maxProfit = 0;

int min = int.MaxValue;

for(int i = 0; i<prices.Length; i++)

{

if (prices[i] < min)

{

min = prices[i];

}

else

maxProfit = Math.Max(maxProfit,prices[i]-min);

}

return maxProfit;

}

**Logic JS**

var maxProfit = function(prices) {

let minPrice = Number.MAX\_SAFE\_INTEGER;

let maxProfit = 0;

for (let i = 0; i < prices.length; i++) {

if ( prices[i] < minPrice) {

minPrice = prices[i];

}

else {

maxProfit = Math.max(maxProfit,prices[i]-minPrice);

}

}

return maxProfit;

};**7. Maximum SubArray(Input:** nums = [-2,1,-3,4,-1,2,1,-5,4],**Output:** 6**)**

***Kaden’s algorithm***

**Logic c#**

public int MaxSubArray(int[] nums) {

if(nums == null)

return 0;

int interval = 0;

int ans = nums[0];

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

interval = Math.Max(nums[i],interval+nums[i]);

ans = Math.Max(interval,ans);

}

return ans;

}

**Logic JS**

var maxSubArray = function(nums) {

if(nums.length == 0)

return;

let ans = nums[0];

let interval = 0;

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

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

ans = Math.max(ans, interval);

}

return ans;

};

**8. Range Sum Query – Immutable(Input**

["NumArray", "sumRange", "sumRange", "sumRange"]

[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]

**Output**

[null, 1, -1, -3]

**)**

***Dynamic Programming***

**Logic c#**

public class NumArray {

int[] sums;

public NumArray(int[] nums) {

var n = nums.Length;

sums = new int[n + 1];

sums[0] = 0;

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

sums[i] = sums[i-1] + nums[i-1];

}

}

public int SumRange(int i, int j) {

return sums[j + 1] - sums[i];

}

}**Logic JS**

let sums = [];

var NumArray = function(nums) {

let n = nums.length;

sums = new Array(n + 1);

sums[0] = 0;

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

sums[i] = sums[i-1] +nums[i-1];

}

};

NumArray.prototype.sumRange = function(i, j) {

return sums[j + 1] - sums[i];

};

**9. Counting Bits(Input:** n = 2,**Output:** [0,1,1]**)**

***Dynamic programming***

**Logic c#**

public int[] CountBits(int num) {

int [] arr = new int[num+1];

arr[0] = 0;

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

if(i%2 == 1){

arr[i] = arr[i-1] +1;

}

else

arr[i] = arr[i/2];

}

return arr;

}

**Logic JS**

var countBits = function(n) {

let arr = new Array(n+1);

arr[0] = 0;

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

if(i%2 == 1){

arr[i] = arr[i-1] +1;

}

else

arr[i] = arr[i/2];

}

return arr;

};

**10.Linked list cycle(Input:** head = [3,2,0,-4], pos = 1,**Output:** true**)**

***Fast nd slow pointers***

**C#Logic**

public bool HasCycle(ListNode head) {

if (head == null || head.next == null)

return false;

ListNode slow = head;

ListNode fast = head.next;

while (slow != fast)

{

if (fast == null || fast.next == null)

return false;

slow = slow.next;

fast = fast.next.next;

}

return true;

}

**Js logic**

var hasCycle = function(head) {

if (head == null || head.next == null)

return false;

let slow = head;

let fast = head.next;

while (slow != fast)

{

if (fast == null || fast.next == null)

return false;

slow = slow.next;

fast = fast.next.next;

}

return true;

};

**11. Middle of linked list (Input:** head = [1,2,3,4,5]**Output:** [3,4,5]**)**

***Fast & slow pointers***

**Logic c#**

public ListNode MiddleNode(ListNode head) {

if(head == null)

return null;

ListNode fast = head;

ListNode slow = head;

while(fast != null && fast.next != null)

{

fast = fast.next.next;

slow = slow.next;

}

return slow;

}

**Logic JS**

var middleNode = function(head) {

if(head == null)

return null;

let slow = head;

let fast = head;

while(fast!= null && fast.next != null){

slow = slow.next;

fast = fast.next.next;

}

return slow;

};

**12. Palindrome Linked List (Input:** head = [1,2,2,1] **Output:** true**)**

***Fast & slow pointers***

**Logic c#**

public bool IsPalindrome(ListNode head) {

if(head == null || head.next == null)

return true;

ListNode slow = head;

ListNode fast = head;

While(fast != null && fast.next != null){

slow = slow.next;

fast = fast.next.next;

}

slow = reverse(slow);

fast = head;

while (slow!= null){

if(slow.val != fast.val)

return false;

slow = slow.next;

fast = fast.next;

}

return true;

}

Private ListNode Reverse(ListNode head){

ListNode prev = null;

While(head != null){

ListNode nextNode = head.next;

head.next = prev;

prev = head;

head = nextNode;

}

return prev;

}

**Logic JS**

var isPalindrome = function(head) {

if(head == null || head.next == null)

return true;

let slow = head;

let fast = head;

while(fast!= null && fast.next!= null)

{

fast = fast.next.next;

slow = slow.next;

}

slow = reverse(slow);

fast = head;

while(slow != null){

if(slow.val!= fast.val){

return false;

}

fast = fast.next;

slow = slow.next;

}

return true;

};

function reverse(head){

if (head == null)

return head;

let prev = null;

while(head != null){

let nextNode = head.next;

head.next = prev;

prev = head;

head = nextNode;

}

return prev;

}

**13. Remove LinkedList(Input:** head = [1,2,6,3,4,5,6], val = 6 **Output:** [1,2,3,4,5]**)**

***Fast & slow pointers***

**Logic c#**

public ListNode RemoveElements(ListNode head, int val) {

while(head != null && head.val == val){

head = head.next;

}

Let currentNode = head;

While(currentNode != null && currentNode.next != null){

If(currentNode.next.val == val){

currentNode.next = currentNode.next.next;

}

else

currentNode = currentNode.next;

}

return head;

}

**Logic JS**

var removeElements = function(head, val) {

while(head != null && head.val == val){

head = head.next;

}

let currentNode = head;

while(currentNode != null && currentNode.next != null){

if(currentNode.next.val == val)

currentNode.next = currentNode.next.next;

else

currentNode = currentNode.next;

}

return head;

};

**14. Remove Duplicates from Sorted List(Input:** head = [1,1,2] **Output:** [1,2]**)**

***Fast and slow pointer***

**Logic c#:**

public ListNode DeleteDuplicates(ListNode head) {

if (head == null || head.next == null)

return head;

}

ListNode currentNode = head;

While(currentNode != null && currentNode.next!= null){

If(currentNode.val == currentNode.next.val){

currentNode.next = currentNode.next.next;

}

currentNode = currentNode.next;

}

return head;

}

**Logic JS**

var deleteDuplicates = function(head) {

if (head == null || head.next == null)

return head;

}

let currentNode = head;

while(currentNode != null && currentNode.next != null){

if(currentNode.next.val == currentNode.val)

currentNode.next = currentNode.next.next;

else

currentNode = currentNode.next;

}

return head;

};

**15. Reverse Linked List**

***Fast and slow pointers***

**Logic c#:**

Public ListNode Reverse(ListNode head){

If(head == null){

return head;

}

ListNode prev = null;

While(head != null)

{

ListNode nextNode = head.next;

Head.next = prev;

Prev= head;

Head = nextNode;

}

Return prev;

}

**Logic JS**

var reverseList = function(head) {

if (head == null)

return head;

let prev = null;

while(head != null){

let nextNode = head.next;

head.next = prev;

prev = head;

head = nextNode;

}

return prev;

};

**16. Merge 2 sorted Lists (Input:** list1 = [1,2,4], list2 = [1,3,4], **Output:** [1,1,2,3,4,4]**)**

***Fast and slow pointers***

**Logic c#**

public ListNode MergeTwoLists(ListNode list1, ListNode list2) {

ListNode temp = new ListNode(0);

ListNode currentNode = temp;

while(list1 != null && list2 != null){

if(list1.val <list2.val){

currentNode.next = list1;

list1 = list1.next;

}

else{

currentNode.next = list2;

list2 = list2.next;

}

currentNode = currentNode.next;

}

if(list1 != null){

currentNode.next = list1;

list1 = list1.next;

}

if(list2 != null){

currentNode.next = list2;

list2 = list2.next;

}

return temp.next;

}

}

**LOGIC JS**

var mergeTwoLists = function(list1, list2) {

let tempNode = new ListNode(0);

let currentNode = tempNode;

while(list1!= null && list2 != null){

if(list1.val <list2.val){

currentNode.next = list1;

list1 = list1.next;

}

else

{

currentNode.next = list2;

list2 = list2.next;

}

currentNode = currentNode.next;

}

if(list1 != null)

{

currentNode.next = list1;

list1 = list1.next;

}

if(list2 != null)

{

currentNode.next = list2;

list2 = list2.next;

}

return tempNode.next;

};

**17. Meeting Rooms (**Input: intervals = [(0,30),(5,10),(15,20)]

Output: false

Explanation:

(0,30), (5,10) and (0,30),(15,20) will conflict

**)**

***Intervals***

**LOGIC C#**

public class Program

{

public class Interval

{

public int start, end;

public Interval(int s, int e)

{

start = s; end = e;

}

}

public bool canAttendAMetting(Interval[] intervals)

{

int[] starts = new int[intervals.Length];

int[] ends = new int[intervals.Length];

for(int i = 0; i<intervals.Length; i++)

{

starts[i] = intervals[i].start;

ends[i] = intervals[i].end;

}

Array.Sort(starts);

Array.Sort(ends);

for(int i = 0; i<starts.Length-1; i++)

{

if (starts[i + 1] < ends[i])

return false;

}

return true;

}

static void Main(string[] args)

{

Program p = new Program();

Interval[] data = new Interval[]

{

new Interval(0,30),

new Interval(5,10),

new Interval(15,20),

//new Interval(5,8),

//new Interval(9,15),

};

bool result = p.canAttendAMetting(data);

Console.WriteLine(result);

}

}

**LOGIC JS**

class Interval {  
 constructor(start,end) {  
 this.start = start  
 this.end = end  
 }  
}  
class meetingRooms {  
  
 canAttendMeetings(intervals) {  
 // Write your code here  
 let starts = new ***Array***(intervals.length);  
 let ends = new ***Array***(intervals.length);  
 for (let i = 0; i < intervals.length; i++) {  
 starts[i] = intervals[i].start;  
 ends[i] = intervals[i].end;  
 }  
 //starts.sort();  
 //ends.sort();  
 for (let i = 0; i < starts.length-1; i++) {  
 if (starts[i + 1] < ends[i])  
 return false;  
 }  
 return true;  
 }  
}  
const ***p*** =new meetingRooms();  
let ***data*** = [  
 new Interval(0, 30),  
 new Interval(5, 10),  
 new Interval(15, 20)  
  
 //new Interval(5,8),  
 //new Interval(9,15),  
 ];  
  
***console***.log(***p***.canAttendMeetings(***data***));

**18. Binary Search (Input:** nums = [-1,0,3,5,9,12], target = 9

**Output:** 4

**Explanation:** 9 exists in nums and its index is 4

**)**

**LOGIC C#**

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

int l = 0;

int r = nums.Length-1;

while(l<=r){

int mid = (l+r)/2;

if(nums[mid] == target)

return mid;

else if(nums[mid] <target)

l = mid +1;

else

r = mid-1;

}

return -1;

}

**LOGIC JS**

var search = function(nums, target) {

let l = 0, r = nums.length-1;

while(l<= r){

let mid = Math.floor((l + r) / 2);

if(nums[mid] == target)

return mid;

if(nums[mid]<target)

l = mid+1;

else

r = mid-1;

}

return -1;

};

**19. Find Smallest Letter Greater Than Target ( Input:** letters = ["c","f","j"], target = "a"

**Output:** "c"**)**

***Binary Search***

**LOGIC C#**

public char NextGreatestLetter(char[] letters, char target) {

int l = 0, r = letters.Length -1;

while(l <=r){

int mid = l +(r-l)/2;

if(letters[mid]<= target)

l = mid+1;

else

r = mid-1;

}

if(l> letters.Length)

l = 0;

return letters[l];

}

**LOGIC JS**

var nextGreatestLetter = function(letters, target) {

let l = 0, r = letters.length-1;

while(l<=r){

let mid = Math.floor((l + r) / 2);

if(letters[mid]<= target) {

l = mid + 1;

}

else{

r = mid-1;

}

}

if(l>= letters.length)

l = 0;

return letters[l];

};

**20. Peak Index in a Mountain Array (Input:** arr = [0,1,0]

**Output:** 1**)**

***Binary Search***

**LOGIC C#**

public int PeakIndexInMountainArray(int[] arr) {

int left = 0; int right = arr.Length -1;

while(left <= right){

int mid = (left+right)/2;

if(arr[mid]<arr[mid+1])

left = mid+1;

else

right = mid-1;

}

return left;

}

**LOGIC JS**

var peakIndexInMountainArray = function(arr) {

let left = 0, right = arr.length-1;

while(left<=right){

let mid = Math.floor((left + right) / 2);

if(arr[mid]<arr[mid+1])

left = mid+1;

else

right = mid-1;

}

return left;

};

**21. Average of Levels in Binary Tree (Input:** root = [3,9,20,null,null,15,7] **Output:** [3.00000,14.50000,11.00000]**)**

***BFS***

**LOGIC C#**

public IList<double> AverageOfLevels(TreeNode root) {

List<double> res = new List<double>();

if (root == null)

return res;

Queue<TreeNode> queue = new Queue<TreeNode>();

queue.Enqueue(root);

while (queue.Count > 0)

{

double sum = 0;

int size = queue.Count;

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

{

var curr = queue.Dequeue();

sum += curr.val;

if (curr.left != null)

queue.Enqueue(curr.left);

if (curr.right != null)

queue.Enqueue(curr.right);

}

res.Add(sum / size);

}

return res;

}

**LOGIC JS**

var averageOfLevels = function(root) {

if (!root) {

return [];

}

let res = [];

let queue = [];

queue.push(root);

while (queue.length > 0) {

let numOfNodes = queue.length;

let nodeSum = 0;

for (let i = 0; i < numOfNodes; i++) {

let node = queue.shift();

nodeSum += node.val;

if (node.left) {

queue.push(node.left);

}

if (node.right) {

queue.push(node.right);

}

}

res.push(nodeSum / numOfNodes);

}

return res;

};

**22. Minimum Depth of Binary Tree (Input:** root = [3,9,20,null,null,15,7],**Output:** 2**)**

***DFS***

**LOGIC C#**

public int MinDepth(TreeNode root) {

if (root == null)

return 0;

if (root.left == null && root.right == null)

return 1;

int left = root.left != null ? MinDepth(root.left) : int.MaxValue;

int right = root.right != null ? MinDepth(root.right) : int.MaxValue;

return Math.Min(left, right) + 1;

}

**LOGIC JS**

var minDepth = function(root) {

if(root == null)

return 0;

if(root.left == null && root.right == null)

return 1;

let left = root.left != null ? minDepth(root.left) : Number.MAX\_SAFE\_INTEGER;

let right = root.right != null ? minDepth(root.right) : Number.MAX\_SAFE\_INTEGER;

return Math.min(left, right) +1;

};

**23. Same Tree ( Input:** p = [1,2,3], q = [1,2,3] **Output:** true**)**

***DFS***

**LOGIC C#**

public bool sameTree(TreeNode p,TreeNode q){

if (p == null || q == null )

return false;

if (p == null && q == null )

return true;

if(p.val == q.val ){

return sameTree(p.left,q.left) && sameTree(p.right,q.right);

}

return false;

}

**LOGIC JS**

var isSameTree = function(p, q) {

if(p === null && q === null)

return true;

if(p === null || q === null)

return false;

if(p.val == q.val){

return isSameTree(p.left,q.left) && isSameTree(p.right,q.right);

}

return false;

};

**24. Path Sum ( Input:** root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22

**Output:** true**)**

***DFS***

public bool HasPathSum(TreeNode root, int targetSum) {

if(root == null )

return false;

if(root.left == null && root.right == null && targetSum == root.val)

return true;

return HasPathSum(root.left,targetSum-root.val) || HasPathSum(root.right,targetSum-root.val)

}

**LOGIC JS**

var hasPathSum = function(root, targetSum) {

if(root == null)

return false;

if(root.left == null && root.right == null && root.val == targetSum)

return true;

return hasPathSum(root.left,targetSum-root.val) || hasPathSum(root.right,targetSum - root.val);

};

**25. Maximum Depth of Binary Tree( Input:** root = [3,9,20,null,null,15,7]

**Output:** 3

**)**

***DFS***

**LOGIC C#**

public int MaxDepth(TreeNode root) {

if(root == null)

return 0;

if(root.left == null && root.right == null)

return 1;

int left = root.left != null ? MaxDepth(root.left) : int.MinValue;

int right= root. right!= null ? MaxDepth(root. right) : int.MinValue;

return Math.Max(left,right) +1;

}

**LOGIC JS**

var maxDepth = function(root) {

if(root == null)

return 0;

if(root.left == null && root.right == null)

return 1;

let left = root.left != null ? maxDepth(root.left) : Number.MIN\_SAFE\_INTEGER;

let right = root.right != null ? maxDepth(root.right) : Number.MIN\_SAFE\_INTEGER;

return Math.max(left, right) +1;

};

**26. Diameter of Binary Tree(Input:** root = [1,2,3,4,5]

**Output:** 3

**Explanation:** 3 is the length of the path [4,2,1,3] or [5,2,1,3].**)**

***DFS***

**LOGIC C#**

private int maxDiameter = 0;

public int DiameterOfBinaryTree(TreeNode root) {

DFS(root);

return maxDiameter;

}

private int DFS(TreeNode node)

{

if (node == null)

{

return 0;

}

int leftLength = DFS(node.left);

int rightLength =DFS(node.right);

var diameter = leftLength + rightLength;

maxDiameter = Math.Max(maxDiameter, diameter);

return Math.Max(leftLength, rightLength) +1;

}

**LOGIC JS**

var diameterOfBinaryTree = function(root) {

let max = 0;

getHeight(root);

return max;

function getHeight(root) {

if (root == null)

return 0;

let leftHeight = getHeight(root.left);

let rightHeight = getHeight(root.right);

let diameter = leftHeight + rightHeight;

max = Math.max(max,diameter);

return Math.max(leftHeight, rightHeight) + 1;

}

};

**27. Merge Two Binary Trees (Input:** root1 = [1,3,2,5], root2 = [2,1,3,null,4,null,7]

**Output:** [3,4,5,5,4,null,7]**)**

Public TreeNode Merge2Tree(TreeNode root1, TreeNode root2)

{

If(root1 == null && root2 == null)

Return null;

If(root1 == null )

Return root2;

If(root2 == null)

Return root1;

TreeNode data = root1.val + root2.val;

Let root = new TreeNode(data);

root.left = Merge2Tree(root1.left,root2.left);

root.right = Merge2Tree(root1.right, root2.right);

return root;

}

**LOGIC JS**

var mergeTrees = function(root1, root2) {

if(root1 == null && root2 == null)

return null;

if (root1 == null)

return root2;

if(root2 == null)

return root1;

let data = root1.val +root2.val;

let root= new TreeNode(data);

root.left = mergeTrees(root1.left, root2.left);

root.right = mergeTrees(root1.right, root2.right);

return root;

};

**28. Lowest Common Ancestor of a Binary Search Tree ( Input:** root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8

**Output:** 6

**)**

***DFS***

public TreeNode LowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

if(root == null)

return null;

if(p.val <root.val && q.val <root.val)

return LowestCommonAncestor(root.left,p,q);

if(p.val >root.val && q.val >root.val)

return LowestCommonAncestor(root.right,p,q);

return root;

}

**29. Subtree of Another Tree (Input:** root = [3,4,5,1,2], subRoot = [4,1,2]

**Output:** true

**)**

**LOGIC C#**

public bool IsSubtree(TreeNode root, TreeNode subRoot) {

if(root == null)

return false;

if(root.val == subRoot.val)

{

if(IsSame(root,subRoot))

return true;

}

return (IsSubtree(root.left, subRoot) || IsSubtree(root.right, subRoot));

}

private bool IsSame(TreeNode s, TreeNode t)

{

if(s == null && t == null)

return true;

if(s!=null && t==null || t!=null && s==null || s.val != t.val)

return false;

return (IsSame(s.left, t.left) && IsSame(s.right, t.right));

}

**LOGIC JS**

var isSubtree = function(root, subRoot) {

if(root == null)

return false;

if(root.val == subRoot.val)

{

if(IsSame(root,subRoot))

return true;

}

return isSubtree(root.left,subRoot) || isSubtree(root.right,subRoot);

};

function IsSame(s,t){

if(s == null && t == null)

return true;

if(s!=null && t==null || t!=null && s==null || s.val != t.val)

return false;

return (IsSame(s.left, t.left) && IsSame(s.right, t.right));

}

**30. Inver Tree (Input:** root = [4,2,7,1,3,6,9]

**Output:** [4,7,2,9,6,3,1]

**)**

***DFS***

**LOGIC C#**

public TreeNode InvertTree(TreeNode root) {

if (root == null)

return null;

var node = new TreeNode(root.val);

node.left = InvertTree(root.right);

node.right = InvertTree(root.left);

return node;

}

**LOGIC JS**

var invertTree = function(root) {

if(root == null)

return null;

let node = new TreeNode(root.val);

node.left = invertTree(root.right);

node.right = invertTree(root.left);

return node;

};

**31. Two Sum (Input:** nums = [2,7,11,15], target = 9

**Output:** [0,1]

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

**)**

***2 Pointers***

**LOGIC C#**

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

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

return new int[2];

}

Dictionary<int,int> dic = new Dictionary<int,int> ();

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

if(dic.ContainsKey(target-nums[i]))

return new int[] {i,dic[target- nums[i]]};

else if(!dic.ContainsKey(nums[i]))

dic.Add(nums[i],i);

}

return new int[2];

}

**LOGIC JS**

var twoSum = function(nums, target) {

let numberMap = new Map();

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

if (numberMap.has(target - nums[i]))

return [i, numberMap.get(target - nums[i])];

else

numberMap.set(nums[i], i);

}

return [];

};

**32. Sorted Squares Array (Input:** nums = [-4,-1,0,3,10]

**Output:** [0,1,9,16,100]

**)**

***Two pointers***

Public int[] sortedSquaresArray(int[] data)

{

Int l = 0;

Int r = nums.Length-1;

Int [] result = new int[nums.Length];

For(int I = 0; i<nums.Length; i++){

nums[i] = nums[i]\* nums[i];

}

For(int i = nums.Length-1; i> = 0; i--)

{

If(nums[l]>nums[r]){

Result[i] = nums[l];

L++;

}

Else{

Result[i] = nums[r];

r--;

}

Return result;

}

}

**LOGIC JS**

var sortedSquares = function(nums) {

let result = [];

let l = 0;

let r = nums.length-1;

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

nums[i] = nums[i] \* nums[i];

}

for(let i = nums.length-1; i>= 0; i--){

if(nums[l]> nums[r]){

result[i] = nums[l];

l++;

}

else{

result[i] = nums[r];

r--;

}

}

return result;

};

**33. Backspace String Compare ( Input:** s = "ab#c", t = "ad#c"

**Output:** true

**)**

***Two Pointers***

**LOGIC C#**

public bool BackspaceCompare(string s, string t) {

string data1 = string.Empty;

string data2 = string.Empty;

foreach(char c in s)

{

if (c != '#')

{

data1 += c.ToString();

}

else if (data1.Length > 0)

{

data1 = data1.Remove(data1.Length - 1);

}

}

foreach (char c in t)

{

if (c != '#')

{

data2 += c.ToString();

}

else if (data2.Length > 0)

{

data2 = data2.Remove(data2.Length - 1);

}

}

return data1 == data2;

}

**LOGIC JS**

var backspaceCompare = function(S, T) {

const backedS = backStr(S)

const backedT = backStr(T)

return backedS == backedT

};

function backStr(str) {

let res = "";

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

if (str.charAt(i) != '#') {

res += str.charAt(i);

} else if (res.length > 0) {

res = res.slice(0, -1);

}

}

return res

}

**34. Index pairs of a string(** Given a text string and words (a list of strings), return all index pairs [i, j] so that the substring text[i]...text[j] is in the list of words.

Input: text = "ababa", words = ["aba","ab"]

Output: [[0,1],[0,2],[2,3],[2,4]]

Explanation:

Notice that matches can overlap, see "aba" is found in [0,2] and[2, 4].**)**

***TRIE***

**LOGIC C#**

class Trie

{

class TrieNode

{

public bool isEnd;

public TrieNode[] children = new TrieNode[26];

public TrieNode()

{

isEnd = false;

for (int i = 0; i < 26; i++)

children[i] = null;

}

}

private TrieNode root;

public Trie()

{

root = new TrieNode();

}

public int[][] indexPairs(string text, string[] words)

{

/\*initializing tire and put all word from words into Trie.\*/

foreach (string s in words)

{

TrieNode cur = root;

foreach (char c in s)

{

if (cur.children[c - 'a'] == null)

{

cur.children[c - 'a'] = new TrieNode();

}

cur = cur.children[c - 'a'];

}

cur.isEnd = true; /\*mark there is a word\*/

}

/\*if text is "ababa", check "ababa","baba","aba","ba","a" individually.\*/

int len = text.Length;

List<int[]> list = new List<int[]>();

for (int l = 0; l < len; l++)

{

TrieNode cur = root;

char cc = text[l];

int j = l; /\*j is our moving index\*/

while (cur.children[cc - 'a'] != null)

{

cur = cur.children[cc - 'a'];

if (cur.isEnd)

{ /\*there is a word ending here, put into our list\*/

list.Add(new int[] { l, j });

}

j++;

if (j == len)

{ /\*reach the end of the text, we stop\*/

break;

}

else

{

cc = text[j];

}

}

}

/\*put all the pairs from list into array\*/

int size = list.Count;

int[][] res = new int[size][];

int i = 0;

foreach (int[] r in list)

{

res[i] = r;

i++;

}

return res;

}

static void Main(string[] args)

{

Trie trie = new Trie();

string[] words = { "aba" ,"ab"};

int[][] result = trie.indexPairs("ababa", words);

foreach (int[] data in result)

{

Console.Write("[");

for (int i = 0; i<data.Length; i++)

Console.Write(data[i] + " ");

Console.Write("]");

}

}

}

**LOGIC JS**

**TODO**

**35. Majority Element (Input:** nums = [3,2,3]

**Output:** 3

**)**

***Sorting***

**LOGIC C#**

public int MajorityElement(int[] nums) {

Dictionary<int,int> dict = new Dictionary<int,int>();

foreach(var K in nums){

if(!dict.ContainsKey(K))

dict.Add(K,0);

dict[K] +=1;

if(dict[K]> nums.Length/2)

return K;

}

return -1;

}

**LOGIC JS**

var majorityElement = function(nums) {

let map=new Map();

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

if(!map.has(nums[i]))

map.set(nums[i],1);

else

map.set(nums[i],map.get(nums[i])+1);

if(map.get(nums[i])> nums.length/2)

return nums[i];

}

return -1;

};

**36. Convert 1D Array Into 2D Array ( Input:** original = [1,2,3,4], m = 2, n = 2

**Output:** [[1,2],[3,4]]**)**

***Arrays***

**LOGIC C#**

public int[][] Construct2DArray(int[] original, int m, int n) {

if(m \* n != original.Length)

return new int[0][];

int[][] res = new int[m][];

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

res[i] = new int[n];

for(int i = 0; i < original.Length; i++)

{

res[i / n][i % n] = original[i];

}

return res;

}

**LOGIC Js**

var construct2DArray = function(original, m, n) {

if (m \* n !== original.length) {

return [];

}

let a = [];

for (let i = 0; i < m; i++) {

a.push(original.splice(0, n));

}

return a;

};