DSA Crash Pack (24 Problems) — Explanations + Go Solutions
This pack is tailored for product/fintech/remote interviews. Each item contains a compact problem statement, reasoning, and an idiomatic Go solution with comments. Common LeetCode structs used below for reference:

TreeNode: *type TreeNode struct { Val int; Left, Right *TreeNode }*
ListNode: *type ListNode struct { Val int; Next *ListNode }*
Graph Node: *type Node struct { Val int; Neighbors []*Node }*

1) Two Sum (#1) — Easy

Problem Statement: Given an array nums and an integer target, return indices i, j such that nums[i] + nums[j] == target. Exactly one solution exists.

Idea: Scan once using a hashmap 'value→index'. For each num, check if target-num already seen; if yes, return cached index and current index. O(n) time, O(n) space.

Go Solution (with comments):
```go
func twoSum(nums []int, target int) []int {
    // Map from value to its index in the array
    idx := make(map[int]int)
    for i, v := range nums {
        // If we have seen target - v before, we found the pair
        if j, ok := idx[target-v]; ok {
            return []int{j, i}
        }
        // Otherwise, remember this value's index
        idx[v] = i
    }
    return nil // Per problem, solution exists; this is a fallback
}
```

2) Valid Parentheses (#20) — Easy

Problem Statement: Given a string s with brackets (), {}, [], determine if it is valid (properly closed and nested).

Idea: Use a stack. Push opening brackets; on closing bracket, top must match the pair. Stack empty at end ⇒ valid.

Go Solution (with comments):
```go
func isValid(s string) bool {
    stack := []rune{}
    pairs := map[rune]rune{')': '(', ']': '[', '}': '{'}

    for _, ch := range s {
        switch ch {
        case '(', '[', '{':
            stack = append(stack, ch) // push
        default:
            if len(stack) == 0 || stack[len(stack)-1] != pairs[ch] {
                return false
            }
            stack = stack[:len(stack)-1] // pop
        }
    }
    return len(stack) == 0
}
```

3) Merge Two Sorted Lists (#21) — Easy

Problem Statement: Merge two sorted linked lists l1 and l2 as a sorted list and return its head.

Idea: Iterative merge with a dummy head pointer. Compare nodes and advance. O(m+n).

Go Solution (with comments):
```
// Definition for singly-linked list used by LeetCode:
// type ListNode struct { Val int; Next *ListNode }
func mergeTwoLists(l1 *ListNode, l2 *ListNode) *ListNode {
    dummy := &ListNode{}
    cur := dummy

    for l1 != nil && l2 != nil {
        if l1.Val < l2.Val {
            cur.Next = l1
            l1 = l1.Next
        } else {
            cur.Next = l2
            l2 = l2.Next
        }
        cur = cur.Next
    }
    // Attach remaining nodes
    if l1 != nil {
        cur.Next = l1
    } else {
        cur.Next = l2
    }
    return dummy.Next
}
```

4) Best Time to Buy and Sell Stock (#121) — Easy

Problem Statement: Given prices[i] for a stock on day i, find the max profit (buy once, sell once).

Idea: Track min price so far and max profit. For each price, profit=max(profit, price-minSoFar). O(n).

Go Solution (with comments):
```
func maxProfit(prices []int) int {
    minSoFar := 1<<31 - 1 // large
    maxProfit := 0
    for _, p := range prices {
        if p < minSoFar {
            minSoFar = p
        } else if p-minSoFar > maxProfit {
            maxProfit = p - minSoFar
        }
    }
    return maxProfit
}
```

5) Valid Palindrome (#125) — Easy

Problem Statement: Given a string s, return true if it is a palindrome considering only alphanumeric chars and ignoring cases.

Idea: Two pointers from ends; skip non-alphanumeric; compare lowercased characters.

Go Solution (with comments):
```
func isPalindrome(s string) bool {
    i, j := 0, len(s)-1
    for i < j {
        // Move i to next alnum
        for i < j && !isAlnum(rune(s[i])) { i++ }
        // Move j to prev alnum
        for i < j && !isAlnum(rune(s[j])) { j-- }
        if toLower(rune(s[i])) != toLower(rune(s[j])) {
            return false
        }
        i++; j--
    }
    return true
}

// helpers
func isAlnum(r rune) bool {
    return (r >= 'a' && r <= 'z') || (r >= 'A' && r <= 'Z') || (r >= '0' && r <= '9')
}
func toLower(r rune) rune {
    if r >= 'A' && r <= 'Z' { return r - 'A' + 'a' }
    return r
}
```

6) Linked List Cycle (#141) — Easy

Problem Statement: Given head of a linked list, return true if it contains a cycle.

Idea: Floyd’s tortoise and hare: move slow by 1, fast by 2; if they meet, cycle exists.

Go Solution (with comments):
```
func hasCycle(head *ListNode) bool {
    if head == nil || head.Next == nil { return false }
    slow, fast := head, head.Next
    for fast != nil && fast.Next != nil {
        if slow == fast { return true }
        slow = slow.Next
        fast = fast.Next.Next
    }
    return false
}
```

7) Maximum Depth of Binary Tree (#104) — Easy

Problem Statement: Return the depth (max levels) of a binary tree.

Idea: Recursive DFS: depth = 1 + max(depth(left), depth(right)).

Go Solution (with comments):
```
func maxDepth(root *TreeNode) int {
    if root == nil { return 0 }
    left := maxDepth(root.Left)
    right := maxDepth(root.Right)
    if left > right { return left + 1 }
    return right + 1
}
```

8) Binary Search (#704) — Easy

Problem Statement: Search target in a sorted array. Return index or -1.

Idea: Classic iterative binary search with left/right pointers. O(log n).

Go Solution (with comments):

```
func search(nums []int, target int) int {
    l, r := 0, len(nums)-1
    for l <= r {
        mid := l + (r-l)/2
        if nums[mid] == target {
            return mid
        } else if nums[mid] < target {
            l = mid + 1
        } else {
            r = mid - 1
        }
    }
    return -1
}
```

9) Flood Fill (#733) — Easy

Problem Statement: Given an image grid, starting pixel (sr, sc) and newColor, recolor the connected component of the start pixel.

Idea: DFS/BFS from (sr, sc) replacing original color; stop when out of bounds or color differs.

Go Solution (with comments):

```
func floodFill(image [][]int, sr int, sc int, newColor int) [][]int {
    orig := image[sr][sc]
    if orig == newColor { return image }
    var dfs func(r, c int)
    dfs = func(r, c int) {
        if r < 0 || c < 0 || r >= len(image) || c >= len(image[0]) { return }
        if image[r][c] != orig { return }
        image[r][c] = newColor
        dfs(r+1, c); dfs(r-1, c); dfs(r, c+1); dfs(r, c-1)
    }
    dfs(sr, sc)
    return image
}
```

10) Lowest Common Ancestor of a BST (#235) — Easy

Problem Statement: Given root of a BST and nodes p and q, return their lowest common ancestor.

Idea: Use BST property: if both < root -> left; both > root -> right; else root is LCA.

Go Solution (with comments):

```
func lowestCommonAncestor(root, p, q *TreeNode) *TreeNode {
    cur := root
    for cur != nil {
        if p.Val < cur.Val && q.Val < cur.Val {
            cur = cur.Left
        } else if p.Val > cur.Val && q.Val > cur.Val {
            cur = cur.Right
        } else {
            return cur // split point
        }
    }
    return nil
}
```

11) Balanced Binary Tree (#110) — Easy

Problem Statement: Return true if the tree is height-balanced (left/right subtree heights differ by no more than 1).

Idea: Post-order recursion returning height; use -1 as a sentinel meaning 'unbalanced'.

Go Solution (with comments):

```
func isBalanced(root *TreeNode) bool {
    return height(root) != -1
}

func height(n *TreeNode) int {
    if n == nil { return 0 }
    lh := height(n.Left)
    if lh == -1 { return -1 } // left unbalanced
    rh := height(n.Right)
    if rh == -1 { return -1 } // right unbalanced
    if abs(lh-rh) > 1 { return -1 }
    if lh > rh { return lh + 1 }
    return rh + 1
```

```go
}

func abs(x int) int { if x < 0 { return -1*x }; return x }
```

12) Implement Queue using Stacks (#232) — Easy

Problem Statement: Implement a queue with standard operations using two stacks.

Idea: Use two stacks: ‘in’ for push, ‘out’ for pop/peek. When ‘out’ empty, pour from ‘in’. Amortized O(1).

Go Solution (with comments):

```go
type MyQueue struct {
    in, out []int
}

func Constructor() MyQueue { return MyQueue{} }

func (q *MyQueue) Push(x int) {
    q.in = append(q.in, x) // push onto input stack
}

func (q *MyQueue) pour() {
    // Move elements to 'out' to reverse order, only when needed
    for len(q.in) > 0 {
        n := len(q.in)
        q.out = append(q.out, q.in[n-1])
        q.in = q.in[:n-1]
    }
}

func (q *MyQueue) Pop() int {
    if len(q.out) == 0 { q.pour() }
    n := len(q.out)
    x := q.out[n-1]
    q.out = q.out[:n-1]
    return x
}

func (q *MyQueue) Peek() int {
    if len(q.out) == 0 { q.pour() }
    return q.out[len(q.out)-1]
}

func (q *MyQueue) Empty() bool {
    return len(q.in) == 0 && len(q.out) == 0
}
```

13) Min Stack (#155) — Easy

Problem Statement: Design a stack that supports push, pop, top, and retrieving the minimum in O(1).

Idea: Keep an auxiliary stack of current minimums or store pairs (val, minSoFar).

Go Solution (with comments):

```
type MinStack struct {
    st   []int
    mins []int
}

func ConstructorMinStack() MinStack { return MinStack{} }

func (m *MinStack) Push(val int) {
    m.st = append(m.st, val)
    if len(m.mins) == 0 || val <= m.mins[len(m.mins)-1] {
        m.mins = append(m.mins, val)
    }
}

func (m *MinStack) Pop() {
    if len(m.st) == 0 { return }
    v := m.st[len(m.st)-1]
    m.st = m.st[:len(m.st)-1]
    if v == m.mins[len(m.mins)-1] {
        m.mins = m.mins[:len(m.mins)-1]
    }
}

func (m *MinStack) Top() int { return m.st[len(m.st)-1] }

func (m *MinStack) GetMin() int { return m.mins[len(m.mins)-1] }
```

14) Invert Binary Tree (#226) — Easy

Problem Statement: Invert a binary tree (mirror it).

Idea: Swap left and right recursively (or iteratively).

Go Solution (with comments):

```
func invertTree(root *TreeNode) *TreeNode {
    if root == nil { return nil }
    root.Left, root.Right = invertTree(root.Right), invertTree(root.Left)
    return root
}
```

15) Product of Array Except Self (#238) — Medium

Problem Statement: Return an array output where output[i] is the product of all elements of nums except nums[i], without using division.

Idea: Two passes: prefix products left→right then multiply by suffix products right→left. O(n) time, O(1) extra space (excluding output).

Go Solution (with comments):

```
func productExceptSelf(nums []int) []int {
    n := len(nums)
    ans := make([]int, n)
    // Build prefix products
    pre := 1
    for i := 0; i < n; i++ {
        ans[i] = pre
        pre *= nums[i]
    }
    // Multiply by suffix products
    suf := 1
    for i := n-1; i >= 0; i-- {
        ans[i] *= suf
        suf *= nums[i]
    }
    return ans
}
```

16) Maximum Subarray (Kadane’s) (#53) — Medium

Problem Statement: Find the contiguous subarray with the largest sum.

Idea: Kadane’s algorithm: running best ending at i and global best. O(n).

Go Solution (with comments):

```
func maxSubArray(nums []int) int {
    bestEnd, best := nums[0], nums[0]
    for i := 1; i < len(nums); i++ {
        // Either extend previous subarray or start new at nums[i]
        if bestEnd+nums[i] > nums[i] {
            bestEnd = bestEnd + nums[i]
        } else {
            bestEnd = nums[i]
        }
        if bestEnd > best { best = bestEnd }
    }
    return best
}
```

17) 3Sum (#15) — Medium

Problem Statement: Find all unique triplets in the array which give the sum of zero.

Idea: Sort, then for each i, two-pointer sweep on remaining range, skipping duplicates.

Go Solution (with comments):

```
import "sort"

func threeSum(nums []int) [][]int {
    sort.Ints(nums)
    res := [][]int{}
    n := len(nums)
    for i := 0; i < n; i++ {
        if i > 0 && nums[i] == nums[i-1] { continue } // skip duplicate i
        l, r := i+1, n-1
        for l < r {
            sum := nums[i] + nums[l] + nums[r]
            if sum == 0 {
                res = append(res, []int{nums[i], nums[l], nums[r]})
                l++
                for l < r && nums[l] == nums[l-1] { l++ } // skip dups
                r--
                for l < r && nums[r] == nums[r+1] { r-- }
            } else if sum < 0 {
                l++
            } else {
                r--
            }
        }
    }
    return res
}
```

18) Container With Most Water (#11) — Medium

Problem Statement: Given an array of heights, find max water container area using two vertical lines.

Idea: Two pointers at ends; move the smaller height inward to possibly find a taller line. O(n).

Go Solution (with comments):

```
func maxArea(height []int) int {
    l, r := 0, len(height)-1
    best := 0
    for l < r {
        // Area is limited by the shorter line
        h := height[l]
        if height[r] < h { h = height[r] }
        area := (r - l) * h
        if area > best { best = area }
        // Move the pointer at the shorter line inward
        if height[l] < height[r] {
            l++
        } else {
            r--
        }
    }
    return best
}
```

19) Longest Substring Without Repeating Characters (#3) — Medium

Problem Statement: Given a string s, find the length of the longest substring without repeating characters.

Idea: Sliding window with last-seen index map. Expand right; move left past last occurrence when duplicate seen.

Go Solution (with comments):
```
func lengthOfLongestSubstring(s string) int {
    last := make(map[rune]int) // last seen index + 1
    best, left := 0, 0
    for i, ch := range []rune(s) {
        if pos, ok := last[ch]; ok && pos > left {
            left = pos // jump left just past previous occurrence
        }
        // window is s[left..i]
        cur := i - left + 1
        if cur > best { best = cur }
        last[ch] = i + 1 // store index+1 to avoid off-by-one
    }
    return best
}
```

20) Top K Frequent Elements (#347) — Medium

Problem Statement: Return the k most frequent elements in nums.

Idea: Count frequencies, then bucket-sort by frequency and collect from high to low. O(n).

Go Solution (with comments):
```
func topKFrequent(nums []int, k int) []int {
    // 1) Count frequency
    freq := make(map[int]int)
    for _, v := range nums { freq[v]++ }

    // 2) Buckets: index = frequency, each holds list of values with that frequency
    buckets := make([][]int, len(nums)+1)
    for v, f := range freq {
        buckets[f] = append(buckets[f], v)
    }

    // 3) Collect from highest frequency down
    res := []int{}
    for f := len(buckets)-1; f >= 0 && len(res) < k; f-- {
        for _, v := range buckets[f] {
            res = append(res, v)
            if len(res) == k { break }
        }
    }
    return res
}
```

21) Number of Islands (#200) — Medium

Problem Statement: Given a 2D grid of '1's (land) and '0's (water), count the number of islands (connected 1s).

Idea: DFS/BFS and mark visited by turning '1' to '0' (or a visited set).

Go Solution (with comments):

```
func numIslands(grid [][]byte) int {
    if len(grid) == 0 { return 0 }
    rows, cols := len(grid), len(grid[0])
    var dfs func(r, c int)
    dfs = func(r, c int) {
        if r < 0 || c < 0 || r >= rows || c >= cols || grid[r][c] != '1' {
            return
        }
        grid[r][c] = '0' // mark visited
        dfs(r+1, c); dfs(r-1, c); dfs(r, c+1); dfs(r, c-1)
    }
    count := 0
    for r := 0; r < rows; r++ {
        for c := 0; c < cols; c++ {
            if grid[r][c] == '1' {
                count++
                dfs(r, c)
            }
        }
    }
    return count
}
```

22) Clone Graph (#133) — Medium

Problem Statement: Clone an undirected graph. Each node has a value and list of neighbors.

Idea: DFS/BFS with a map old→new. Create node on first visit, then recursively/iteratively attach cloned neighbors.

Go Solution (with comments):

```
// Node definition used by LeetCode:
// type Node struct { Val int; Neighbors []*Node }
func cloneGraph(node *Node) *Node {
    if node == nil { return nil }
    seen := map[*Node]*Node{} // original -> clone

    var dfs func(*Node) *Node
    dfs = func(n *Node) *Node {
        if n == nil { return nil }
        if cp, ok := seen[n]; ok { return cp }
        // clone the node (empty neighbors for now)
        copy := &Node{Val: n.Val}
        seen[n] = copy
        // clone neighbors
        for _, nei := range n.Neighbors {
            copy.Neighbors = append(copy.Neighbors, dfs(nei))
        }
        return copy
    }
    return dfs(node)
}
```

23) House Robber (#198) — Medium

Problem Statement: Given non-negative integers representing amount of money of each house, determine max amount without robbing adjacent houses.

Idea: DP with two states: include/exclude. Iteratively keep prev and prev2 (space-optimized).

Go Solution (with comments):

```go
func rob(nums []int) int {
    if len(nums) == 0 { return 0 }
    if len(nums) == 1 { return nums[0] }
    prev2, prev1 := nums[0], max(nums[0], nums[1])
    for i := 2; i < len(nums); i++ {
        cur := max(prev1, prev2+nums[i]) // rob this or skip
        prev2, prev1 = prev1, cur
    }
    return prev1
}
func max(a, b int) int { if a > b { return a }; return b }
```

24) Course Schedule (#207) — Medium

Problem Statement: Given number of courses and prerequisite pairs, determine if you can finish all courses.

Idea: Kahn’s algorithm for topological sort: compute in-degrees, process nodes with 0 in-degree; if processed count equals numCourses, it’s possible.

Go Solution (with comments):

```go
func canFinish(numCourses int, prerequisites [][]int) bool {
    // Build graph and indegree
    g := make([][]int, numCourses)
    indeg := make([]int, numCourses)
    for _, p := range prerequisites {
        to, from := p[0], p[1]
        g[from] = append(g[from], to)
        indeg[to]++
    }

    // Queue for nodes with indegree 0
    q := []int{}
    for i := 0; i < numCourses; i++ {
        if indeg[i] == 0 { q = append(q, i) }
    }

    taken := 0
    for len(q) > 0 {
        // pop front
        x := q[0]; q = q[1:]
        taken++
        // reduce indegree of neighbors
        for _, nb := range g[x] {
            indeg[nb]--
            if indeg[nb] == 0 {
                q = append(q, nb)
            }
        }
    }
    return taken == numCourses
}
```