Essential Problem-Solving Patterns for Coding Interviews

1. Two Pointers Pattern

When to Use

- Working with sorted arrays or linked lists
- Finding pairs with a specific sum/difference
- Comparing elements from both ends

Key Characteristics

- Uses two pointers moving towards each other or in the same direction
- Often $O(n)$ time complexity instead of $O(n^2)$

Common Problems

- Two Sum (sorted array)
- Three Sum
- Container With Most Water
- Remove Duplicates from Sorted Array

Basic Template

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

    while left < right:
        # Process current pair
        if condition_met:
            return result
        elif need_smaller_sum:
            right -= 1
        else:
            left += 1
```

2. Sliding Window Pattern

When to Use

- Dealing with subarrays/substrings
- Finding longest/shortest substring with constraints

- Fixed or variable window size problems

Key Characteristics

- Maintains a window that slides through the data
- Expands and contracts based on conditions
- Tracks window state with variables/hash maps

Common Problems

- Maximum Sum Subarray of Size K
- Longest Substring Without Repeating Characters
- Minimum Window Substring
- Fruits Into Baskets

Basic Template

python

```python
def sliding_window(arr):
    window_start = 0
    window_sum = 0
    max_length = 0

    for window_end in range(len(arr)):
        # Add element at window_end
        window_sum += arr[window_end]

        # Shrink window if needed
        while window_invalid:
            window_sum -= arr[window_start]
            window_start += 1

        # Update result
        max_length = max(max_length, window_end - window_start + 1)
```

3. Fast & Slow Pointers (Floyd's Algorithm)

When to Use

- Detecting cycles in linked lists or arrays
- Finding middle element
- Finding kth element from end

Key Characteristics

- Two pointers moving at different speeds
- Fast pointer moves twice as fast as slow
- They meet if there's a cycle

Common Problems

- Linked List Cycle
- Find Middle of Linked List
- Happy Number
- Cycle in Circular Array

Basic Template

python

```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

    return False
```

4. Merge Intervals Pattern

When to Use

- Dealing with overlapping intervals
- Finding gaps between intervals
- Merging overlapping ranges

Key Characteristics

- Sort intervals by start time
- Compare current interval with previous
- Merge if overlapping

Common Problems

- Merge Intervals

- Insert Interval
- Intervals Intersection
- Meeting Rooms

Basic Template

python

```python
def merge_intervals(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
```

5. Cyclic Sort Pattern

When to Use

- Array contains numbers in range 1 to n
- Finding missing/duplicate numbers
- O(n) time and O(1) space required

Key Characteristics

- Places each number at its correct index
- Numbers are in a specific range
- In-place sorting

Common Problems

- Missing Number
- Find All Duplicates
- Find the Duplicate Number
- First Missing Positive

Basic Template

python

```python
def cyclic_sort(nums):
    i = 0
    while i < len(nums):
        correct_index = nums[i] - 1
        if nums[i] != nums[correct_index]:
            nums[i], nums[correct_index] = nums[correct_index], nums[i]
        else:
            i += 1
```

6. In-place Reversal of LinkedList

When to Use

- Reversing linked list or its sub-parts
- Changing link directions
- Space complexity must be O(1)

Key Characteristics

- Changes next pointers
- Uses temporary variables
- Often combined with other patterns

Common Problems

- Reverse Linked List
- Reverse Linked List II
- Reverse Nodes in k-Group
- Rotate List

Basic Template

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

    while current:
        next_temp = current.next
        current.next = prev
        prev = current
        current = next_temp

    return prev
```

7. Tree BFS Pattern

When to Use

- Level-by-level traversal
- Finding shortest path in unweighted tree
- Zigzag traversal

Key Characteristics

- Uses queue data structure
- Processes nodes level by level
- Can track level information

Common Problems

- Binary Tree Level Order Traversal
- Zigzag Traversal
- Minimum Depth of Binary Tree
- Connect Level Order Siblings

Basic Template

```python
from collections import deque

def level_order_traversal(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
```

8. Tree DFS Pattern

When to Use

- Searching for path with specific properties
- Computing tree properties recursively
- Pre-order, in-order, post-order traversals

Key Characteristics

- Uses recursion or stack
- Goes deep before wide
- Can track path information

Common Problems

- Path Sum

- All Paths for a Sum
- Sum of Path Numbers
- Maximum Depth of Binary Tree

Basic Template

python

```python
def dfs_recursive(root):
    if not root:
        return

    # Pre-order: process current node
    process(root.val)

    # Recurse on children
    dfs_recursive(root.left)
    dfs_recursive(root.right)

    # Post-order: process after children
```

9. Two Heaps Pattern

When to Use

- Finding median in a stream
- Scheduling problems
- Need to track both minimum and maximum

Key Characteristics

- Uses min heap and max heap together
- Balances elements between heaps
- Provides O(1) median access

Common Problems

- Find Median from Data Stream
- Sliding Window Median
- IPO (Maximize Capital)

Basic Template

```python
import heapq

class MedianFinder:
    def __init__(self):
        self.max_heap = []  # First half (smaller numbers)
        self.min_heap = []  # Second half (larger numbers)

    def add_num(self, num):
        heapq.heappush(self.max_heap, -num)

        # Balance heaps
        heapq.heappush(self.min_heap, -heapq.heappop(self.max_heap))

        # Ensure max_heap has equal or one more element
        if len(self.min_heap) > len(self.max_heap):
            heapq.heappush(self.max_heap, -heapq.heappop(self.min_heap))
```

10. Subsets Pattern (Backtracking)

When to Use

- Finding all combinations/permutations
- Generating all possible subsets
- Problems with "find all" requirements

Key Characteristics

- Uses backtracking or BFS approach
- Builds solutions incrementally
- Explores all possibilities

Common Problems

- Subsets
- Subsets II (with duplicates)
- Permutations
- Letter Combinations of a Phone Number

Basic Template

python

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

    def backtrack(start, path):
        result.append(path[:])

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

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

11. Modified Binary Search Pattern

When to Use

- Searching in sorted/rotated arrays
- Finding boundaries
- Optimization problems with monotonic property

Key Characteristics

- Modifies standard binary search
- Works on sorted or partially sorted data
- O(log n) time complexity

Common Problems

- Search in Rotated Sorted Array
- Find Minimum in Rotated Sorted Array
- Search a 2D Matrix
- Find Peak Element

Basic Template

python

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

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

        if arr[mid] == target:
            return mid

        # Modify condition based on problem
        if condition:
            left = mid + 1
        else:
            right = mid - 1

    return -1
```

12. Top K Elements Pattern

When to Use

- Finding k largest/smallest elements
- Finding k most frequent elements
- Problems involving "top k" or "kth"

Key Characteristics

- Uses heap data structure
- Maintains heap of size k
- O(n log k) complexity

Common Problems

- Kth Largest Element
- Top K Frequent Elements
- K Closest Points to Origin
- Sort Characters by Frequency

Basic Template

```python
import heapq

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

    for num in nums:
        heapq.heappush(min_heap, num)
        if len(min_heap) > k:
            heapq.heappop(min_heap)

    return min_heap
```

13. K-way Merge Pattern

When to Use

- Merging k sorted arrays/lists
- Finding smallest range covering k lists
- Problems with multiple sorted inputs

Key Characteristics

- Uses min heap to track smallest elements
- Processes one element at a time
- Maintains pointers for each list

Common Problems

- Merge k Sorted Lists
- Kth Smallest Element in Sorted Matrix
- Smallest Range Covering Elements from K Lists

Basic Template

```python
import heapq

def merge_k_lists(lists):
    min_heap = []

    # Add first element from each list
    for i, lst in enumerate(lists):
        if lst:
            heapq.heappush(min_heap, (lst[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]):
            heapq.heappush(min_heap,
                          (lists[list_idx][elem_idx + 1], list_idx, elem_idx + 1))

    return result
```

14. Dynamic Programming Patterns

Common DP Patterns

1. 0/1 Knapsack

- Each item can be included or excluded
- Examples: Subset Sum, Equal Partition

2. Unbounded Knapsack

- Items can be used multiple times
- Examples: Coin Change, Rod Cutting

3. Fibonacci Numbers

- Current state depends on previous states
- Examples: House Robber, Climbing Stairs

4. Palindromic Subsequence

- Finding palindromic patterns

- Examples: Longest Palindromic Subsequence

5. Longest Common Substring/Subsequence

- Comparing two sequences
- Examples: Edit Distance, LCS

Basic Template

python

```python
# Bottom-up DP
def dp_solution(arr):
    n = len(arr)
    dp = [0] * (n + 1)

    # Base case
    dp[0] = base_value

    # Fill dp table
    for i in range(1, n + 1):
        dp[i] = max(include_current, exclude_current)

    return dp[n]

# Top-down with memoization
def dp_memoization(arr):
    memo = {}

    def helper(index, state):
        if (index, state) in memo:
            return memo[(index, state)]

        # Base case
        if index >= len(arr):
            return 0

        # Recursive case
        result = max(include, exclude)
        memo[(index, state)] = result
        return result

    return helper(0, initial_state)
```

15. Graph Patterns

Common Graph Algorithms

1. BFS for Shortest Path

- Unweighted graphs
- Level-by-level exploration

2. DFS for Path Finding

- Detecting cycles
- Topological sort
- Connected components

3. Union Find (Disjoint Set)

- Detecting cycles in undirected graphs
- Finding connected components
- Kruskal's algorithm

Basic Templates

python

```python
# BFS
from collections import deque

def bfs(graph, start):
    visited = set([start])
    queue = deque([start])

    while queue:
        node = queue.popleft()

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

# DFS
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# Union Find
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])
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px == py:
            return False

        if self.rank[px] < self.rank[py]:
            px, py = py, px

        self.parent[py] = px
```

```python
        if self.rank[px] == self.rank[py]:
            self.rank[px] += 1

        return True
```

Tips for Identifying Patterns

1. **Read the problem carefully**: Look for keywords like "subarray", "subsequence", "pairs", "sorted", "cycle", etc.
2. **Analyze constraints**:
    - Array of size n with values 1 to n → Cyclic Sort
    - Need all combinations → Subsets/Backtracking
    - K smallest/largest → Heap patterns
3. **Consider data structures**:
    - Need fast lookups → Hash Map
    - Need ordered data → Heap or TreeMap
    - Need LIFO → Stack
    - Need FIFO → Queue
4. **Time/Space requirements**:
    - O(n log n) allowed → Can sort first
    - O(1) space → In-place algorithms
    - O(n) time for sorted array → Two Pointers
5. **Practice recognition**: The more problems you solve, the better you'll become at quickly identifying which pattern to apply.