Coding Interview Patterns - Complete Revision Guide

🎯 Quick Pattern Recognition Flowchart

```
Start → Read Problem
        ↓
Is it about arrays/strings?
 ├─YES→ Contiguous elements? → Sliding Window
 |    Sorted array? → Two Pointers / Binary Search
 |    Find duplicates/missing? → Cyclic Sort
 |    Subarray/Substring? → Sliding Window / Kadane's
 |
 ├─Is it about intervals? → Merge Intervals
 |
 ├─Is it about linked lists?
 |  ├─ Reverse? → In-place Reversal
 |  └─ Cycle? → Fast & Slow Pointers
 |
 ├─Is it about trees/graphs?
 |  ├─ Level by level? → BFS
 |  ├─ Path finding? → DFS
 |  └─ Dependencies? → Topological Sort
 |
 ├─Is it about optimization?
 |  ├─ Best K elements? → Heap / Top K
 |  └─ Min/Max with choices? → Dynamic Programming
 |
 └─Is it about combinations? → Subsets / Backtracking
```

📚 Pattern 1: SLIDING WINDOW

🔑 Key Concept

Maintain a window that slides over data to avoid recalculating overlapping elements.

🎯 Recognition Triggers

- Contiguous subarray/substring
- "Maximum/minimum sum of size K"
- "Longest/shortest substring with condition"
- "Window of size K"

💡 Core Template

```python
def sliding_window(arr):
    window_start = 0
    window_sum = 0
    max_sum = float('-inf')

    for window_end in range(len(arr)):
        window_sum += arr[window_end]  # Expand window

        if window_end >= k - 1:  # Window size reached
            max_sum = max(max_sum, window_sum)
            window_sum -= arr[window_start]  # Shrink window
            window_start += 1

    return max_sum
```

📝 Essential Problems

Problem	Difficulty	Key Trick
Maximum Sum Subarray of Size K	Easy	Fixed window, track sum
Longest Substring with K Distinct Characters	Medium	HashMap to track chars, shrink when > K
Fruits into Baskets	Medium	Same as K distinct with K=2
Longest Substring Without Repeating	Medium	HashMap, shrink when duplicate found
Permutation in String	Hard	Character frequency map comparison
Minimum Window Substring	Hard	Two hashmaps, expand then shrink

⚡ Quick Tips

- Use hashmap for character/element frequency
- Expand first, then shrink to maintain valid window
- For fixed size: slide when window_end >= K-1
- For variable size: use while loop to shrink

📚 Pattern 2: TWO POINTERS

🔑 Key Concept

Use two pointers to traverse array/list efficiently, often avoiding nested loops.

🎯 Recognition Triggers

- Sorted arrays

- Find pair/triplet with target sum
- Compare elements from both ends
- Remove duplicates in-place

💡 Core Template

```python
def two_pointers(arr, target):
    left, right = 0, len(arr) - 1

    while left < right:
        current_sum = arr[left] + arr[right]
        if current_sum == target:
            return [left, right]
        elif current_sum < target:
            left += 1  # Need larger sum
        else:
            right -= 1  # Need smaller sum

    return []
```

📝 Essential Problems

Problem	Difficulty	Key Trick
Two Sum (sorted)	Easy	Move pointers based on sum vs target
Remove Duplicates	Easy	Fast pointer scans, slow pointer places
Square a Sorted Array	Easy	Two pointers from ends, compare absolute values
3Sum	Medium	Fix one pointer, two-pointer for rest
3Sum Closest	Medium	Track minimum difference
Dutch National Flag	Medium	Three pointers for three regions
Container With Most Water	Medium	Move pointer with smaller height

⚡ Quick Tips

- Array usually needs to be sorted
- For triplets: fix one, use two pointers for pair
- Skip duplicates: `while left < right and arr[left] == arr[left-1]: left += 1`

📚 Pattern 3: FAST & SLOW POINTERS

🔑 Key Concept

Two pointers moving at different speeds; they meet if there's a cycle.

🎯 Recognition Triggers

- Detect cycle in linked list
- Find middle element
- Find cycle start
- Happy number problem

💡 Core Template

```python
def has_cycle(head):
    slow = fast = head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True  # Cycle detected

    return False

def find_middle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow  # Slow is at middle
```

📝 Essential Problems

Problem	Difficulty	Key Trick
Linked List Cycle	Easy	Fast meets slow = cycle
Middle of Linked List	Easy	When fast ends, slow is at middle
Linked List Cycle II	Medium	Find meet point, then find cycle start
Happy Number	Easy	Treat as cycle detection in sequence
Palindrome Linked List	Medium	Find middle, reverse second half, compare
Reorder List	Medium	Find middle, reverse second half, merge

⚡ Quick Tips

- Fast moves 2x speed of slow
- To find cycle start: Reset one pointer to head after meeting, move both by 1
- Middle finding: Slow is at middle when fast reaches end

📚 Pattern 4: MERGE INTERVALS

🔑 Key Concept

Sort intervals by start time, then merge overlapping ones.

🎯 Recognition Triggers

- Overlapping intervals
- Meeting rooms
- Insert interval
- Time conflicts

💡 Core Template

python

```python
def merge_intervals(intervals):
    if not intervals:
        return []

    intervals.sort(key=lambda x: x[0])  # Sort by start
    merged = [intervals[0]]

    for i in range(1, len(intervals)):
        last = merged[-1]
        current = intervals[i]

        if current[0] <= last[1]:  # Overlapping
            merged[-1] = [last[0], max(last[1], current[1])]
        else:
            merged.append(current)

    return merged
```

📝 Essential Problems

Problem	Difficulty	Key Trick
Merge Intervals	Medium	Sort, then merge if start <= prev_end
Insert Interval	Medium	Merge new interval with overlapping ones
Intervals Intersection	Medium	Two pointers, find overlap of each pair
Meeting Rooms	Easy	Sort, check if any overlap
Meeting Rooms II	Medium	Min heap for end times
Maximum CPU Load	Medium	Min heap, track concurrent tasks

⚡ Quick Tips

- Always sort by start time first
- Overlap condition: `current.start <= previous.end`
- For minimum meeting rooms: use heap to track end times

📚 Pattern 5: CYCLIC SORT

🔑 Key Concept

When dealing with arrays containing numbers in range [1, n], place each number at its correct index.

🎯 Recognition Triggers

- Numbers from 1 to n
- Find missing/duplicate numbers
- Find corrupt pair
- First missing positive

💡 Core Template

python

```python
def cyclic_sort(nums):
    i = 0
    while i < len(nums):
        correct_index = nums[i] - 1  # For 1 to n range
        # Swap if not at correct position
        if nums[i] != nums[correct_index]:
            nums[i], nums[correct_index] = nums[correct_index], nums[i]
        else:
            i += 1
    return nums
```

📝 Essential Problems

Problem	Difficulty	Key Trick
Cyclic Sort	Easy	Place number i at index i-1
Find Missing Number	Easy	After sort, index != value-1
Find All Missing Numbers	Easy	Collect all index != value-1
Find Duplicate	Easy	Number not at correct position after sort
Find Corrupt Pair	Easy	Find duplicate and missing together
First Missing Positive	Hard	Ignore numbers outside [1, n] range

⚡ Quick Tips

- Number n should be at index n-1
- After sorting, scan to find anomalies
- Handle duplicates by checking if target position already correct

---

📚 Pattern 6: IN-PLACE REVERSAL OF LINKED LIST

🔑 Key Concept

Reverse pointers without extra space using previous, current, next tracking.

🎯 Recognition Triggers

- Reverse linked list
- Reverse sub-list
- Reverse in groups
- Rotate list

💡 Core Template

```python
```

```python
def reverse_linked_list(head):
    prev = None
    current = head

    while current:
        next_temp = current.next  # Save next
        current.next = prev       # Reverse pointer
        prev = current           # Move prev forward
        current = next_temp      # Move current forward

    return prev  # New head

def reverse_between(head, left, right):
    # 1. Reach the left position
    # 2. Reverse the sub-list
    # 3. Connect back properly
```

📝 Essential Problems

Problem	Difficulty	Key Trick
Reverse Linked List	Easy	Track prev, current, next
Reverse Sub-list	Medium	Find start, reverse portion, reconnect
Reverse Every K-Group	Hard	Reverse k nodes, check if k nodes exist
Reverse Alternating K-Group	Medium	Reverse k, skip k, repeat
Rotate List	Medium	Make circular, then break at n-k

⚡ Quick Tips

- Always save next before reversing pointer
- For sub-list: track node before start
- For k-group: check if k nodes available first

📚 Pattern 7: TREE BFS (Level Order Traversal)

🔑 Key Concept

Use queue to traverse tree level by level.

🎯 Recognition Triggers

- Level order traversal
- Level averages/sums
- Minimum depth

- Zigzag traversal
- Connect level siblings

💡 Core Template

```python
from collections import deque

def level_order(root):
    if not root:
        return []

    result = []
    queue = deque([root])

    while queue:
        level_size = len(queue)
        current_level = []

        for _ in range(level_size):
            node = queue.popleft()
            current_level.append(node.val)

            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        result.append(current_level)

    return result
```

📝 Essential Problems

Problem	Difficulty	Key Trick
Binary Tree Level Order	Medium	Process all nodes at current level
Reverse Level Order	Easy	Insert levels at beginning of result
Zigzag Traversal	Medium	Alternate append direction per level
Level Averages	Easy	Sum level / level size
Minimum Depth	Easy	Return when first leaf found
Level Order Successor	Easy	Return next node after finding target
Connect Level Order Siblings	Medium	Connect nodes at same level

⚡ Quick Tips

- Use queue size to process exactly one level
- For zigzag: use deque, alternate append left/right
- For minimum depth: return as soon as leaf found

📚 Pattern 8: TREE DFS

🔑 Key Concept

Explore paths from root to leaves using recursion or stack.

🎯 Recognition Triggers

- Path sum problems
- Root-to-leaf paths
- Tree diameter
- Count paths
- Maximum path sum

💡 Core Template

```python
```

```python
def has_path_sum(root, target):
    if not root:
        return False

    # Leaf node check
    if not root.left and not root.right:
        return root.val == target

    # Recursive check
    remaining = target - root.val
    return (has_path_sum(root.left, remaining) or
            has_path_sum(root.right, remaining))

def find_paths(root, target):
    all_paths = []

    def dfs(node, remaining, path):
        if not node:
            return

        path.append(node.val)

        if not node.left and not node.right and remaining == node.val:
            all_paths.append(list(path))
        else:
            dfs(node.left, remaining - node.val, path)
            dfs(node.right, remaining - node.val, path)

        path.pop()  # Backtrack

    dfs(root, target, [])
    return all_paths
```

📝 Essential Problems

Problem	Difficulty	Key Trick
Binary Tree Path Sum	Easy	Check if leaf and sum equals target
All Paths for Sum	Medium	Collect all paths, backtrack
Sum of Path Numbers	Medium	Form number from root to leaf
Path With Given Sequence	Medium	Match sequence while traversing
Count Paths for Sum	Medium	Count paths from any node
Tree Diameter	Medium	Max(left_height + right_height) at each node
Path with Maximum Sum	Hard	Consider including/excluding paths through node

⚡ Quick Tips

- For all paths: use backtracking with path.pop()
- For any node start: run DFS from every node
- For diameter/max sum: track global maximum

📚 Pattern 9: TWO HEAPS

🔑 Key Concept

Use max heap for smaller half and min heap for larger half to track median.

🎯 Recognition Triggers

- Find median from stream
- Sliding window median
- Maximize capital

💡 Core Template

```python
```

```python
import heapq

class MedianFinder:
    def __init__(self):
        self.small = []  # max heap (negate values)
        self.large = []  # min heap

    def add_num(self, num):
        # Add to small heap first
        heapq.heappush(self.small, -num)

        # Ensure max of small <= min of large
        if self.small and self.large:
            if -self.small[0] > self.large[0]:
                val = -heapq.heappop(self.small)
                heapq.heappush(self.large, val)

        # Balance sizes (small can have 1 extra)
        if len(self.small) > len(self.large) + 1:
            val = -heapq.heappop(self.small)
            heapq.heappush(self.large, val)
        elif len(self.large) > len(self.small):
            val = heapq.heappop(self.large)
            heapq.heappush(self.small, -val)

    def find_median(self):
        if len(self.small) > len(self.large):
            return -self.small[0]
        return (-self.small[0] + self.large[0]) / 2.0
```

📝 Essential Problems

Problem	Difficulty	Key Trick
Find Median from Data Stream	Hard	Balance two heaps
Sliding Window Median	Hard	Remove elements when sliding
Maximize Capital (IPO)	Hard	Min heap for capital, max heap for profits

⚡ Quick Tips

- Small heap (max) has smaller half of numbers
- Large heap (min) has larger half
- Keep sizes balanced (differ by at most 1)

Pattern 10: SUBSETS (BACKTRACKING)

Key Concept

Build solution incrementally, explore all possibilities using include/exclude at each step.

Recognition Triggers

- Generate all subsets
- Generate permutations
- Generate combinations
- Letter combinations
- Generate parentheses

Core Template

python

```python
# Subsets - Include/Exclude Pattern
def find_subsets(nums):
    result = []

    def backtrack(start, path):
        result.append(path[:])  # Add current subset

        for i in range(start, len(nums)):
            path.append(nums[i])  # Include
            backtrack(i + 1, path)  # Explore
            path.pop()  # Exclude (backtrack)

    backtrack(0, [])
    return result

# Permutations
def permute(nums):
    result = []

    def backtrack(path, remaining):
        if not remaining:
            result.append(path[:])
            return

        for i in range(len(remaining)):
            path.append(remaining[i])
            backtrack(path, remaining[:i] + remaining[i+1:])
            path.pop()

    backtrack([], nums)
    return result
```

📝 Essential Problems

Problem	Difficulty	Key Trick
Subsets	Medium	Include/exclude each element
Subsets with Duplicates	Medium	Sort first, skip duplicates
Permutations	Medium	Use each element once per position
String Permutations by Case	Medium	For each letter, try upper and lower
Generate Parentheses	Medium	Track open and close counts
Letter Combinations	Medium	BFS or DFS through digit mappings
Combination Sum	Medium	Can reuse elements, start from current index

⚡ Quick Tips

- For subsets: iterate and add element to all existing subsets
- For duplicates: sort first, skip if `nums[i] == nums[i-1]`
- For permutations: swap elements or use visited array

📚 Pattern 11: MODIFIED BINARY SEARCH

🔑 Key Concept

Adapt binary search for rotated arrays, infinite arrays, or finding boundaries.

🎯 Recognition Triggers

- Sorted or rotated array
- Find in infinite array
- Search for range
- Find peak/minimum
- Search in matrix

💡 Core Template

```python
```

```python
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1

def search_rotated(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2

        if arr[mid] == target:
            return mid

        # Check which half is sorted
        if arr[left] <= arr[mid]:  # Left sorted
            if arr[left] <= target < arr[mid]:
                right = mid - 1
            else:
                left = mid + 1
        else:  # Right sorted
            if arr[mid] < target <= arr[right]:
                left = mid + 1
            else:
                right = mid - 1

    return -1
```

📝 Essential Problems

Problem	Difficulty	Key Trick
Order-agnostic Binary Search	Easy	Check if ascending or descending
Ceiling of Number	Easy	Find smallest >= target
Next Letter	Medium	Wrap around with modulo
Number Range	Medium	Find first and last position

Problem	Difficulty	Key Trick
Search in Sorted Infinite Array	Medium	Double the bounds to find range
Minimum Difference Element	Medium	Binary search, compare neighbors
Bitonic Array Maximum	Easy	Peak where arr[mid] > both neighbors
Search in Rotated Array	Medium	Identify which half is sorted
Rotation Count	Medium	Find index of minimum element

⚡ Quick Tips

- For rotated: identify which half is sorted
- For ceiling/floor: return left/right pointer after loop
- For infinite array: find bounds first by doubling

📚 Pattern 12: TOP K ELEMENTS

🔑 Key Concept

Use heap of size K to efficiently find K largest/smallest elements.

🎯 Recognition Triggers

- K largest/smallest
- K most frequent
- K closest points
- Kth smallest/largest

💡 Core Template

```
python
```

```python
import heapq

def find_k_largest_numbers(nums, k):
    min_heap = []

    # Keep heap of size k
    for num in nums:
        heapq.heappush(min_heap, num)
        if len(min_heap) > k:
            heapq.heappop(min_heap)

    return min_heap

def find_k_smallest_numbers(nums, k):
    max_heap = []

    for num in nums:
        heapq.heappush(max_heap, -num)  # Negate for max heap
        if len(max_heap) > k:
            heapq.heappop(max_heap)

    return [-x for x in max_heap]
```

📝 Essential Problems

Problem	Difficulty	Key Trick
Kth Smallest Number	Easy	Min heap of size k, return root
K Closest Points to Origin	Medium	Max heap based on distance
Connect Ropes	Easy	Min heap, combine two smallest
Top K Frequent Numbers	Medium	Count frequency, heap by frequency
Frequency Sort	Medium	Bucket sort or heap
Kth Largest in Stream	Medium	Maintain min heap of size k
K Closest Numbers	Medium	Binary search + two pointers
Maximum Distinct Elements	Medium	Keep frequent, heap for rest

⚡ Quick Tips

- For K largest: use MIN heap of size K
- For K smallest: use MAX heap of size K
- Python: negate values for max heap
- Heap[0] gives Kth largest/smallest

📚 Pattern 13: K-WAY MERGE

🔑 Key Concept

Use min heap to efficiently merge K sorted arrays/lists.

🎯 Recognition Triggers

- Merge K sorted lists
- Kth smallest in M sorted lists
- Smallest range covering K lists

💡 Core Template

```python
import heapq

def merge_k_sorted_lists(lists):
    min_heap = []

    # Add first element from each list
    for i in range(len(lists)):
        if lists[i]:
            heapq.heappush(min_heap, (lists[i][0], i, 0))

    result = []
    while min_heap:
        val, list_idx, elem_idx = heapq.heappop(min_heap)
        result.append(val)

        # Add next element from same list
        if elem_idx + 1 < len(lists[list_idx]):
            next_elem = lists[list_idx][elem_idx + 1]
            heapq.heappush(min_heap, (next_elem, list_idx, elem_idx + 1))

    return result
```

📝 Essential Problems

Problem	Difficulty	Key Trick
Merge K Sorted Lists	Hard	Heap with (value, list_index, element_index)
Kth Smallest in M Sorted Arrays	Hard	Same as merge, stop at kth
Kth Smallest in Sorted Matrix	Hard	Treat rows or columns as sorted lists
Smallest Range Covering K Lists	Hard	Track min/max in current window

Problem	Difficulty	Key Trick
Find K Pairs with Smallest Sums	Medium	Heap with sum and indices

⚡ Quick Tips

- Heap elements: (value, list_index, element_index)
- Always add next element from same list
- For range problems: track current min/max

📚 Pattern 14: DYNAMIC PROGRAMMING

🔑 Key Concept

Break problem into overlapping subproblems, store solutions to avoid recomputation.

🎯 Recognition Triggers

- Optimization (min/max)
- Count distinct ways
- Make decisions at each step
- Can't use greedy
- Overlapping subproblems

💡 Core Templates

Top-Down (Memoization)

```python
def fibonacci_memo(n):
    memo = {}

    def dp(n):
        if n <= 1:
            return n
        if n in memo:
            return memo[n]

        memo[n] = dp(n-1) + dp(n-2)
        return memo[n]

    return dp(n)
```

Bottom-Up (Tabulation)

```python
def fibonacci_tab(n):
    if n <= 1:
        return n

    dp = [0] * (n + 1)
    dp[0], dp[1] = 0, 1

    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]
```

📝 Common DP Patterns

Pattern Type	Recognition	Example Problems	State Variables
0/1 Knapsack	Each item: take or leave	Subset Sum, Equal Partition	(index, capacity)
Unbounded Knapsack	Unlimited use of items	Coin Change, Rod Cutting	(capacity)
Fibonacci	Current depends on previous	House Robber, Climbing Stairs	(n)
Palindromes	Substring/subsequence	Longest Palindrome Subsequence	(i, j)
LCS Pattern	Two sequences	Longest Common Subsequence	(i, j)
LIS Pattern	Increasing subsequence	Longest Increasing Subsequence	(i) or (i, prev)
Kadane's	Maximum subarray	Maximum Subarray Sum	current_max, global_max
Grid Traversal	Paths in matrix	Unique Paths, Min Path Sum	(row, col)

📝 Essential Problems

Problem	Difficulty	Pattern	Key Insight
Fibonacci Number	Easy	Fibonacci	dp[i] = dp[i-1] + dp[i-2]
Climbing Stairs	Easy	Fibonacci	Same as fibonacci
House Robber	Medium	Fibonacci variant	Max(rob current + i-2, skip to i-1)
0/1 Knapsack	Medium	0/1 Knapsack	Include or exclude each item
Subset Sum	Medium	0/1 Knapsack	Can we make target sum?
Coin Change	Medium	Unbounded Knapsack	Min coins for amount

Problem	Difficulty	Pattern	Key Insight
Longest Common Subsequence	Medium	LCS	Match or skip character
Longest Increasing Subsequence	Medium	LIS	Include if greater than prev
Edit Distance	Hard	2D DP	Insert/delete/replace
Maximum Subarray	Easy	Kadane's	Local max vs global max
Unique Paths	Medium	Grid	Paths = top + left
Jump Game	Medium	Greedy/DP	Can reach position?

⚡ Quick Tips

- Start with recursive solution, then add memo
- Identify state variables (what defines subproblem)
- Draw decision tree for small input
- For optimization: dp[i] = optimize(choices)
- For counting: dp[i] = sum(ways)
- Space optimization: Often can use just 2 rows/variables

📚 Pattern 15: TOPOLOGICAL SORT

🔑 Key Concept

Linear ordering of vertices in Directed Acyclic Graph (DAG) where u comes before v if edge u→v exists.

🎯 Recognition Triggers

- Prerequisites/dependencies
- Build order
- Course schedule
- Alien dictionary

💡 Core Template (Kahn's Algorithm)

python

```python
from collections import deque, defaultdict

def topological_sort(vertices, edges):
    # Build graph and in-degree
    graph = defaultdict(list)
    in_degree = {i: 0 for i in range(vertices)}

    for parent, child in edges:
        graph[parent].append(child)
        in_degree[child] += 1

    # Find sources (in-degree = 0)
    sources = deque([v for v in in_degree if in_degree[v] == 0])
    sorted_order = []

    while sources:
        vertex = sources.popleft()
        sorted_order.append(vertex)

        # Reduce in-degree of children
        for child in graph[vertex]:
            in_degree[child] -= 1
            if in_degree[child] == 0:
                sources.append(child)

    # Check if topological sort possible
    if len(sorted_order) != vertices:
        return []  # Cycle exists

    return sorted_order
```

📝 Essential Problems

Problem	Difficulty	Key Trick
Topological Sort	Medium	Basic Kahn's algorithm
Course Schedule	Medium	Detect if cycle exists
Course Schedule II	Medium	Return actual order
Alien Dictionary	Hard	Build graph from word order
Sequence Reconstruction	Hard	Check if unique topological sort
Minimum Height Trees	Hard	Start from leaves, remove layer by layer

⚡ Quick Tips

- In-degree = number of incoming edges

- Start with vertices having 0 in-degree
- If sorted_order.length != vertices: cycle exists
- For all possible sorts: use DFS backtracking

🔥 BONUS PATTERNS

📚 Pattern 16: TRIE (Prefix Tree)

🔑 Key Concept

Tree-like data structure for efficient string prefix operations.

💡 Core Template

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end
```

📝 Essential Problems

- Implement Trie
- Word Search II

- Design Add and Search Words
- Replace Words

📚 Pattern 17: UNION FIND (Disjoint Set)

🔑 Key Concept

Track connected components, detect cycles in undirected graphs.

💡 Core Template

python

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # Path compression
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px == py:
            return False
        # Union by rank
        if self.rank[px] < self.rank[py]:
            self.parent[px] = py
        elif self.rank[px] > self.rank[py]:
            self.parent[py] = px
        else:
            self.parent[py] = px
            self.rank[px] += 1
        return True
```

📝 Essential Problems

- Number of Islands (alternative solution)
- Graph Valid Tree
- Number of Connected Components
- Redundant Connection

💪 Problem-Solving Strategy

1️⃣ UNDERSTAND (2-3 min)

- [ ] Read problem carefully
- [ ] Identify input/output types
- [ ] Ask clarifying questions
- [ ] Discuss edge cases

2️⃣ MATCH (1-2 min)

- [ ] Identify problem pattern
- [ ] Recall similar problems
- [ ] Consider multiple approaches

3️⃣ PLAN (3-5 min)

- [ ] Explain approach
- [ ] Discuss time/space complexity
- [ ] Write pseudocode if complex

4️⃣ IMPLEMENT (15-20 min)

- [ ] Write clean code
- [ ] Use meaningful variable names
- [ ] Handle edge cases

5️⃣ TEST (3-5 min)

- [ ] Test with example
- [ ] Test edge cases
- [ ] Fix any bugs

6️⃣ OPTIMIZE (2-3 min)

- [ ] Discuss optimizations
- [ ] Consider trade-offs

🎯 Time Complexity Patterns

Pattern	Typical Time	Space
Sliding Window	O(n)	O(1) or O(k)
Two Pointers	O(n) or O(n²)	O(1)
Fast & Slow	O(n)	O(1)
Merge Intervals	O(n log n)	O(n)

Pattern	Typical Time	Space
Cyclic Sort	O(n)	O(1)
Tree BFS	O(n)	O(w) width
Tree DFS	O(n)	O(h) height
Two Heaps	O(log n) insert	O(n)
Subsets	$O(2^n)$	$O(2^n)$
Modified Binary Search	O(log n)	O(1)
Top K Elements	O(n log k)	O(k)
K-way Merge	O(n log k)	O(k)
Dynamic Programming	$O(n^2)$ or O(n·m)	O(n) or O(n·m)
Topological Sort	O(V + E)	O(V + E)

🚀 Last-Minute Review Checklist

Before Interview

- [ ] Review this pattern guide
- [ ] Practice writing core templates from memory
- [ ] Review your weak patterns
- [ ] Do 2-3 warm-up problems

During Interview

- [ ] Stay calm, think aloud
- [ ] Start with brute force if stuck
- [ ] Write clean code first, optimize later
- [ ] Test your code thoroughly

Common Mistakes to Avoid

- ❌ Jumping to code without planning
- ❌ Not handling edge cases
- ❌ Forgetting to test the code
- ❌ Over-complicating the solution
- ❌ Not asking clarifying questions

📖 Recommended Practice Order

Week 1: Foundation

1. Two Pointers (5 problems)
2. Fast & Slow Pointers (3 problems)
3. Sliding Window (5 problems)

Week 2: Sorting & Searching

4. Merge Intervals (4 problems)
5. Cyclic Sort (3 problems)
6. Binary Search (5 problems)

Week 3: Trees & Graphs

7. Tree BFS (5 problems)
8. Tree DFS (5 problems)
9. Two Heaps (2 problems)

Week 4: Advanced

10. Subsets (5 problems)
11. Top K Elements (5 problems)
12. K-way Merge (2 problems)
13. Topological Sort (3 problems)

Week 5+: Mastery

14. Dynamic Programming (10+ problems)
15. Practice mixed problems
16. Focus on weak areas

📝 Notes Section

Use this space to add your own notes, problem-solving tricks, and insights as you practice.

Good luck with your preparation! 🎯 Remember: **Pattern recognition + Practice = Success!**