Blind 75 Reference Playbook

Complete Quick-Glance Guide for the Classic Set

Updated November 5, 2025

Curated notes to speed up refresh sessions before interviews

Contents

1 Array & Hashing Overview

ID	Problem	Patterns & Tactical Notes
1	Two Sum	Hash map complements; watch for duplicate indices.
121	Best Time to Buy and Sell Stock	Track running minimum price; one pass $O(n)$.
217	Contains Duplicate	Set membership or sort; $O(n)$ with hash set.
238	Product of Array Except Self	Prefix/suffix products without division; constant extra space.
53	Maximum Subarray	Kadane's algorithm; track best ending here vs. global best.
152	Maximum Product Subarray	Maintain max/min prefix products to handle negatives.
153	Find Minimum in Rotated Sorted Array	Binary search pivot; compare mid with right.
33	Search in Rotated Sorted Array	Identify sorted half each iteration; adjust binary search bounds.
15	3Sum	Sort + two pointers; skip duplicates to stay $O(n^2)$.
11	Container With Most Water	Two pointers moving inward; shrink shorter wall.

2  Binary & Bit Manipulation Overview

ID	Problem	Patterns & Tactical Notes
371	Sum of Two Integers	Bitwise XOR for sum, AND for carry; iterate until carry vanishes.
191	Number of 1 Bits	Use n&=n-1 trick to clear lowest set bit.
338	Counting Bits	Build DP by offset or popcount relation: $dp[i] = dp[i/2] + i\%2$.
268	Missing Number	XOR index with value or use Gauss summation.
190	Reverse Bits	Bitwise shifts; mask low bit, build reversed result.

3 Dynamic Programming Overview

ID	Problem	Patterns & Tactical Notes
70	Climbing Stairs	Classic Fibonacci DP; iterative two-variable solution.
322	Coin Change	Bottom-up DP; initialize to inf, iterate coins outer.
300	Longest Increasing Subsequence	Patience sorting with binary search for $O(n \log n)$.
1143	Longest Common Subsequence	2D DP; reuse rolling rows for $O(nm)$ time, $O(m)$ space.
139	Word Break	DP on prefix validity using hash set dictionary.
377	Combination Sum IV	Order matters; one-dimensional DP across target.
198	House Robber	Either rob or skip; tracking two states is enough.
213	House Robber II	Handle circularity via two linear robber runs (exclude first/last).
91	Decode Ways	DP on index; beware invalid zeros and bounds.
62	Unique Paths	Combinatorics or DP grid; row-by-row accumulation.
55	Jump Game	Greedy reachability; maintain furthest index reachable so far.

4 Graphs & Traversals Overview

ID	Problem	Patterns & Tactical Notes
133	Clone Graph	DFS/BFS with hash map from original node to clone.
207	Course Schedule	Topological sort via Kahn's algorithm or DFS cycle detection.
417	Pacific Atlantic Water Flow	Reverse BFS/DFS from oceans; intersect reachable sets.
200	Number of Islands	Flood-fill grid; mark visited in-place for $O(1)$ extra space.
261	Graph Valid Tree	Check edges == n-1 and no cycle via UF or DFS.
323	Number of Connected Components in an Undirected Graph	Union-find or DFS across adjacency list.
127	Word Ladder	Bidirectional BFS on wildcard adjacency to reduce branching.

5 Interval Scheduling Overview

ID	Problem	Patterns & Tactical Notes
57	Insert Interval	Linear sweep merging overlaps; append tail segments.
56	Merge Intervals	Sort by start then merge greedily.
435	Non-overlapping Intervals	Greedy by earliest finish; count removals.
252	Meeting Rooms	Sort starts vs. ends; ensure rooms needed does not exceed 1.

6 🔗 Linked List Overview

ID	Problem	Patterns & Tactical Notes
206	Reverse Linked List	Iterative pointer flip; maintain prev, curr, next.
21	Merge Two Sorted Lists	Dummy head + two pointers; operate in-place.
143	Reorder List	Split middle, reverse second half, interleave nodes.
19	Remove Nth Node From End of List	Two-pointer window or sentinel + length count.
141	Linked List Cycle	Floyd’s tortoise and hare cycle detection.

7 🪄 Matrix Manipulation Overview

ID	Problem	Patterns & Tactical Notes
73	Set Matrix Zeroes	Use first row/col as markers to keep $O(1)$ space.
54	Spiral Matrix	Iteratively peel layers tracking four boundaries.
48	Rotate Image	Transpose + reverse rows for in-place rotation.
79	Word Search	Backtracking DFS with visited state; prune early on mismatch.

8 String Patterns Overview

ID	Problem	Patterns & Tactical Notes
3	Longest Substring Without Repeating Characters	Sliding window with index map; shrink when duplicate seen.
424	Longest Repeating Character Replacement	Window maintain max frequency; shrink when window - max > k.
76	Minimum Window Substring	Two-pointer window counting required chars; expand/contract with frequency map.
242	Valid Anagram	Char count array for lowercase; compare counts.
49	Group Anagrams	Sort signature or letter counts as hash key.
20	Valid Parentheses	Stack track opening brackets; ensure final stack empty.
125	Valid Palindrome	Two pointers ignore non-alphanumeric; case-normalize.
5	Longest Palindromic Substring	Expand around centers; track best window.
647	Palindromic Substrings	Center expansion counts palindromes in $O(n^2)$.
271	Encode and Decode Strings	Length-prefix encoding to handle delimiter collisions.

9 Binary Tree Toolkit Overview

ID	Problem	Patterns & Tactical Notes
104	Maximum Depth of Binary Tree	DFS depth calculation; return 1 + max(left, right).
100	Same Tree	Recursive structural + value equality check.
226	Invert Binary Tree	Swap children recursively or via BFS.
102	Binary Tree Level Order Traversal	Queue BFS capturing per-level nodes.
572	Subtree of Another Tree	DFS roots comparing structures; serialize for optimization.
105	Construct Binary Tree from Preorder and Inorder Traversal	Use preorder root + inorder index map for recursion.
98	Validate Binary Search Tree	Maintain min/max bounds during DFS.
230	Kth Smallest Element in a BST	Inorder traversal yields sorted order; stop at k.
235	Lowest Common Ancestor of a Binary Search Tree	Traverse using BST property; diverging node is LCA.
124	Binary Tree Maximum Path Sum	Postorder tracking max gain from children; update global best.
297	Serialize and Deserialize Binary Tree	Preorder with null markers; use iterator for decode.

10 💎 Heaps & Priority Queues Overview

ID	Problem	Patterns & Tactical Notes
23	Merge k Sorted Lists	Min-heap of list heads; push next node after pop.
347	Top K Frequent Elements	Frequency map + heap or bucket sort for $O(n)$.
295	Find Median from Data Stream	Two heaps balancing lower, upper halves.

11 🎭 Backtracking Patterns Overview

ID	Problem	Patterns & Tactical Notes
78	Subsets	DFS include/exclude pattern; use iterative expansion as alternative.
39	Combination Sum	DFS with repeat allowed; maintain start index, prune when sum exceeds target.
46	Permutations	Swap in-place or track used set; depth equals length.
131	Palindrome Partitioning	Backtrack with palindrome check; precompute pal table if needed.
17	Letter Combinations of a Phone Number	Map digits to letters; build combinations recursively.

12 Deep Dive Playbook

Each Blind 75 problem below includes a focused reminder, complexity summary, and a clean Python 3 template you can adapt quickly. Mirror the structure you favor during interviews and annotate directly in the PDF as you review.

13 Array & Hashing Deep Dive

13.1 1. Two Sum

Key Idea: Store complements in a hash map while scanning once. **Complexity:** Time $O(n)$, Space $O(n)$.

```python
def twoSum(nums, target):
    index = {}
    for i, num in enumerate(nums):
        diff = target - num
        if diff in index:
            return [index[diff], i]
        index[num] = i
    return []
```

13.2 121. Best Time to Buy and Sell Stock

Key Idea: Track the running minimum price and best profit in a single pass. **Complexity:** Time $O(n)$, Space $O(1)$.

```python
def maxProfit(prices):
    best = 0
    min_price = prices[0]
    for price in prices[1:]:
        best = max(best, price - min_price)
        min_price = min(min_price, price)
    return best
```

13.3 217. Contains Duplicate

Key Idea: Use a hash set to short-circuit on first repeat. **Complexity:** Time $O(n)$, Space $O(n)$.

```python
def containsDuplicate(nums):
    seen = set()
    for num in nums:
        if num in seen:
            return True
        seen.add(num)
    return False
```

13.4 238. Product of Array Except Self

Key Idea: Two passes for prefix and suffix products without division. **Complexity:** Time $O(n)$, Space $O(1)$ extra.

```
def productExceptSelf(nums):
    res = [1] * len(nums)
    prefix = 1
    for i, num in enumerate(nums):
        res[i] = prefix
        prefix *= num
    suffix = 1
    for i in range(len(nums) - 1, -1, -1):
        res[i] *= suffix
        suffix *= nums[i]
    return res
```

13.5 53. Maximum Subarray

Key Idea: Kadane's algorithm keeps best subarray ending at current index. **Complexity:** Time $O(n)$, Space $O(1)$.

```
def maxSubArray(nums):
    best = cur = nums[0]
    for num in nums[1:]:
        cur = max(num, cur + num)
        best = max(best, cur)
    return best
```

13.6 152. Maximum Product Subarray

Key Idea: Track both max and min products to handle negative flips. **Complexity:** Time $O(n)$, Space $O(1)$.

```
def maxProduct(nums):
    best = cur_max = cur_min = nums[0]
    for num in nums[1:]:
        if num < 0:
            cur_max, cur_min = cur_min, cur_max
        cur_max = max(num, cur_max * num)
        cur_min = min(num, cur_min * num)
        best = max(best, cur_max)
    return best
```

13.7 153. Find Minimum in Rotated Sorted Array

Key Idea: Binary search compares mid and right to locate pivot. **Complexity:** Time $O(\log n)$, Space $O(1)$.

```python
def findMin(nums):
    left, right = 0, len(nums) - 1
    while left < right:
        mid = (left + right) // 2
        if nums[mid] > nums[right]:
            left = mid + 1
        else:
            right = mid
    return nums[left]
```

13.8 33. Search in Rotated Sorted Array

Key Idea: Decide which half is sorted each loop and tighten bounds. **Complexity:** Time $O(\log n)$, Space $O(1)$.

```python
def search(nums, target):
    left, right = 0, len(nums) - 1
    while left <= right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid
        if nums[left] <= nums[mid]:
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        else:
            if nums[mid] < target <= nums[right]:
                left = mid + 1
            else:
                right = mid - 1
    return -1
```

13.9 15. 3Sum

Key Idea: Sort then sweep with two pointers, skipping duplicates. **Complexity:** Time $O(n^2)$, Space $O(1)$ plus output.

```
def threeSum(nums):
    nums.sort()
    triplets = []
    for i in range(len(nums) - 2):
        if i and nums[i] == nums[i - 1]:
            continue
        left, right = i + 1, len(nums) - 1
        while left < right:
            s = nums[i] + nums[left] + nums[right]
            if s == 0:
                triplets.append([nums[i], nums[left], nums[right]])
                left += 1
                right -= 1
                while left < right and nums[left] == nums[left - 1]:
                    left += 1
                while left < right and nums[right] == nums[right + 1]:
                    right -= 1
            elif s < 0:
                left += 1
            else:
                right -= 1
    return triplets
```

13.10 11. Container With Most Water

Key Idea: Two pointers move inward, always dropping shorter wall. **Complexity:** Time $O(n)$, Space $O(1)$.

```
def maxArea(height):
    left, right = 0, len(height) - 1
    best = 0
    while left < right:
        width = right - left
        best = max(best, width * min(height[left], height[right]))
        if height[left] < height[right]:
            left += 1
        else:
            right -= 1
    return best
```

14 Binary & Bit Manipulation Deep Dive

14.1 371. Sum of Two Integers

Key Idea: Use XOR for partial sum and AND for carry until no carry remains. **Complexity:** Time $O(1)$ with fixed-width ints, Space $O(1)$.

```python
def getSum(a, b):
    mask = 0xFFFFFFFF
    max_int = 0x7FFFFFFF
    while b:
        a, b = (a ^ b) & mask, ((a & b) << 1) & mask
    return a if a <= max_int else ~(a ^ mask)
```

14.2 191. Number of 1 Bits

Key Idea: Repeatedly clear the lowest set bit with $n \;\&= n-1$. **Complexity:** Time $O(k)$ where k is number of set bits, Space $O(1)$.

```python
def hammingWeight(n):
    ones = 0
    while n:
        n &= n - 1
        ones += 1
    return ones
```

14.3 338. Counting Bits

Key Idea: Build DP using highest power of two seen so far. **Complexity:** Time $O(n)$, Space $O(n)$.

```python
def countBits(n):
    bits = [0] * (n + 1)
    offset = 1
    for i in range(1, n + 1):
        if offset * 2 == i:
            offset = i
        bits[i] = 1 + bits[i - offset]
    return bits
```

14.4 268. Missing Number

Key Idea: XOR indices and values to cancel every present number. **Complexity:** Time $O(n)$, Space $O(1)$.

```python
def missingNumber(nums):
    missing = len(nums)
    for i, num in enumerate(nums):
        missing ^= i ^ num
    return missing
```

14.5 190. Reverse Bits

Key Idea: Shift result left each step and pull lowest bit from input. **Complexity:** Time $O(1)$ for 32 bits, Space $O(1)$.

```python
def reverseBits(n):
    result = 0
    for _ in range(32):
        result = (result << 1) | (n & 1)
        n >>= 1
    return result
```

15 ☘ Dynamic Programming Deep Dive

15.1 70. Climbing Stairs

Key Idea: Fibonacci-style recurrence using two rolling variables. **Complexity:** Time $O(n)$, Space $O(1)$.

```python
def climbStairs(n):
    if n <= 2:
        return n
    one, two = 2, 1
    for _ in range(3, n + 1):
        one, two = one + two, one
    return one
```

15.2 322. Coin Change

Key Idea: Bottom-up DP for minimum coins, initialized to sentinel large value. **Complexity:** Time $O(n \cdot \text{amount})$, Space $O(\text{amount})$.

```python
def coinChange(coins, amount):
    dp = [amount + 1] * (amount + 1)
    dp[0] = 0
    for coin in coins:
        for total in range(coin, amount + 1):
            dp[total] = min(dp[total], 1 + dp[total - coin])
    return dp[amount] if dp[amount] <= amount else -1
```

15.3 300. Longest Increasing Subsequence

Key Idea: Patience sorting with binary search maintains best tail for each length. **Complexity:** Time $O(n \log n)$, Space $O(n)$.

```python
from bisect import bisect_left


def lengthOfLIS(nums):
    tails = []
    for num in nums:
        idx = bisect_left(tails, num)
        if idx == len(tails):
            tails.append(num)
        else:
            tails[idx] = num
    return len(tails)
```

15.4 1143. Longest Common Subsequence

Key Idea: Rolling two-row DP saves memory while comparing characters. **Complexity:** Time $O(nm)$, Space $O(m)$.

```python
def longestCommonSubsequence(text1, text2):
    if len(text1) < len(text2):
        text1, text2 = text2, text1
    prev = [0] * (len(text2) + 1)
    for ch1 in text1:
        curr = [0]
        for j, ch2 in enumerate(text2, start=1):
            if ch1 == ch2:
                curr.append(1 + prev[j - 1])
            else:
                curr.append(max(prev[j], curr[-1]))
        prev = curr
    return prev[-1]
```

15.5 139. Word Break

Key Idea: Boolean DP on prefixes using dictionary set for quick lookup. **Complexity:** Time $O(n^2)$, Space $O(n)$.

```python
def wordBreak(s, wordDict):
    words = set(wordDict)
    dp = [False] * (len(s) + 1)
    dp[0] = True
    for i in range(1, len(s) + 1):
        for j in range(i):
            if dp[j] and s[j:i] in words:
                dp[i] = True
                break
    return dp[-1]
```

15.6 377. Combination Sum IV

Key Idea: Order matters; iterate through target and sum counts. **Complexity:** Time $O(n \cdot \text{target})$, Space $O(\text{target})$.

```python
def combinationSum4(nums, target):
    dp = [0] * (target + 1)
    dp[0] = 1
    for total in range(1, target + 1):
        for num in nums:
            if total >= num:
                dp[total] += dp[total - num]
    return dp[target]
```

15.7 198. House Robber

Key Idea: Track best loot with and without current house. **Complexity:** Time $O(n)$, Space $O(1)$.

```python
def rob(nums):
    rob_prev, skip_prev = 0, 0
    for value in nums:
        rob_prev, skip_prev = skip_prev + value, max(rob_prev, skip_prev)
    return max(rob_prev, skip_prev)
```

15.8 213. House Robber II

Key Idea: Solve two linear runs excluding first or last house and take max. **Complexity:** Time $O(n)$, Space $O(1)$.

```python
def rob(nums):
    if len(nums) == 1:
        return nums[0]

    def rob_line(houses):
        rob_prev, skip_prev = 0, 0
        for value in houses:
            rob_prev, skip_prev = skip_prev + value, max(rob_prev, skip_prev)
        return max(rob_prev, skip_prev)

    return max(rob_line(nums[1:]), rob_line(nums[:-1]))
```

15.9 91. Decode Ways

Key Idea: DP on index with handling for zeros and two-digit combinations. **Complexity:** Time $O(n)$, Space $O(1)$.

```python
def numDecodings(s):
    if not s or s[0] == "0":
        return 0
    one, two = 1, 1
    for i in range(1, len(s)):
        cur = 0
        if s[i] != "0":
            cur += one
        two_digit = int(s[i - 1 : i + 1])
        if 10 <= two_digit <= 26:
            cur += two
        if cur == 0:
            return 0
        two, one = one, cur
    return one
```

15.10 62. Unique Paths

Key Idea: DP accumulation per row (or compute combinatorially). **Complexity:** Time $O(mn)$, Space $O(n)$.

```python
def uniquePaths(m, n):
    row = [1] * n
    for _ in range(1, m):
        for j in range(1, n):
            row[j] += row[j - 1]
    return row[-1]
```

15.11 55. Jump Game

Key Idea: Greedy keep furthest reachable index and ensure current index is reachable. **Complexity:** Time $O(n)$, Space $O(1)$.

```python
def canJump(nums):
    reach = 0
    for i, jump in enumerate(nums):
        if i > reach:
            return False
        reach = max(reach, i + jump)
    return True
```

16 Graphs & Traversals Deep Dive

16.1 133. Clone Graph

Key Idea: DFS with hash map storing originals to clones. **Complexity:** Time $O(V + E)$, Space $O(V)$.

```python
class Solution:
    def cloneGraph(self, node):
        if not node:
            return None
        clones = {}

        def dfs(cur):
            if cur in clones:
                return clones[cur]
            copy = Node(cur.val)
            clones[cur] = copy
            for neighbor in cur.neighbors:
                copy.neighbors.append(dfs(neighbor))
            return copy

        return dfs(node)
```

16.2 207. Course Schedule

Key Idea: Kahn's algorithm counts nodes processed; detect cycle if total mismatch. **Complexity:** Time $O(V + E)$, Space $O(V + E)$.

```python
from collections import defaultdict, deque


def canFinish(numCourses, prerequisites):
    graph = defaultdict(list)
    indegree = [0] * numCourses
    for nxt, pre in prerequisites:
        graph[pre].append(nxt)
        indegree[nxt] += 1
    queue = deque(i for i, deg in enumerate(indegree) if deg == 0)
    visited = 0
    while queue:
        course = queue.popleft()
        visited += 1
        for nxt in graph[course]:
            indegree[nxt] -= 1
            if indegree[nxt] == 0:
                queue.append(nxt)
    return visited == numCourses
```

16.3 417. Pacific Atlantic Water Flow

Key Idea: Reverse flood fill from both oceans and intersect reachable cells. **Complexity:** Time $O(mn)$, Space $O(mn)$.

```
def pacificAtlantic(heights):
    rows, cols = len(heights), len(heights[0])

    def bfs(starts):
        reach = set(starts)
        queue = list(starts)
        for r, c in queue:
            for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
                nr, nc = r + dr, c + dc
                if 0 <= nr < rows and 0 <= nc < cols:
                    if (nr, nc) not in reach and heights[nr][nc] >=
                    ↪  heights[r][c]:
                        reach.add((nr, nc))
                        queue.append((nr, nc))
        return reach

    pacific = bfs([(0, c) for c in range(cols)] + [(r, 0) for r in
    ↪  range(rows)])
    atlantic = bfs([(rows - 1, c) for c in range(cols)] + [(r, cols - 1) for r
    ↪  in range(rows)])
    return [[r, c] for (r, c) in pacific & atlantic]
```

16.4 200. Number of Islands

Key Idea: DFS flood fill to mark land as visited in-place. **Complexity:** Time $O(mn)$, Space $O(mn)$ worst case recursion.

```
def numIslands(grid):
    rows, cols = len(grid), len(grid[0])

    def dfs(r, c):
        if r < 0 or r >= rows or c < 0 or c >= cols or grid[r][c] != "1":
            return
        grid[r][c] = "0"
        dfs(r + 1, c)
        dfs(r - 1, c)
        dfs(r, c + 1)
        dfs(r, c - 1)

    count = 0
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == "1":
```

```
                count += 1
                dfs(r, c)
    return count
```

16.5 261. Graph Valid Tree

Key Idea: Union-find ensures no cycles and connectedness. **Complexity:** Time $O(E\alpha(V))$, Space $O(V)$.

```
def validTree(n, edges):
    if len(edges) != n - 1:
        return False

    parent = list(range(n))
    rank = [0] * n

    def find(x):
        while parent[x] != x:
            parent[x] = parent[parent[x]]
            x = parent[x]
        return x

    def union(x, y):
        root_x, root_y = find(x), find(y)
        if root_x == root_y:
            return False
        if rank[root_x] < rank[root_y]:
            parent[root_x] = root_y
        elif rank[root_x] > rank[root_y]:
            parent[root_y] = root_x
        else:
            parent[root_y] = root_x
            rank[root_x] += 1
        return True

    return all(union(u, v) for u, v in edges)
```

16.6 323. Number of Connected Components

Key Idea: DFS across adjacency lists counts components. **Complexity:** Time $O(V + E)$, Space $O(V)$.

```python
from collections import defaultdict


def countComponents(n, edges):
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    seen = set()

    def dfs(node):
        stack = [node]
        while stack:
            cur = stack.pop()
            if cur in seen:
                continue
            seen.add(cur)
            stack.extend(graph[cur])

    components = 0
    for node in range(n):
        if node not in seen:
            components += 1
            dfs(node)
    return components
```

16.7 127. Word Ladder

Key Idea: Bidirectional BFS reduces branching using generic wildcard buckets. **Complexity:** Time $O(M^2N)$ where M is word length, Space $O(M^2N)$.

```python
from collections import defaultdict, deque


def ladderLength(beginWord, endWord, wordList):
    if endWord not in wordList:
        return 0
    neighbors = defaultdict(list)
    for word in wordList:
        for i in range(len(word)):
            generic = word[:i] + "*" + word[i + 1 :]
            neighbors[generic].append(word)

    begin_visit = {beginWord: 1}
    end_visit = {endWord: 1}
    queue_begin = deque([beginWord])
    queue_end = deque([endWord])

    def visit(queue, visit_map, other_map):
        word = queue.popleft()
        level = visit_map[word]
        for i in range(len(word)):
            generic = word[:i] + "*" + word[i + 1 :]
            for neighbor in neighbors[generic]:
                if neighbor in other_map:
                    return level + other_map[neighbor]
                if neighbor not in visit_map:
                    visit_map[neighbor] = level + 1
                    queue.append(neighbor)
        return None

    while queue_begin and queue_end:
        ans = visit(queue_begin, begin_visit, end_visit)
        if ans:
            return ans
        ans = visit(queue_end, end_visit, begin_visit)
        if ans:
            return ans
    return 0
```

17 Interval Scheduling Deep Dive

17.1 57. Insert Interval

Key Idea: Merge overlaps by appending segments as you scan. **Complexity:** Time $O(n)$, Space $O(1)$ extra.

```python
def insert(intervals, newInterval):
    result = []
    i = 0
    start, end = newInterval
    while i < len(intervals) and intervals[i][1] < start:
        result.append(intervals[i])
        i += 1
    while i < len(intervals) and intervals[i][0] <= end:
        start = min(start, intervals[i][0])
        end = max(end, intervals[i][1])
        i += 1
    result.append([start, end])
    result.extend(intervals[i:])
    return result
```

17.2 56. Merge Intervals

Key Idea: Sort by start then aggregate overlapping ranges. **Complexity:** Time $O(n \log n)$, Space $O(1)$ extra.

```python
def merge(intervals):
    intervals.sort(key=lambda x: x[0])
    merged = []
    for interval in intervals:
        if not merged or merged[-1][1] < interval[0]:
            merged.append(interval[:])
        else:
            merged[-1][1] = max(merged[-1][1], interval[1])
    return merged
```

17.3 435. Non-overlapping Intervals

Key Idea: Greedy keep earliest finishing interval and count removals. **Complexity:** Time $O(n \log n)$, Space $O(1)$.

```python
def eraseOverlapIntervals(intervals):
    if not intervals:
        return 0
    intervals.sort(key=lambda x: x[1])
    keep_end = intervals[0][1]
    removals = 0
    for start, end in intervals[1:]:
        if start < keep_end:
            removals += 1
        else:
            keep_end = end
    return removals
```

17.4 252. Meeting Rooms

Key Idea: Sort meetings and ensure each start is after prior end. **Complexity:** Time $O(n \log n)$, Space $O(1)$.

```python
def canAttendMeetings(intervals):
    intervals.sort()
    for i in range(1, len(intervals)):
        if intervals[i][0] < intervals[i - 1][1]:
            return False
    return True
```

18 🔗 Linked List Deep Dive

18.1 206. Reverse Linked List

Key Idea: Iteratively flip pointers while tracking previous node. **Complexity:** Time $O(n)$, Space $O(1)$.

```python
def reverseList(head):
    prev = None
    curr = head
    while curr:
        nxt = curr.next
        curr.next = prev
        prev = curr
        curr = nxt
    return prev
```

18.2 21. Merge Two Sorted Lists

Key Idea: Dummy sentinel simplifies tail handling during merge. **Complexity:** Time $O(n + m)$, Space $O(1)$.

```python
def mergeTwoLists(list1, list2):
    dummy = tail = ListNode()
    while list1 and list2:
        if list1.val < list2.val:
            tail.next, list1 = list1, list1.next
        else:
            tail.next, list2 = list2, list2.next
        tail = tail.next
    tail.next = list1 or list2
    return dummy.next
```

18.3 143. Reorder List

Key Idea: Split list, reverse second half, then interleave nodes. **Complexity:** Time $O(n)$, Space $O(1)$.

```python
def reorderList(head):
    if not head or not head.next:
        return
    slow, fast = head, head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    second = slow.next
    slow.next = None

    prev = None
    curr = second
    while curr:
        nxt = curr.next
        curr.next = prev
        prev = curr
        curr = nxt
    second = prev

    first = head
    while second:
        tmp1, tmp2 = first.next, second.next
        first.next = second
        second.next = tmp1
        first, second = tmp1, tmp2
```

18.4 19. Remove Nth Node From End of List

Key Idea: Two-pointer gap of n nodes allows single pass removal. **Complexity:** Time $O(n)$, Space $O(1)$.

```python
def removeNthFromEnd(head, n):
    dummy = ListNode(0, head)
    fast = slow = dummy
    for _ in range(n):
        fast = fast.next
    while fast.next:
        fast = fast.next
        slow = slow.next
    slow.next = slow.next.next
    return dummy.next
```

18.5 141. Linked List Cycle

Key Idea: Floyd's tortoise and hare detects cycle via pointer speeds. **Complexity:** Time $O(n)$, Space $O(1)$.

```python
def hasCycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow is fast:
            return True
    return False
```

19 🪄 Matrix Manipulation Deep Dive

19.1 73. Set Matrix Zeroes

Key Idea: Use first row/column as markers and extra flag for first column. **Complexity:** Time $O(mn)$, Space $O(1)$ extra.

```python
def setZeroes(matrix):
    rows, cols = len(matrix), len(matrix[0])
    first_col_zero = any(matrix[r][0] == 0 for r in range(rows))
    first_row_zero = any(matrix[0][c] == 0 for c in range(cols))

    for r in range(1, rows):
        for c in range(1, cols):
            if matrix[r][c] == 0:
                matrix[r][0] = matrix[0][c] = 0

    for r in range(1, rows):
        for c in range(1, cols):
            if matrix[r][0] == 0 or matrix[0][c] == 0:
                matrix[r][c] = 0

    if first_row_zero:
        for c in range(cols):
            matrix[0][c] = 0
    if first_col_zero:
        for r in range(rows):
            matrix[r][0] = 0
```

19.2 54. Spiral Matrix

Key Idea: Traverse using shrinking boundaries while collecting elements. **Complexity:** Time $O(mn)$, Space $O(1)$ extra.

```python
def spiralOrder(matrix):
    result = []
    top, bottom = 0, len(matrix) - 1
    left, right = 0, len(matrix[0]) - 1
    while top <= bottom and left <= right:
        for c in range(left, right + 1):
            result.append(matrix[top][c])
        top += 1
        for r in range(top, bottom + 1):
            result.append(matrix[r][right])
        right -= 1
        if top <= bottom:
            for c in range(right, left - 1, -1):
                result.append(matrix[bottom][c])
            bottom -= 1
        if left <= right:
            for r in range(bottom, top - 1, -1):
                result.append(matrix[r][left])
            left += 1
    return result
```

19.3 48. Rotate Image

Key Idea: Transpose then reverse each row in-place. **Complexity:** Time $O(n^2)$, Space $O(1)$.

```python
def rotate(matrix):
    n = len(matrix)
    for r in range(n):
        for c in range(r, n):
            matrix[r][c], matrix[c][r] = matrix[c][r], matrix[r][c]
    for row in matrix:
        row.reverse()
```

19.4 79. Word Search

Key Idea: Backtrack with visited marking per DFS path. **Complexity:** Time $O(mn \cdot 4^L)$, Space $O(L)$.

```python
def exist(board, word):
    rows, cols = len(board), len(board[0])

    def dfs(r, c, idx):
        if idx == len(word):
            return True
        if r < 0 or r >= rows or c < 0 or c >= cols:
            return False
        if board[r][c] != word[idx]:
            return False
        temp, board[r][c] = board[r][c], "#"
        found = (
            dfs(r + 1, c, idx + 1)
            or dfs(r - 1, c, idx + 1)
            or dfs(r, c + 1, idx + 1)
            or dfs(r, c - 1, idx + 1)
        )
        board[r][c] = temp
        return found

    for r in range(rows):
        for c in range(cols):
            if dfs(r, c, 0):
                return True
    return False
```

20 String Patterns Deep Dive

20.1 3. Longest Substring Without Repeating Characters

Key Idea: Sliding window with index map to skip duplicates. **Complexity:** Time $O(n)$, Space $O(k)$.

```python
def lengthOfLongestSubstring(s):
    seen = {}
    left = best = 0
    for right, ch in enumerate(s):
        if ch in seen and seen[ch] >= left:
            left = seen[ch] + 1
        seen[ch] = right
        best = max(best, right - left + 1)
    return best
```

20.2 424. Longest Repeating Character Replacement

Key Idea: Expand window, shrink when replacements exceed k using max count. **Complexity:** Time $O(n)$, Space $O(1)$.

```python
from collections import Counter


def characterReplacement(s, k):
    counts = Counter()
    left = best = 0
    max_freq = 0
    for right, ch in enumerate(s):
        counts[ch] += 1
        max_freq = max(max_freq, counts[ch])
        while (right - left + 1) - max_freq > k:
            counts[s[left]] -= 1
            left += 1
        best = max(best, right - left + 1)
    return best
```

20.3 76. Minimum Window Substring

Key Idea: Maintain window counts, contract when window satisfies requirement. **Complexity:** Time $O(n)$, Space $O(1)$ for ASCII.

```python
from collections import Counter


def minWindow(s, t):
    need = Counter(t)
    missing = len(t)
    left = start = end = 0
    for right, ch in enumerate(s, start=1):
        missing -= need[ch] > 0
        need[ch] -= 1
        if missing == 0:
            while left < right and need[s[left]] < 0:
                need[s[left]] += 1
                left += 1
            if not end or right - left < end - start:
                start, end = left, right
            need[s[left]] += 1
            missing += 1
            left += 1
    return s[start:end]
```

20.4 242. Valid Anagram

Key Idea: Frequency counts must match for all letters. **Complexity:** Time $O(n)$, Space $O(1)$ for 26 letters.

```python
from collections import Counter


def isAnagram(s, t):
    return Counter(s) == Counter(t)
```

20.5 49. Group Anagrams

Key Idea: Sort letters or count signature to bucket words. **Complexity:** Time $O(nk \log k)$ with sort, Space $O(nk)$.

```python
from collections import defaultdict


def groupAnagrams(strs):
    groups = defaultdict(list)
    for word in strs:
        key = "".join(sorted(word))
        groups[key].append(word)
    return list(groups.values())
```

20.6 20. Valid Parentheses

Key Idea: Stack ensures matching bracket types and order. **Complexity:** Time $O(n)$, Space $O(n)$.

```python
def isValid(s):
    pairs = {")": "(", "]": "[", "}": "{"}
    stack = []
    for ch in s:
        if ch in pairs.values():
            stack.append(ch)
        else:
            if not stack or stack.pop() != pairs[ch]:
                return False
    return not stack
```

20.7 125. Valid Palindrome

Key Idea: Two pointers skipping non-alphanumeric characters. **Complexity:** Time $O(n)$, Space $O(1)$.

```
def isPalindrome(s):
    left, right = 0, len(s) - 1
    while left < right:
        while left < right and not s[left].isalnum():
            left += 1
        while left < right and not s[right].isalnum():
            right -= 1
        if s[left].lower() != s[right].lower():
            return False
        left += 1
        right -= 1
    return True
```

20.8 5. Longest Palindromic Substring

Key Idea: Expand around every center and record best span. **Complexity:** Time $O(n^2)$, Space $O(1)$.

```
def longestPalindrome(s):
    best = (0, 1)

    def expand(left, right):
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return left + 1, right

    for i in range(len(s)):
        best = max(best, expand(i, i), expand(i, i + 1), key=lambda x: x[1] -
        ↪  x[0])
    return s[best[0] : best[1]]
```

20.9 647. Palindromic Substrings

Key Idea: Expand from each center counting palindromes. **Complexity:** Time $O(n^2)$, Space $O(1)$.

```python
def countSubstrings(s):
    def expand(left, right):
        total = 0
        while left >= 0 and right < len(s) and s[left] == s[right]:
            total += 1
            left -= 1
            right += 1
        return total

    count = 0
    for i in range(len(s)):
        count += expand(i, i)
        count += expand(i, i + 1)
    return count
```

20.10 271. Encode and Decode Strings

Key Idea: Length-prefix avoids delimiter collisions. **Complexity:** Time $O(n)$, Space $O(n)$.

```python
class Codec:
    def encode(self, strs):
        return "".join(f"{len(s)}#{s}" for s in strs)

    def decode(self, s):
        result = []
        i = 0
        while i < len(s):
            j = s.find("#", i)
            length = int(s[i:j])
            j += 1
            result.append(s[j : j + length])
            i = j + length
        return result
```

21 Binary Tree Toolkit Deep Dive

21.1 104. Maximum Depth of Binary Tree

Key Idea: DFS returns depth as 1 + max child depth. **Complexity:** Time $O(n)$, Space $O(h)$ recursion.

```python
def maxDepth(root):
    if not root:
        return 0
    return 1 + max(maxDepth(root.left), maxDepth(root.right))
```

21.2 100. Same Tree

Key Idea: Recursively compare structure and values. **Complexity:** Time $O(n)$, Space $O(h)$.

```python
def isSameTree(p, q):
    if not p and not q:
        return True
    if not p or not q or p.val != q.val:
        return False
    return isSameTree(p.left, q.left) and isSameTree(p.right, q.right)
```

21.3 226. Invert Binary Tree

Key Idea: Swap children recursively or iteratively. **Complexity:** Time $O(n)$, Space $O(h)$.

```python
def invertTree(root):
    if not root:
        return None
    root.left, root.right = invertTree(root.right), invertTree(root.left)
    return root
```

21.4 102. Binary Tree Level Order Traversal

Key Idea: BFS queue collects nodes level by level. **Complexity:** Time $O(n)$, Space $O(n)$.

```
from collections import deque


def levelOrder(root):
    if not root:
        return []
    queue = deque([root])
    levels = []
    while queue:
        level_size = len(queue)
        level = []
        for _ in range(level_size):
            node = queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        levels.append(level)
    return levels
```

21.5 572. Subtree of Another Tree

Key Idea: Check equality at each node; bail early if subtree matches. **Complexity:** Time $O(nm)$ worst, Space $O(h)$.

```
def isSubtree(root, subRoot):
    if not subRoot:
        return True
    if not root:
        return False

    def same(a, b):
        if not a and not b:
            return True
        if not a or not b or a.val != b.val:
            return False
        return same(a.left, b.left) and same(a.right, b.right)

    if same(root, subRoot):
        return True
    return isSubtree(root.left, subRoot) or isSubtree(root.right, subRoot)
```

21.6 105. Construct Binary Tree from Preorder and Inorder Traversal

Key Idea: Preorder gives root; use inorder index map to split subtrees. **Complexity:** Time $O(n)$, Space $O(n)$.

```
def buildTree(preorder, inorder):
    index = {value: i for i, value in enumerate(inorder)}

    def helper(pre_left, pre_right, in_left):
        if pre_left > pre_right:
            return None
        root_val = preorder[pre_left]
        root = TreeNode(root_val)
        in_root = index[root_val]
        left_size = in_root - in_left
        root.left = helper(pre_left + 1, pre_left + left_size, in_left)
        root.right = helper(pre_left + left_size + 1, pre_right, in_root + 1)
        return root

    return helper(0, len(preorder) - 1, 0)
```

21.7 98. Validate Binary Search Tree

Key Idea: Maintain valid min/max bounds per node. **Complexity:** Time $O(n)$, Space $O(h)$.

```
def isValidBST(root):
    def validate(node, low, high):
        if not node:
            return True
        if not (low < node.val < high):
            return False
        return validate(node.left, low, node.val) and validate(node.right,
        ↪  node.val, high)

    return validate(root, float("-inf"), float("inf"))
```

21.8 230. Kth Smallest Element in a BST

Key Idea: Inorder traversal yields increasing order; stop at k. **Complexity:** Time $O(k)$, Space $O(h)$.

```python
def kthSmallest(root, k):
    stack = []
    while True:
        while root:
            stack.append(root)
            root = root.left
        node = stack.pop()
        k -= 1
        if k == 0:
            return node.val
        root = node.right
```

21.9 235. Lowest Common Ancestor of a BST

Key Idea: Use BST ordering to move toward divergence point. **Complexity:** Time $O(h)$, Space $O(1)$.

```python
def lowestCommonAncestor(root, p, q):
    while root:
        if p.val < root.val and q.val < root.val:
            root = root.left
        elif p.val > root.val and q.val > root.val:
            root = root.right
        else:
            return root
```

21.10 124. Binary Tree Maximum Path Sum

Key Idea: Postorder computes max gain from each node while updating global best. **Complexity:** Time $O(n)$, Space $O(h)$.

```python
def maxPathSum(root):
    best = float("-inf")

    def gain(node):
        nonlocal best
        if not node:
            return 0
        left = max(gain(node.left), 0)
        right = max(gain(node.right), 0)
        best = max(best, node.val + left + right)
        return node.val + max(left, right)

    gain(root)
    return best
```

21.11 297. Serialize and Deserialize Binary Tree

Key Idea: Preorder traversal with sentinel markers for null nodes. **Complexity:** Time $O(n)$, Space $O(n)$.

```python
class Codec:
    def serialize(self, root):
        values = []

        def dfs(node):
            if not node:
                values.append("#")
                return
            values.append(str(node.val))
            dfs(node.left)
            dfs(node.right)

        dfs(root)
        return " ".join(values)

    def deserialize(self, data):
        values = iter(data.split())

        def dfs():
            val = next(values)
            if val == "#":
                return None
            node = TreeNode(int(val))
            node.left = dfs()
```

```
            node.right = dfs()
            return node

        return dfs()
```

22 💎 Heaps & Priority Queues Deep Dive

22.1 23. Merge k Sorted Lists

Key Idea: Push the head of each list into a min-heap and stream smallest nodes. **Complexity:** Time $O(n \log k)$, Space $O(k)$.

```python
import heapq


def mergeKLists(lists):
    heap = []
    for i, node in enumerate(lists):
        if node:
            heapq.heappush(heap, (node.val, i, node))

    dummy = tail = ListNode()
    while heap:
        _, i, node = heapq.heappop(heap)
        tail.next = node
        tail = tail.next
        if node.next:
            heapq.heappush(heap, (node.next.val, i, node.next))
    return dummy.next
```

22.2 347. Top K Frequent Elements

Key Idea: Use bucket sort or heap; here we use bucket array for linear time. **Complexity:** Time $O(n)$, Space $O(n)$.

```python
from collections import Counter


def topKFrequent(nums, k):
    freq = Counter(nums)
    buckets = [[] for _ in range(len(nums) + 1)]
    for num, count in freq.items():
        buckets[count].append(num)
    result = []
    for count in range(len(buckets) - 1, -1, -1):
        for num in buckets[count]:
            result.append(num)
            if len(result) == k:
                return result
```

22.3 295. Find Median from Data Stream

Key Idea: Two heaps maintain lower and upper halves balanced by size. **Complexity:** Time $O(\log n)$ per insert, Space $O(n)$.

```python
import heapq


class MedianFinder:
    def __init__(self):
        self.low = []   # max-heap via negated values
        self.high = []  # min-heap

    def addNum(self, num):
        heapq.heappush(self.low, -num)
        heapq.heappush(self.high, -heapq.heappop(self.low))
        if len(self.high) > len(self.low):
            heapq.heappush(self.low, -heapq.heappop(self.high))

    def findMedian(self):
        if len(self.low) > len(self.high):
            return -self.low[0]
        return (-self.low[0] + self.high[0]) / 2
```

23 🎭 Backtracking Patterns Deep Dive

23.1 78. Subsets

Key Idea: DFS explores include/exclude choices building powerset. **Complexity:** Time $O(n2^n)$, Space $O(n)$.

```python
def subsets(nums):
    result = []
    subset = []

    def dfs(index):
        if index == len(nums):
            result.append(subset[:])
            return
        subset.append(nums[index])
        dfs(index + 1)
        subset.pop()
        dfs(index + 1)

    dfs(0)
    return result
```

23.2 39. Combination Sum

Key Idea: Backtrack with current sum and allow reuse of the same candidate. **Complexity:** Time exponential, Space $O(target/\min(nums))$ recursion.

```python
def combinationSum(candidates, target):
    result = []
    path = []

    def dfs(index, remaining):
        if remaining == 0:
            result.append(path[:])
            return
        if remaining < 0 or index == len(candidates):
            return
        path.append(candidates[index])
        dfs(index, remaining - candidates[index])
        path.pop()
        dfs(index + 1, remaining)

    dfs(0, target)
    return result
```

23.3 46. Permutations

Key Idea: Swap in-place to build permutations without extra visited set. **Complexity:** Time $O(n \cdot n!)$, Space $O(n)$ recursion.

```python
def permute(nums):
    result = []

    def dfs(start):
        if start == len(nums):
            result.append(nums[:])
            return
        for i in range(start, len(nums)):
            nums[start], nums[i] = nums[i], nums[start]
            dfs(start + 1)
            nums[start], nums[i] = nums[i], nums[start]

    dfs(0)
    return result
```

23.4 131. Palindrome Partitioning

Key Idea: Backtrack on prefixes while pruning non-palindromes. **Complexity:** Time exponential, Space $O(n)$ recursion plus output.

```python
def partition(s):
    result = []
    path = []

    def dfs(start):
        if start == len(s):
            result.append(path[:])
            return
        for end in range(start + 1, len(s) + 1):
            if s[start:end] == s[start:end][::-1]:
                path.append(s[start:end])
                dfs(end)
                path.pop()

    dfs(0)
    return result
```

23.5 17. Letter Combinations of a Phone Number

Key Idea: Map digits to letters and build strings recursively. **Complexity:** Time $O(4^n)$, Space $O(n)$.

```
def letterCombinations(digits):
    if not digits:
        return []
    mapping = {
        "2": "abc",
        "3": "def",
        "4": "ghi",
        "5": "jkl",
        "6": "mno",
        "7": "pqrs",
        "8": "tuv",
        "9": "wxyz",
    }
    result = []

    def dfs(index, path):
        if index == len(digits):
            result.append("".join(path))
            return
        for ch in mapping[digits[index]]:
            path.append(ch)
            dfs(index + 1, path)
            path.pop()

    dfs(0, [])
    return result
```