1) Two Sum (Easy)

Summary: Given an array and target, return indices of two numbers that add to target. Use a hash map for O(n).

Java:

```
class Solution {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer,Integer> map = new HashMap<>();
        for(int i=0;i<nums.length;i++){
            int need = target - nums[i];
            if(map.containsKey(need)) return new int[]{map.get(need),
i};
            map.put(nums[i], i);
        }
        return new int[]{-1,-1};
    }
}
```

---

2) Add Two Numbers (Medium)

Summary: Two numbers represented by linked lists (reverse order). Add digit-by-digit with carry.

Java:

```
class ListNode { int val; ListNode next; ListNode(int v){val=v;} }
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode dummy=new ListNode(0), cur=dummy;
        int carry=0;
        while(l1!=null || l2!=null || carry!=0){
            int s = carry + (l1!=null?l1.val:0) + (l2!=null?l2.val:0);
            carry = s/10;
            cur.next = new ListNode(s%10);
            cur = cur.next;
```

```
            if(l1!=null) l1 = l1.next;
            if(l2!=null) l2 = l2.next;
        }
        return dummy.next;
    }
}
```

---

3) Longest Substring Without Repeating Characters (Medium)

Summary: Sliding window with last-seen map to maintain unique substring, O(n).
Java:

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        int[] last = new int[128];
        Arrays.fill(last, -1);
        int start=0, max=0;
        for(int i=0;i<s.length();i++){
            char c = s.charAt(i);
            start = Math.max(start, last[c]+1);
            max = Math.max(max, i-start+1);
            last[c] = i;
        }
        return max;
    }
}
```

---

4) Median of Two Sorted Arrays (Hard)

Summary: Find median of two sorted arrays in O(log(min(m,n))) via binary search on partition.
Java:

```
class Solution {
    public double findMedianSortedArrays(int[] A, int[] B) {
        if(A.length > B.length) return findMedianSortedArrays(B, A);
        int m=A.length, n=B.length;
        int low=0, high=m;
        while(low<=high){
            int i = (low+high)/2;
            int j = (m+n+1)/2 - i;
            int Aleft = (i==0)? Integer.MIN_VALUE : A[i-1];
            int Aright= (i==m)? Integer.MAX_VALUE : A[i];
            int Bleft = (j==0)? Integer.MIN_VALUE : B[j-1];
            int Bright= (j==n)? Integer.MAX_VALUE : B[j];
            if(Aleft <= Bright && Bleft <= Aright){
                if((m+n)%2==1) return Math.max(Aleft, Bleft);
                return ( (double)Math.max(Aleft,Bleft) +
Math.min(Aright,Bright) )/2.0;
            } else if(Aleft > Bright) high = i-1;
            else low = i+1;
        }
        throw new IllegalArgumentException();
    }
}
```

---

5) Longest Palindromic Substring (Medium)

Summary: Expand-around-center for each center (O(n^2)), or Manacher's algorithm for O(n).
Java (expand):

```
class Solution {
    public String longestPalindrome(String s) {
        if(s==null||s.length()<1) return "";
        int start=0,end=0;
        for(int i=0;i<s.length();i++){
            int len1 = expand(s,i,i);
```

```
            int len2 = expand(s,i,i+1);
            int len = Math.max(len1,len2);
            if(len > end-start+1){
                start = i - (len-1)/2;
                end = i + len/2;
            }
        }
        return s.substring(start,end+1);
    }
    private int expand(String s,int l,int r){
        while(l>=0 && r<s.length() && s.charAt(l)==s.charAt(r)){ l--;
r++; }
        return r-l-1;
    }
}
```

---

6) Container With Most Water (Medium)

Summary: Two-pointer approach from both ends, move smaller pointer inward, O(n).

Java:

```
class Solution {
    public int maxArea(int[] height) {
        int l=0, r=height.length-1, max=0;
        while(l<r){
            int h = Math.min(height[l], height[r]);
            max = Math.max(max, h*(r-l));
            if(height[l] < height[r]) l++; else r--;
        }
        return max;
    }
}
```

---

7) Valid Parentheses (Easy)

Summary: Use a stack to match opening/closing brackets.
Java:

```
class Solution {
    public boolean isValid(String s) {
        Deque<Character> st = new ArrayDeque<>();
        for(char c: s.toCharArray()){
            if(c=='('||c=='['||c=='{') st.push(c);
            else {
                if(st.isEmpty()) return false;
                char t = st.pop();
                if((c==')' && t!='(') || (c==']' && t!='[') || (c=='}'
&& t!='{')) return false;
            }
        }
        return st.isEmpty();
    }
}
```

---

8) Merge Two Sorted Lists (Easy)

Summary: Merge two sorted linked lists iteratively using a dummy node.
Java:

```
class ListNode { int val; ListNode next; ListNode(int v){val=v;} }
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy=new ListNode(0), cur=dummy;
        while(l1!=null && l2!=null){
            if(l1.val < l2.val){ cur.next = l1; l1 = l1.next; }
            else { cur.next = l2; l2 = l2.next; }
            cur = cur.next;
        }
        cur.next = (l1!=null)?l1:l2;
```

```
        return dummy.next;
    }
}
```

---

9) Remove Nth Node From End of List (Medium)

Summary: Two-pointer (fast/slow) separated by n nodes; remove slow.next.
Java:

```
class ListNode { int val; ListNode next; ListNode(int v){val=v;} }
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode dummy=new ListNode(0); dummy.next = head;
        ListNode fast=dummy, slow=dummy;
        for(int i=0;i<n;i++) fast = fast.next;
        while(fast.next!=null){
            fast = fast.next; slow = slow.next;
        }
        slow.next = slow.next.next;
        return dummy.next;
    }
}
```

---

10) Reverse Linked List (Easy)

Summary: Iterative pointer reversal (prev/curr) in O(n), O(1) space.
Java:

```
class ListNode { int val; ListNode next; ListNode(int v){val=v;} }
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null, cur = head;
```

```
        while(cur!=null){
            ListNode nxt = cur.next;
            cur.next = prev;
            prev = cur;
            cur = nxt;
        }
        return prev;
    }
}
```

---

11) Linked List Cycle (Easy)

Summary: Detect cycle using Floyd's tortoise and hare (fast/slow pointers).
Java:

```
class ListNode { int val; ListNode next; ListNode(int v){val=v;} }
class Solution {
    public boolean hasCycle(ListNode head) {
        ListNode slow=head, fast=head;
        while(fast!=null && fast.next!=null){
            slow = slow.next;
            fast = fast.next.next;
            if(slow==fast) return true;
        }
        return false;
    }
}
```

---

12) Binary Tree Inorder Traversal (Medium)

Summary: Iterative using a stack or recursion. Iterative shown below.
Java:

```
class TreeNode { int val; TreeNode left,right; TreeNode(int v){val=v;}
}
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        Deque<TreeNode> st = new ArrayDeque<>();
        TreeNode cur = root;
        while(cur!=null || !st.isEmpty()){
            while(cur!=null){ st.push(cur); cur = cur.left; }
            cur = st.pop();
            res.add(cur.val);
            cur = cur.right;
        }
        return res;
    }
}
```

---

13) Binary Tree Level Order Traversal (Medium)

Summary: BFS with a queue, collecting nodes level by level.

Java:

```
class TreeNode { int val; TreeNode left,right; TreeNode(int v){val=v;}
}
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<>();
        if(root==null) return res;
        Queue<TreeNode> q = new ArrayDeque<>();
        q.add(root);
        while(!q.isEmpty()){
            int sz = q.size();
            List<Integer> level = new ArrayList<>();
            for(int i=0;i<sz;i++){
```

```
                TreeNode n = q.poll();
                level.add(n.val);
                if(n.left!=null) q.add(n.left);
                if(n.right!=null) q.add(n.right);
            }
            res.add(level);
        }
        return res;
    }
}
```

---

14) Validate Binary Search Tree (Medium)

Summary: Inorder traversal should produce strictly increasing sequence; or validate recursively with min/max.

Java (inorder):

```
class TreeNode { int val; TreeNode left,right; TreeNode(int v){val=v;}
}
class Solution {
    private Integer prev = null;
    public boolean isValidBST(TreeNode root) {
        prev = null;
        return inorder(root);
    }
    private boolean inorder(TreeNode node){
        if(node==null) return true;
        if(!inorder(node.left)) return false;
        if(prev!=null && node.val <= prev) return false;
        prev = node.val;
        return inorder(node.right);
    }
}
```

---

15) Maximum Depth of Binary Tree (Easy)

Summary: DFS recursion to compute 1 + max(leftDepth,rightDepth).
Java:

```
class TreeNode { int val; TreeNode left,right; TreeNode(int v){val=v;}
}
class Solution {
    public int maxDepth(TreeNode root) {
        if(root==null) return 0;
        return 1 + Math.max(maxDepth(root.left),
maxDepth(root.right));
    }
}
```

---

16) Serialize and Deserialize Binary Tree (Hard)

Summary: Convert tree to a string (e.g., BFS with null markers) and parse back.
Java (BFS):

```
class TreeNode { int val; TreeNode left,right; TreeNode(int v){val=v;}
}
class Codec {
    public String serialize(TreeNode root) {
        if(root==null) return "";
        StringBuilder sb = new StringBuilder();
        Queue<TreeNode> q = new LinkedList<>();
        q.add(root);
        while(!q.isEmpty()){
            TreeNode n = q.poll();
            if(n==null) sb.append("n,");
            else { sb.append(n.val).append(','); q.add(n.left);
q.add(n.right); }
        }
```

```
        return sb.toString();
    }
    public TreeNode deserialize(String data) {
        if(data==null||data.length()==0) return null;
        String[] parts = data.split(",");
        Queue<TreeNode> q = new LinkedList<>();
        TreeNode root = new TreeNode(Integer.parseInt(parts[0]));
        q.add(root);
        int i=1;
        while(!q.isEmpty() && i<parts.length){
            TreeNode node = q.poll();
            if(!parts[i].equals("n")){ node.left = new
TreeNode(Integer.parseInt(parts[i])); q.add(node.left); }
            i++;
            if(i<parts.length && !parts[i].equals("n")){ node.right =
new TreeNode(Integer.parseInt(parts[i])); q.add(node.right); }
            i++;
        }
        return root;
    }
}
```

---

17) Number of Islands (Medium)

Summary: Count connected components of '1's in grid using DFS/BFS.
Java (DFS):

```
class Solution {
    public int numIslands(char[][] grid) {
        if(grid==null||grid.length==0) return 0;
        int m=grid.length, n=grid[0].length, cnt=0;
        for(int i=0;i<m;i++) for(int j=0;j<n;j++){
            if(grid[i][j]=='1'){ cnt++; dfs(grid,i,j); }
        }
        return cnt;
    }
```

```
    private void dfs(char[][] g,int i,int j){
        if(i<0||j<0||i>=g.length||j>=g[0].length||g[i][j]=='0')
return;
        g[i][j] = '0';
        dfs(g,i+1,j); dfs(g,i-1,j); dfs(g,i,j+1); dfs(g,i,j-1);
    }
}
```

---

18) Course Schedule (207) (Medium)

Summary: Detect if all courses can be finished given prerequisites (directed graph); use topological sort (Kahn) or DFS cycle detection.

Java (Kahn):

```
class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        List<List<Integer>> g = new ArrayList<>();
        for(int i=0;i<numCourses;i++) g.add(new ArrayList<>());
        int[] indeg = new int[numCourses];
        for(int[] p: prerequisites){ g.get(p[1]).add(p[0]);
indeg[p[0]]++; }
        Queue<Integer> q = new ArrayDeque<>();
        for(int i=0;i<numCourses;i++) if(indeg[i]==0) q.add(i);
        int seen=0;
        while(!q.isEmpty()){
            int u=q.poll(); seen++;
            for(int v: g.get(u)) if(--indeg[v]==0) q.add(v);
        }
        return seen==numCourses;
    }
}
```

---

19) LRU Cache (Medium)

Summary: Design cache with O(1) get/put — use LinkedHashMap or Doubly Linked List + HashMap.

Java (LinkedHashMap simple):

```
class LRUCache extends LinkedHashMap<Integer,Integer> {
    private int capacity;
    public LRUCache(int capacity){
        super(capacity, 0.75f, true);
        this.capacity = capacity;
    }
    public int get(int key){
        return super.getOrDefault(key, -1);
    }
    public void put(int key, int value){
        super.put(key, value);
    }
    @Override
    protected boolean removeEldestEntry(Map.Entry<Integer,Integer>
eldest){
        return size() > capacity;
    }
}
```

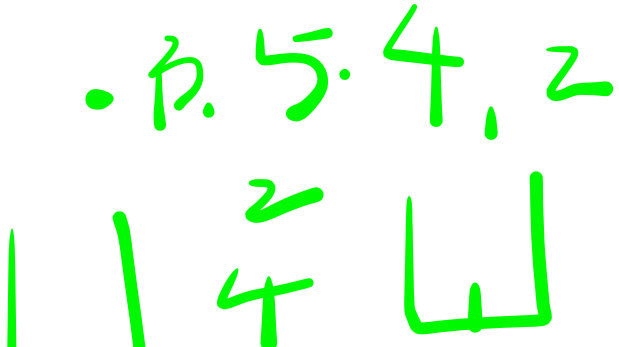
---

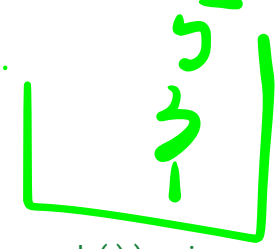
20) Min Stack (Easy)

Summary: Stack supporting push/pop/top/getMin in O(1) — implement with an extra stack of mins.

Java:

```
class MinStack {
    private Deque<Integer> stack;
    private Deque<Integer> mins;
    public MinStack() {
        stack = new ArrayDeque<>();
        mins = new ArrayDeque<>();
    }
    public void push(int val) {
```





```
        stack.push(val);
        if(mins.isEmpty() || val <= mins.peek()) mins.push(val);
    }
    public void pop() {
        int v = stack.pop();
        if(v == mins.peek()) mins.pop();
    }
    public int top() { return stack.peek(); }
    public int getMin() { return mins.peek(); }
}
```

---

21) Top K Frequent Elements (Medium)

Summary: Return k most frequent elements — use frequency map + min-heap of size k or bucket sort.

Java (heap):

```
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        Map<Integer,Integer> freq = new HashMap<>();
        for(int n: nums) freq.put(n, freq.getOrDefault(n,0)+1);
        PriorityQueue<Integer> pq = new
PriorityQueue<>((a,b)->freq.get(a)-freq.get(b));
        for(int key: freq.keySet()){
            pq.offer(key);
            if(pq.size()>k) pq.poll();
        }
        int[] res = new int[k];
        for(int i=k-1;i>=0;i--) res[i]=pq.poll();
        return res;
    }
}
```

---

22) Merge Intervals (Medium)

Summary: Sort intervals by start, then merge overlapping ones into a result list.

Java:

```
class Solution {
    public int[][] merge(int[][] intervals) {
        if(intervals.length==0) return new int[0][];
        Arrays.sort(intervals, (a,b) -> a[0]-b[0]);
        List<int[]> res = new ArrayList<>();
        int[] cur = intervals[0];
        for(int i=1;i<intervals.length;i++){
            if(intervals[i][0] <= cur[1]) cur[1] = Math.max(cur[1],
intervals[i][1]);
            else { res.add(cur); cur = intervals[i]; }
        }
        res.add(cur);
        return res.toArray(new int[res.size()][]);
    }
}
```

---

23) Search in Rotated Sorted Array (Medium)

Summary: Modified binary search: compare mid to edges to decide which half is sorted, then decide direction.

Java:

```
class Solution {
    public int search(int[] nums, int target) {
        int l=0, r=nums.length-1;
        while(l<=r){
            int m = (l+r)/2;
            if(nums[m]==target) return m;
            if(nums[l] <= nums[m]) { // left sorted
                if(target >= nums[l] && target < nums[m]) r = m-1;
                else l = m+1;
            } else { // right sorted
```

```
                if(target > nums[m] && target <= nums[r]) l = m+1;
                else r = m-1;
            }
        }
        return -1;
    }
}
```

---

24) Kth Largest Element in an Array (Medium)

Summary: Quickselect average O(n) or use a min-heap of size k.

Java (Quickselect):

```
class Solution {
    private Random rand = new Random();
    public int findKthLargest(int[] nums, int k) {
        return quickselect(nums, 0, nums.length-1, nums.length-k);
    }
    private int quickselect(int[] a,int l,int r,int kSmall){
        if(l==r) return a[l];
        int pivotIdx = l + rand.nextInt(r-l+1);
        pivotIdx = partition(a, l, r, pivotIdx);
        if(kSmall == pivotIdx) return a[kSmall];
        else if(kSmall < pivotIdx) return quickselect(a, l,
pivotIdx-1, kSmall);
        else return quickselect(a, pivotIdx+1, r, kSmall);
    }
    private int partition(int[] a,int l,int r,int pivotIdx){
        int pivot = a[pivotIdx];
        swap(a, pivotIdx, r);
        int store = l;
        for(int i=l;i<r;i++){
            if(a[i] < pivot){ swap(a, store, i); store++; }
        }
```

```
        swap(a, store, r);
        return store;
    }
    private void swap(int[] a,int i,int j){ int t=a[i]; a[i]=a[j];
a[j]=t; }
}
```

---

25) Word Break (139) (Medium)

Summary: Given string and dictionary, determine if it can be segmented — DP where dp[i] true if s[0..i) can be formed.

Java:

```
class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        Set<String> dict = new HashSet<>(wordDict);
        boolean[] dp = new boolean[s.length()+1];
        dp[0] = true;
        for(int i=1;i<=s.length();i++){
            for(int j=0;j<i;j++){
                if(dp[j] && dict.contains(s.substring(j,i))){
dp[i]=true; break; }
            }
        }
        return dp[s.length()];
    }
}
```

---

26) Word Search (79) (Medium)

Summary: Backtracking DFS from each cell, marking visited, check if word exists.

Java:

```
class Solution {
    public boolean exist(char[][] board, String word) {
        int m=board.length, n=board[0].length;
        for(int i=0;i<m;i++)
            for(int j=0;j<n;j++)
                if(dfs(board,word,i,j,0)) return true;
        return false;
    }
    private boolean dfs(char[][] b,String w,int i,int j,int k){
        if(k==w.length()) return true;

if(i<0||j<0||i>=b.length||j>=b[0].length||b[i][j]!=w.charAt(k)) return
false;
        char tmp = b[i][j]; b[i][j] = '#';
        boolean found = dfs(b,w,i+1,j,k+1) || dfs(b,w,i-1,j,k+1) ||
                        dfs(b,w,i,j+1,k+1) || dfs(b,w,i,j-1,k+1);
        b[i][j] = tmp;
        return found;
    }
}
```

---

27) Combination Sum (39) (Medium)

Summary: Backtracking — try each candidate repeatedly, build valid sums equal to target.

Java:

```
class Solution {
    public List<List<Integer>> combinationSum(int[] candidates, int
target) {
        List<List<Integer>> res = new ArrayList<>();
        backtrack(res, new ArrayList<>(), candidates, target, 0);
        return res;
    }
    private void backtrack(List<List<Integer>> res, List<Integer> cur,
int[] c, int remain, int start){
        if(remain==0){ res.add(new ArrayList<>(cur)); return; }
```

```
        for(int i=start;i<c.length;i++){
            if(c[i] > remain) continue;
            cur.add(c[i]);
            backtrack(res,cur,c,remain-c[i],i);
            cur.remove(cur.size()-1);
        }
    }
}
```

---

28) Subsets (78) (Medium)

Summary: Generate all subsets using backtracking or bitmasks.
Java (backtracking):

```
class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        backtrack(res, new ArrayList<>(), nums, 0);
        return res;
    }
    private void backtrack(List<List<Integer>> res, List<Integer> cur,
int[] nums, int start){
        res.add(new ArrayList<>(cur));
        for(int i=start;i<nums.length;i++){
            cur.add(nums[i]);
            backtrack(res,cur,nums,i+1);
            cur.remove(cur.size()-1);
        }
    }
}
```

---

29) Permutations (46) (Medium)

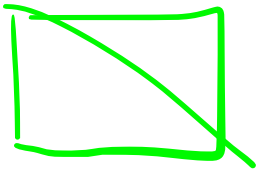
Summary: Backtracking — build all permutations using used[] flag.
Java:

```
class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> res = new ArrayList<>();
        boolean[] used = new boolean[nums.length];
        backtrack(res,new ArrayList<>(),nums,used);
        return res;
    }
    private void backtrack(List<List<Integer>> res,List<Integer>
cur,int[] nums,boolean[] used){
        if(cur.size()==nums.length){ res.add(new ArrayList<>(cur));
return; }
        for(int i=0;i<nums.length;i++){
            if(used[i]) continue;
            used[i]=true; cur.add(nums[i]);
            backtrack(res,cur,nums,used);
            cur.remove(cur.size()-1); used[i]=false;
        }
    }
}
```

---

30) Rotate Image (48) (Medium)

Summary: In-place matrix rotation by transpose + reverse each row.
Java:



```
class Solution {
    public void rotate(int[][] m) {
        int n=m.length;
        for(int i=0;i<n;i++)
            for(int j=i;j<n;j++){
                int t=m[i][j]; m[i][j]=m[j][i]; m[j][i]=t;
            }
        for(int i=0;i<n;i++)
            for(int j=0;j<n/2;j++){
```

```
                int t=m[i][j]; m[i][j]=m[i][n-1-j]; m[i][n-1-j]=t;
            }
    }
}
```

---

31) Spiral Matrix (54) (Medium)

Summary: Traverse matrix in spiral order by controlling boundaries.

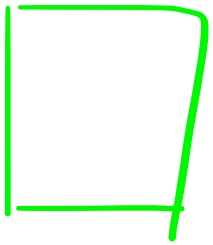
Java:

```
class Solution {
    public List<Integer> spiralOrder(int[][] m) {
        List<Integer> res=new ArrayList<>();
        if(m.length==0) return res;
        int top=0,bottom=m.length-1,left=0,right=m[0].length-1;
        while(top<=bottom && left<=right){
            for(int j=left;j<=right;j++) res.add(m[top][j]);
            top++;
            for(int i=top;i<=bottom;i++) res.add(m[i][right]);
            right--;
            if(top<=bottom){
                for(int j=right;j>=left;j--) res.add(m[bottom][j]);
                bottom--;
            }
            if(left<=right){
                for(int i=bottom;i>=top;i--) res.add(m[i][left]);
                left++;
            }
        }
        return res;
    }
}
```

---

32) Set Matrix Zeroes (73) (Medium)

Summary: Use first row/col as markers to set rows/cols to zero in-place.



Java:

```
class Solution {
    public void setZeroes(int[][] m) {
        int R=m.length, C=m[0].length;
        boolean fr=false, fc=false;
        for(int i=0;i<R;i++) if(m[i][0]==0) fc=true;
        for(int j=0;j<C;j++) if(m[0][j]==0) fr=true;
        for(int i=1;i<R;i++)
            for(int j=1;j<C;j++)
                if(m[i][j]==0){ m[i][0]=0; m[0][j]=0; }
        for(int i=1;i<R;i++)
            for(int j=1;j<C;j++)
                if(m[i][0]==0||m[0][j]==0) m[i][j]=0;
        if(fc) for(int i=0;i<R;i++) m[i][0]=0;
        if(fr) for(int j=0;j<C;j++) m[0][j]=0;
    }
}
```

---

33) Search a 2D Matrix (74) (Medium)

Summary: Treat as sorted 1D array and binary search.

Java:

```
class Solution {
    public boolean searchMatrix(int[][] m, int target) {
        int R=m.length, C=m[0].length;
        int l=0, r=R*C-1;
        while(l<=r){
            int mid=(l+r)/2;
            int val=m[mid/C][mid%C];
            if(val==target) return true;
            else if(val<target) l=mid+1;
```

```
            else r=mid-1;
        }
        return false;
    }
}
```

---

34) Trapping Rain Water (42) (Hard)

Summary: Two-pointer method tracking leftMax/rightMax, O(n).
Java:

```
class Solution {
    public int trap(int[] h) {
        int l=0,r=h.length-1,lm=0,rm=0,res=0;
        while(l<r){
            if(h[l]<h[r]){
                if(h[l]>=lm) lm=h[l]; else res+=lm-h[l];
                l++;
            } else {
                if(h[r]>=rm) rm=h[r]; else res+=rm-h[r];
                r--;
            }
        }
        return res;
    }
}
```

---

35) Valid Sudoku (36) (Medium)

Summary: Check row, column, and 3×3 boxes using sets/boolean arrays.
Java:

```
class Solution {
    public boolean isValidSudoku(char[][] b) {
```

```
        boolean[][] rows=new boolean[9][9], cols=new boolean[9][9],
boxes=new boolean[9][9];
        for(int i=0;i<9;i++){
            for(int j=0;j<9;j++){
                if(b[i][j]=='.') continue;
                int d=b[i][j]-'1';
                int box=(i/3)*3+j/3;
                if(rows[i][d]||cols[j][d]||boxes[box][d]) return
false;
                rows[i][d]=cols[j][d]=boxes[box][d]=true;
            }
        }
        return true;
    }
}
```

---

36) Rotate List (61) (Medium)

Summary: Rotate linked list right by k — connect tail to head, break at new head.

Java:

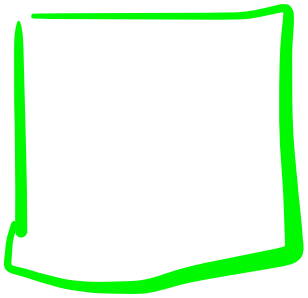
```
class ListNode { int val; ListNode next; ListNode(int v){val=v;} }
class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        if(head==null) return null;
        int n=1; ListNode tail=head;
        while(tail.next!=null){ n++; tail=tail.next; }
        k%=n;
        if(k==0) return head;
        tail.next=head; // make circle
        ListNode newTail=head;
        for(int i=0;i<n-k-1;i++) newTail=newTail.next;
        ListNode newHead=newTail.next;
        newTail.next=null;
        return newHead;
    }
```

```
}
```

---

37) Unique Paths (62) (Medium)

Summary: DP grid counting ways from top-left to bottom-right.
Java:



```
class Solution {
    public int uniquePaths(int m, int n) {
        int[][] dp=new int[m][n];
        for(int i=0;i<m;i++) dp[i][0]=1;
        for(int j=0;j<n;j++) dp[0][j]=1;
        for(int i=1;i<m;i++)
            for(int j=1;j<n;j++)
                dp[i][j]=dp[i-1][j]+dp[i][j-1];
        return dp[m-1][n-1];
    }
}
```

---

38) Climbing Stairs (70) (Easy)

Summary: Fibonacci DP — ways[n]=ways[n-1]+ways[n-2].
Java:

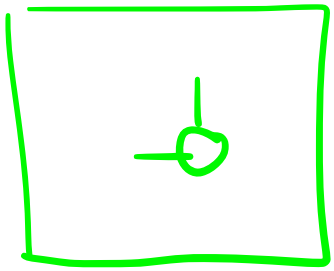
```
class Solution {
    public int climbStairs(int n) {
        if(n<=2) return n;
        int a=1,b=2;
        for(int i=3;i<=n;i++){
            int c=a+b; a=b; b=c;
        }
        return b;
    }
}
```

---

39) Minimum Path Sum (64) (Medium)

Summary: DP — each cell min sum = grid[i][j]+min(top,left).

Java:



```
class Solution {
    public int minPathSum(int[][] g) {
        int m=g.length,n=g[0].length;
        for(int i=0;i<m;i++){
            for(int j=0;j<n;j++){
                if(i==0&&j==0) continue;
                if(i==0) g[i][j]+=g[i][j-1];
                else if(j==0) g[i][j]+=g[i-1][j];
                else g[i][j]+=Math.min(g[i-1][j], g[i][j-1]);
            }
        }
        return g[m-1][n-1];
    }
}
```

---

40) Edit Distance (72) (Hard)

Summary: DP with operations insert/delete/replace.

Java:

```
class Solution {
    public int minDistance(String w1, String w2) {
        int m=w1.length(),n=w2.length();
        int[][] dp=new int[m+1][n+1];
        for(int i=0;i<=m;i++) dp[i][0]=i;
        for(int j=0;j<=n;j++) dp[0][j]=j;
        for(int i=1;i<=m;i++)
            for(int j=1;j<=n;j++){
```

```
                if(w1.charAt(i-1)==w2.charAt(j-1))
dp[i][j]=dp[i-1][j-1];
                else dp[i][j]=1+Math.min(dp[i-1][j-1],
Math.min(dp[i-1][j], dp[i][j-1]));
            }
        return dp[m][n];
    }
}
```

---

41) Maximum Subarray (53) (Easy)

Summary: Kadane's algorithm — max sum ending here, track max overall.
Java:

```
class Solution {
    public int maxSubArray(int[] nums) {
        int max=nums[0], cur=nums[0];
        for(int i=1;i<nums.length;i++){
            cur=Math.max(nums[i],cur+nums[i]);
            max=Math.max(max,cur);
        }
        return max;
    }
}
```

---

42) Jump Game (55) (Medium)

Summary: Greedy — track farthest reachable index, check if >= last.
Java:

```
class Solution {
    public boolean canJump(int[] nums) {
        int reach=0;
        for(int i=0;i<nums.length;i++){
```

```
            if(i>reach) return false;
            reach=Math.max(reach,i+nums[i]);
        }
        return true;
    }
}
```

---

43) Merge Sorted Array (88) (Easy)

Summary: Merge from the back to avoid overwriting in nums1.
Java:

```
class Solution {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int i=m-1,j=n-1,k=m+n-1;
        while(i>=0 && j>=0){
            if(nums1[i]>nums2[j]) nums1[k--]=nums1[i--];
            else nums1[k--]=nums2[j--];
        }
        while(j>=0) nums1[k--]=nums2[j--];
    }
}
```

---

44) Binary Search (704) (Easy)

Summary: Standard iterative binary search.
Java:

```
class Solution {
    public int search(int[] nums, int target) {
        int l=0,r=nums.length-1;
        while(l<=r){
            int m=(l+r)/2;
            if(nums[m]==target) return m;
```

```
            else if(nums[m]<target) l=m+1;
            else r=m-1;
        }
        return -1;
    }
}
```

---

45) First Bad Version (278) (Easy)

Summary: Binary search using API `isBadVersion`.

Java:

```
public class Solution extends VersionControl {
    public int firstBadVersion(int n) {
        int l=1,r=n;
        while(l<r){
            int m=l+(r-l)/2;
            if(isBadVersion(m)) r=m;
            else l=m+1;
        }
        return l;
    }
}
```

---

46) Find Peak Element (162) (Medium)

Summary: Binary search — move to side with greater neighbor.

Java:

```
class Solution {
    public int findPeakElement(int[] nums) {
        int l=0,r=nums.length-1;
        while(l<r){
            int m=(l+r)/2;
```

```
            if(nums[m]>nums[m+1]) r=m;
            else l=m+1;
        }
        return l;
    }
}
```

---

47) Search Insert Position (35) (Easy)

Summary: Binary search to find insertion point.
Java:

```
class Solution {
    public int searchInsert(int[] nums, int target) {
        int l=0,r=nums.length-1;
        while(l<=r){
            int m=(l+r)/2;
            if(nums[m]==target) return m;
            else if(nums[m]<target) l=m+1;
            else r=m-1;
        }
        return l;
    }
}
```

---

48) Pow(x, n) (50) (Medium)

Summary: Fast exponentiation (binary exponentiation).
Java:

```
class Solution {
    public double myPow(double x, int n) {
        long N=n;
        if(N<0){ x=1/x; N=-N; }
```

```
        double res=1;
        while(N>0){
            if((N&1)==1) res*=x;
            x*=x;
            N>>=1;
        }
        return res;
    }
}
```

---

49) Sqrt(x) (69) (Easy)

Summary: Binary search for sqrt, or Newton's method.
Java (binary search):

```
class Solution {
    public int mySqrt(int x) {
        if(x<2) return x;
        int l=1,r=x/2,ans=0;
        while(l<=r){
            int m=l+(r-l)/2;
            if((long)m*m <= x){ ans=m; l=m+1; }
            else r=m-1;
        }
        return ans;
    }
}
```

---

50) Find Minimum in Rotated Sorted Array (153) (Medium)

Summary: Binary search for inflection point.
Java:

```java
class Solution {
    public int findMin(int[] nums) {
        int l=0,r=nums.length-1;
        while(l<r){
            int m=(l+r)/2;
            if(nums[m]>nums[r]) l=m+1;
            else r=m;
        }
        return nums[l];
    }
}
```

51. Merge Intervals

Summary: Given a collection of intervals, merge all overlapping intervals.

```java
class Solution {
    public int[][] merge(int[][] intervals) {
        if (intervals.length <= 1) return intervals;
        Arrays.sort(intervals, (a,b) -> a[0]-b[0]);
        List<int[]> result = new ArrayList<>();
        int[] current = intervals[0];
        for (int i = 1; i < intervals.length; i++) {
            if (current[1] >= intervals[i][0]) {
                current[1] = Math.max(current[1], intervals[i][1]);
            } else {
                result.add(current);
                current = intervals[i];
            }
        }
        result.add(current);
        return result.toArray(new int[result.size()][]);
    }
}
```

---

52. Insert Interval

Summary: Insert a new interval into a list of non-overlapping intervals and merge if necessary.

```
class Solution {
    public int[][] insert(int[][] intervals, int[] newInterval) {
        List<int[]> result = new ArrayList<>();
        int i = 0;
        while (i < intervals.length && intervals[i][1] <
newInterval[0]) {
            result.add(intervals[i++]);
        }
        while (i < intervals.length && intervals[i][0] <=
newInterval[1]) {
            newInterval[0] = Math.min(newInterval[0],
intervals[i][0]);
            newInterval[1] = Math.max(newInterval[1],
intervals[i][1]);
            i++;
        }
        result.add(newInterval);
        while (i < intervals.length) result.add(intervals[i++]);
        return result.toArray(new int[result.size()][]);
    }
}
```

---

53. Unique Paths

Summary: Find number of unique paths in an `m x n` grid.

```
class Solution {
    public int uniquePaths(int m, int n) {
        int[][] dp = new int[m][n];
        for (int i=0;i<m;i++) dp[i][0]=1;
        for (int j=0;j<n;j++) dp[0][j]=1;
        for (int i=1;i<m;i++) {
            for (int j=1;j<n;j++) {
                dp[i][j] = dp[i-1][j] + dp[i][j-1];
            }
        }
```

```
        return dp[m-1][n-1];
    }
}
```

---

54. Minimum Path Sum

Summary: Find a path from top-left to bottom-right which minimizes the sum.

```
class Solution {
    public int minPathSum(int[][] grid) {
        int m = grid.length, n = grid[0].length;
        int[][] dp = new int[m][n];
        dp[0][0] = grid[0][0];
        for (int i=1;i<m;i++) dp[i][0]=dp[i-1][0]+grid[i][0];
        for (int j=1;j<n;j++) dp[0][j]=dp[0][j-1]+grid[0][j];
        for (int i=1;i<m;i++) {
            for (int j=1;j<n;j++) {
                dp[i][j] = grid[i][j] + Math.min(dp[i-1][j],
dp[i][j-1]);
            }
        }
        return dp[m-1][n-1];
    }
}
```

---

55. Climbing Stairs

Summary: Number of ways to climb n stairs with 1 or 2 steps at a time.

```
class Solution {
    public int climbStairs(int n) {
        if (n <= 2) return n;
        int a=1, b=2;
        for (int i=3;i<=n;i++) {
            int temp = a+b;
            a=b;
```

```
            b=temp;
        }
        return b;
    }
}
```

---

56. Edit Distance

Summary: Minimum operations to convert word1 → word2 (insert, delete, replace).

```
class Solution {
    public int minDistance(String word1, String word2) {
        int m = word1.length(), n = word2.length();
        int[][] dp = new int[m+1][n+1];
        for (int i=0;i<=m;i++) dp[i][0]=i;
        for (int j=0;j<=n;j++) dp[0][j]=j;
        for (int i=1;i<=m;i++) {
            for (int j=1;j<=n;j++) {
                if (word1.charAt(i-1)==word2.charAt(j-1)) {
                    dp[i][j]=dp[i-1][j-1];
                } else {
                    dp[i][j]=1+Math.min(dp[i-1][j-1],
Math.min(dp[i-1][j], dp[i][j-1]));
                }
            }
        }
        return dp[m][n];
    }
}
```

---

57. Set Matrix Zeroes

Summary: If an element is 0, set its row & column to 0 in-place.

```
class Solution {
    public void setZeroes(int[][] matrix) {
        int m=matrix.length, n=matrix[0].length;
```

```
        boolean firstRow=false, firstCol=false;
        for (int i=0;i<m;i++) if (matrix[i][0]==0) firstCol=true;
        for (int j=0;j<n;j++) if (matrix[0][j]==0) firstRow=true;
        for (int i=1;i<m;i++) {
            for (int j=1;j<n;j++) {
                if (matrix[i][j]==0) {
                    matrix[i][0]=0;
                    matrix[0][j]=0;
                }
            }
        }
        for (int i=1;i<m;i++) {
            for (int j=1;j<n;j++) {
                if (matrix[i][0]==0 || matrix[0][j]==0)
matrix[i][j]=0;
            }
        }
        if (firstCol) for (int i=0;i<m;i++) matrix[i][0]=0;
        if (firstRow) for (int j=0;j<n;j++) matrix[0][j]=0;
    }
}
```

---

58. Search a 2D Matrix

Summary: Search for a target in a matrix where rows and cols are sorted.

```
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        int m = matrix.length, n = matrix[0].length;
        int left=0, right=m*n-1;
        while (left<=right) {
            int mid=(left+right)/2;
            int val=matrix[mid/n][mid%n];
            if (val==target) return true;
            else if (val<target) left=mid+1;
            else right=mid-1;
        }
        return false;
```

```
    }
}
```

---

59. Sort Colors

Summary: Sort an array with values 0, 1, 2 (Dutch National Flag).

```
class Solution {
    public void sortColors(int[] nums) {
        int low=0, mid=0, high=nums.length-1;
        while (mid<=high) {
            if (nums[mid]==0) {
                int tmp=nums[low]; nums[low]=nums[mid]; nums[mid]=tmp;
                low++; mid++;
            } else if (nums[mid]==1) {
                mid++;
            } else {
                int tmp=nums[mid]; nums[mid]=nums[high];
nums[high]=tmp;
                high--;
            }
        }
    }
}
```

---

60. Subsets

Summary: Generate all subsets (the power set).

```
class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(result, new ArrayList<>(), nums, 0);
        return result;
    }
    private void backtrack(List<List<Integer>> res, List<Integer>
temp, int[] nums, int start) {
```

```
        res.add(new ArrayList<>(temp));
        for (int i=start; i<nums.length; i++) {
            temp.add(nums[i]);
            backtrack(res, temp, nums, i+1);
            temp.remove(temp.size()-1);
        }
    }
}
```

---

61. Subsets II

Summary: Like subsets, but may contain duplicates — return unique subsets.

```
class Solution {
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        Arrays.sort(nums);
        List<List<Integer>> result = new ArrayList<>();
        backtrack(result, new ArrayList<>(), nums, 0);
        return result;
    }
    private void backtrack(List<List<Integer>> res, List<Integer>
temp, int[] nums, int start) {
        res.add(new ArrayList<>(temp));
        for (int i=start;i<nums.length;i++) {
            if (i>start && nums[i]==nums[i-1]) continue;
            temp.add(nums[i]);
            backtrack(res, temp, nums, i+1);
            temp.remove(temp.size()-1);
        }
    }
}
```

---

62. Word Search

Summary: Check if word exists in grid via backtracking.

```
class Solution {
```

```
    public boolean exist(char[][] board, String word) {
        for (int i=0;i<board.length;i++) {
            for (int j=0;j<board[0].length;j++) {
                if (dfs(board, word, i, j, 0)) return true;
            }
        }
        return false;
    }
    private boolean dfs(char[][] board, String word, int i, int j, int
idx) {
        if (idx==word.length()) return true;
        if (i<0 || i>=board.length || j<0 || j>=board[0].length ||
board[i][j]!=word.charAt(idx)) return false;
        char temp=board[i][j];
        board[i][j]='#';
        boolean found = dfs(board, word, i+1,j,idx+1) || dfs(board,
word, i-1,j,idx+1) ||
                        dfs(board, word, i,j+1,idx+1) || dfs(board,
word, i,j-1,idx+1);
        board[i][j]=temp;
        return found;
    }
}
```

63. Unique Paths II

- **Summary:** A robot is located at the top-left corner of an `m x n` grid. Some cells are blocked by obstacles. Count how many unique paths exist from start to finish.
- **Key Idea:** Dynamic Programming with obstacle checks.

```
class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        int m = obstacleGrid.length, n = obstacleGrid[0].length;
        int[][] dp = new int[m][n];
        if (obstacleGrid[0][0] == 1) return 0;
        dp[0][0] = 1;
```

```
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (obstacleGrid[i][j] == 1) {
                    dp[i][j] = 0;
                } else {
                    if (i > 0) dp[i][j] += dp[i-1][j];
                    if (j > 0) dp[i][j] += dp[i][j-1];
                }
            }
        }
        return dp[m-1][n-1];
    }
}
```

---

64. Minimum Path Sum

- **Summary:** Find the path from top-left to bottom-right of a grid that minimizes the sum of values.

- **Key Idea:** Dynamic Programming to accumulate min path sum.

```
class Solution {
    public int minPathSum(int[][] grid) {
        int m = grid.length, n = grid[0].length;
        int[][] dp = new int[m][n];
        dp[0][0] = grid[0][0];
        for (int i = 1; i < m; i++) dp[i][0] = dp[i-1][0] +
grid[i][0];
        for (int j = 1; j < n; j++) dp[0][j] = dp[0][j-1] +
grid[0][j];
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                dp[i][j] = Math.min(dp[i-1][j], dp[i][j-1]) +
grid[i][j];
            }
        }
        return dp[m-1][n-1];
```

```
    }
}
```

---

70. Climbing Stairs

- **Summary:** Count ways to climb `n` stairs where each step can be 1 or 2.
- **Key Idea:** Fibonacci sequence (DP).

```
class Solution {
    public int climbStairs(int n) {
        if (n <= 2) return n;
        int a = 1, b = 2;
        for (int i = 3; i <= n; i++) {
            int c = a + b;
            a = b;
            b = c;
        }
        return b;
    }
}
```

---

72. Edit Distance

- **Summary:** Find minimum number of operations (insert, delete, replace) to convert word1 → word2.
- **Key Idea:** DP with states `dp[i][j]` = edit distance for `word1[0..i-1]` and `word2[0..j-1]`.

```
class Solution {
    public int minDistance(String word1, String word2) {
        int m = word1.length(), n = word2.length();
        int[][] dp = new int[m+1][n+1];
        for (int i = 0; i <= m; i++) dp[i][0] = i;
```

```
        for (int j = 0; j <= n; j++) dp[0][j] = j;
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (word1.charAt(i-1) == word2.charAt(j-1)) {
                    dp[i][j] = dp[i-1][j-1];
                } else {
                    dp[i][j] = 1 + Math.min(dp[i-1][j-1],
                                    Math.min(dp[i-1][j], dp[i][j-1]));
                }
            }
        }
        return dp[m][n];
    }
}
```

---

75. Sort Colors

- **Summary:** Sort array of 0,1,2 (Dutch National Flag problem).
- **Key Idea:** Three pointers (low, mid, high).

```
class Solution {
    public void sortColors(int[] nums) {
        int low = 0, mid = 0, high = nums.length - 1;
        while (mid <= high) {
            if (nums[mid] == 0) {
                int temp = nums[low];
                nums[low] = nums[mid];
                nums[mid] = temp;
                low++; mid++;
            } else if (nums[mid] == 1) {
                mid++;
            } else {
                int temp = nums[mid];
                nums[mid] = nums[high];
                nums[high] = temp;
                high--;
```

```
            }
        }
    }
}
```

76. Maximum Subarray (LC #53)

- **Summary:** Find contiguous subarray with the largest sum.
- **Key Idea:** Kadane's algorithm.

```
class Solution {
    public int maxSubArray(int[] nums) {
        int maxSum = nums[0], currSum = nums[0];
        for (int i = 1; i < nums.length; i++) {
            currSum = Math.max(nums[i], currSum + nums[i]);
            maxSum = Math.max(maxSum, currSum);
        }
        return maxSum;
    }
}
```

---

77. Best Time to Buy and Sell Stock (LC #121)

- **Summary:** Max profit from buying and selling one stock.
- **Key Idea:** Track min price so far.

```
class Solution {
    public int maxProfit(int[] prices) {
        int minPrice = Integer.MAX_VALUE, maxProfit = 0;
        for (int price : prices) {
            minPrice = Math.min(minPrice, price);
            maxProfit = Math.max(maxProfit, price - minPrice);
        }
        return maxProfit;
    }
```

```
}
```

---

78. Product of Array Except Self (LC #238)

- **Summary:** Return array where each element is product of all other elements.
- **Key Idea:** Prefix & suffix products.

```
class Solution {
    public int[] productExceptSelf(int[] nums) {
        int n = nums.length;
        int[] result = new int[n];
        result[0] = 1;
        for (int i = 1; i < n; i++) result[i] = result[i-1] *
nums[i-1];
        int suffix = 1;
        for (int i = n-1; i >= 0; i--) {
            result[i] *= suffix;
            suffix *= nums[i];
        }
        return result;
    }
}
```

---

79. Maximum Product Subarray (LC #152)

- **Summary:** Max product of contiguous subarray.
- **Key Idea:** Track max/min due to negative numbers.

```
class Solution {
    public int maxProduct(int[] nums) {
        int maxProd = nums[0], minProd = nums[0], result = nums[0];
        for (int i = 1; i < nums.length; i++) {
            int num = nums[i];
```

```
            int tempMax = Math.max(num, Math.max(maxProd * num,
minProd * num));
            minProd = Math.min(num, Math.min(maxProd * num, minProd *
num));
            maxProd = tempMax;
            result = Math.max(result, maxProd);
        }
        return result;
    }
}
```

---

80. Find Minimum in Rotated Sorted Array (LC #153)

- **Summary:** Array sorted then rotated; find min element.
- **Key Idea:** Binary search.

```
class Solution {
    public int findMin(int[] nums) {
        int left = 0, right = nums.length - 1;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] > nums[right]) left = mid + 1;
            else right = mid;
        }
        return nums[left];
    }
}
```

---

81. Search in Rotated Sorted Array (LC #33)

- **Summary:** Search target in rotated sorted array.
- **Key Idea:** Binary search with rotation check.

```
class Solution {
    public int search(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left <= right) {
            int mid = left + (right - left)/2;
            if (nums[mid] == target) return mid;
            if (nums[left] <= nums[mid]) {
                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;
    }
}
```

---

82. Container With Most Water (LC #11)

- **Summary:** Max area formed by lines on x-axis.
- **Key Idea:** Two-pointer technique.

```
class Solution {
    public int maxArea(int[] height) {
        int left = 0, right = height.length-1, maxArea = 0;
        while (left < right) {
            maxArea = Math.max(maxArea, Math.min(height[left],
height[right]) * (right-left));
            if (height[left] < height[right]) left++;
            else right--;
        }
        return maxArea;
    }
```

```
}
```

---

83. Valid Parentheses (LC #20)

- **Summary:** Check if string of brackets is valid.
- **Key Idea:** Stack.

```
class Solution {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();
        for (char c : s.toCharArray()) {
            if (c=='(') stack.push(')');
            else if (c=='{') stack.push('}');
            else if (c=='[') stack.push(']');
            else if (stack.isEmpty() || stack.pop() != c) return
false;
        }
        return stack.isEmpty();
    }
}
```

---

84. Merge Two Sorted Lists (LC #21)

- **Summary:** Merge two sorted linked lists.
- **Key Idea:** Two pointers.

```
class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0), curr = dummy;
        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) { curr.next = l1; l1 = l1.next; }
            else { curr.next = l2; l2 = l2.next; }
            curr = curr.next;
```

```
        }
        curr.next = (l1 != null) ? l1 : l2;
        return dummy.next;
    }
}
```

---

85. Intersection of Two Linked Lists (LC #160)

- **Summary:** Find intersection node of two singly linked lists.
- **Key Idea:** Two pointers traverse both lists.

```
class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode
headB) {
        ListNode a = headA, b = headB;
        while (a != b) {
            a = (a == null) ? headB : a.next;
            b = (b == null) ? headA : b.next;
        }
        return a;
    }
}
```

---

86. Linked List Cycle (LC #141)

- **Summary:** Detect if a linked list has a cycle.
- **Key Idea:** Fast & slow pointers.

```
class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null || head.next == null) return false;
        ListNode slow = head, fast = head.next;
        while (slow != fast) {
```

```
            if (fast == null || fast.next == null) return false;
            slow = slow.next;
            fast = fast.next.next;
        }
        return true;
    }
}
```

---

87. Binary Tree Inorder Traversal (LC #94)

- **Summary:** Return inorder traversal of a binary tree.
- **Key Idea:** Recursive or iterative (stack) solution.

```
class Solution {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        TreeNode curr = root;
        while (curr != null || !stack.isEmpty()) {
            while (curr != null) {
                stack.push(curr);
                curr = curr.left;
            }
            curr = stack.pop();
            res.add(curr.val);
            curr = curr.right;
        }
        return res;
    }
}
```

---

88. Binary Tree Level Order Traversal (LC #102)

- **Summary:** Return level-order traversal of a binary tree.

- **Key Idea:** BFS using queue.

```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> res = new ArrayList<>();
        if (root == null) return res;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()) {
            int size = queue.size();
            List<Integer> level = new ArrayList<>();
            for (int i=0; i<size; i++) {
                TreeNode node = queue.poll();
                level.add(node.val);
                if (node.left != null) queue.add(node.left);
                if (node.right != null) queue.add(node.right);
            }
            res.add(level);
        }
        return res;
    }
}
```

---

89. Maximum Depth of Binary Tree (LC #104)

- **Summary:** Return max depth of binary tree.
- **Key Idea:** DFS recursion.

```
class Solution {
    public int maxDepth(TreeNode root) {
        if (root == null) return 0;
        return 1 + Math.max(maxDepth(root.left),
maxDepth(root.right));
    }
}
```

---

90. Validate Binary Search Tree (LC #98)

- **Summary:** Check if a binary tree is a valid BST.
- **Key Idea:** DFS with min/max boundaries.

```
class Solution {
    public boolean isValidBST(TreeNode root) {
        return helper(root, null, null);
    }
    private boolean helper(TreeNode node, Integer min, Integer max) {
        if (node == null) return true;
        if ((min != null && node.val <= min) || (max != null &&
node.val >= max)) return false;
        return helper(node.left, min, node.val) && helper(node.right,
node.val, max);
    }
}
```

---

91. Serialize and Deserialize Binary Tree (LC #297)

- **Summary:** Encode/decode binary tree as a string.
- **Key Idea:** BFS or DFS serialization.

```
class Codec {
    public String serialize(TreeNode root) {
        if (root == null) return "null";
        return root.val + "," + serialize(root.left) + "," +
serialize(root.right);
    }
    public TreeNode deserialize(String data) {
        Queue<String> q = new
LinkedList<>(Arrays.asList(data.split(",")));
        return helper(q);
```

```
    }
    private TreeNode helper(Queue<String> q) {
        String val = q.poll();
        if (val.equals("null")) return null;
        TreeNode node = new TreeNode(Integer.parseInt(val));
        node.left = helper(q);
        node.right = helper(q);
        return node;
    }
}
```

---

92. Maximum Depth of N-ary Tree (LC #559)

- **Summary:** Return max depth of an N-ary tree.
- **Key Idea:** DFS recursion.

```
class Solution {
    public int maxDepth(Node root) {
        if (root == null) return 0;
        int max = 0;
        for (Node child : root.children) {
            max = Math.max(max, maxDepth(child));
        }
        return max + 1;
    }
}
```

---

93. Serialize and Deserialize N-ary Tree (LC #428)

- **Summary:** Encode/decode N-ary tree.
- **Key Idea:** DFS with marker for children count.

```
class Codec {
```

```
    public String serialize(Node root) {
        if (root == null) return "";
        StringBuilder sb = new StringBuilder();
        serializeHelper(root, sb);
        return sb.toString();
    }
    private void serializeHelper(Node root, StringBuilder sb) {
        if (root == null) return;

sb.append(root.val).append(",").append(root.children.size()).append(",
");
        for (Node child : root.children) serializeHelper(child, sb);
    }
    public Node deserialize(String data) {
        if (data.isEmpty()) return null;
        Queue<String> q = new
LinkedList<>(Arrays.asList(data.split(",")));
        return deserializeHelper(q);
    }
    private Node deserializeHelper(Queue<String> q) {
        int val = Integer.parseInt(q.poll());
        int size = Integer.parseInt(q.poll());
        Node node = new Node(val, new ArrayList<>());
        for (int i = 0; i < size; i++)
node.children.add(deserializeHelper(q));
        return node;
    }
}
```