Data Structures & Algorithms - Essential Patterns Guide

Array & String Patterns

1. Two Pointers

Use Cases: Pair problems, palindromes, sorted array operations **Examples**: Two Sum in sorted array, Remove duplicates, Valid palindrome

```python
# Template
left, right = 0, len(arr) - 1
while left < right:
    # Process based on condition
    if condition:
        left += 1
    else:
        right -= 1
```

2. Sliding Window

Use Cases: Subarray/substring problems, fixed/variable window size **Examples**: Maximum sum subarray, Longest substring without repeating characters

```python
# Fixed window
def sliding_window_fixed(arr, k):
    window_sum = sum(arr[:k])
    max_sum = window_sum
    for i in range(k, len(arr)):
        window_sum = window_sum - arr[i-k] + arr[i]
        max_sum = max(max_sum, window_sum)
    return max_sum

# Variable window
def sliding_window_variable(arr, target):
    left = 0
    for right in range(len(arr)):
        # Add arr[right] to window
        while window_condition_violated:
            # Remove arr[left] from window
            left += 1
        # Update result
```

3. Fast & Slow Pointers

Use Cases: Cycle detection, finding middle element, linked list problems **Examples**: Detect cycle in linked list, Find middle of linked list

```python
# Template
slow = fast = head
while fast and fast.next:
    slow = slow.next
    fast = fast.next.next
    if slow == fast:  # Cycle detected
        break
```

4. Prefix Sum

Use Cases: Range sum queries, subarray sum problems **Examples**: Subarray sum equals K, Range sum query

```python
# Template
def prefix_sum(arr):
    prefix = [0] * (len(arr) + 1)
    for i in range(len(arr)):
        prefix[i + 1] = prefix[i] + arr[i]
    return prefix
```

Tree Patterns

5. Tree Traversal

Use Cases: Tree processing, path problems, tree construction **Examples**: Binary tree paths, Validate BST, Tree serialization

DFS (Depth-First Search)

```python
def dfs(node):
    if not node:
        return
    # Process current node
    dfs(node.left)
    dfs(node.right)
```

BFS (Breadth-First Search)

```python
from collections import deque
def bfs(root):
    queue = deque([root])
    while queue:
        node = queue.popleft()
        # Process node
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

6. Binary Search on Trees

Use Cases: Search in BST, closest element, range queries **Examples**: Search in BST, Kth smallest in BST

```python
def search_bst(root, target):
    while root:
        if root.val == target:
            return root
        elif root.val < target:
            root = root.right
        else:
            root = root.left
    return None
```

Graph Patterns

7. Graph Traversal (DFS/BFS)

Use Cases: Connected components, shortest path, cycle detection **Examples**: Number of islands, Clone graph, Course schedule

DFS Template

python

```python
def dfs(graph, node, visited):
    if node in visited:
        return
    visited.add(node)
    for neighbor in graph[node]:
        dfs(graph, neighbor, visited)
```

BFS Template

python

```python
from collections import deque
def bfs(graph, start):
    queue = deque([start])
    visited = {start}
    while queue:
        node = queue.popleft()
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

8. Topological Sort

Use Cases: Task scheduling, dependency resolution, course prerequisites **Examples**: Course schedule, Alien dictionary

python

```python
def topological_sort(graph):
    in_degree = {node: 0 for node in graph}
    for node in graph:
        for neighbor in graph[node]:
            in_degree[neighbor] += 1

    queue = deque([node for node in in_degree if in_degree[node] == 0])
    result = []

    while queue:
        node = queue.popleft()
        result.append(node)
        for neighbor in graph[node]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    return result if len(result) == len(graph) else []
```

Search Patterns

9. Binary Search

Use Cases: Search in sorted array, find boundaries, optimization problems **Examples**: Find target, Search in rotated array, Find peak element

python

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

10. Binary Search on Answer

Use Cases: Optimization problems, find minimum/maximum value **Examples**: Minimum eating speed, Split array largest sum

python

```python
def binary_search_answer(arr, check_function):
    left, right = min_possible, max_possible
    while left < right:
        mid = (left + right) // 2
        if check_function(arr, mid):
            right = mid
        else:
            left = mid + 1
    return left
```

Dynamic Programming Patterns

11. Linear DP

Use Cases: Sequential decision problems, optimization **Examples**: Fibonacci, House robber, Climbing stairs

python

```python
def linear_dp(arr):
    dp = [0] * len(arr)
    dp[0] = arr[0]
    for i in range(1, len(arr)):
        dp[i] = max(dp[i-1], arr[i])  # Example logic
    return dp[-1]
```

12. 2D DP

Use Cases: Grid problems, string matching, path counting **Examples**: Unique paths, Edit distance, Longest common subsequence

```python
def grid_dp(grid):
    m, n = len(grid), len(grid[0])
    dp = [[0] * n for _ in range(m)]
    dp[0][0] = grid[0][0]

    for i in range(m):
        for j in range(n):
            if i > 0:
                dp[i][j] = max(dp[i][j], dp[i-1][j] + grid[i][j])
            if j > 0:
                dp[i][j] = max(dp[i][j], dp[i][j-1] + grid[i][j])

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

13. Knapsack Pattern

Use Cases: Subset selection, optimization with constraints **Examples**: 0/1 Knapsack, Subset sum, Partition equal subset sum

```python
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i-1] <= w:
                dp[i][w] = max(dp[i-1][w], dp[i-1][w-weights[i-1]] + values[i-1])
            else:
                dp[i][w] = dp[i-1][w]

    return dp[n][capacity]
```

Backtracking Patterns

14. Backtracking

Use Cases: Generate all combinations, permutations, solutions **Examples**: N-Queens, Sudoku solver, Generate parentheses

```python
def backtrack(path, choices):
    if is_valid_solution(path):
        result.append(path[:])
        return

    for choice in choices:
        if is_valid_choice(path, choice):
            path.append(choice)
            backtrack(path, get_next_choices(choices, choice))
            path.pop()
```

Sorting & Searching Patterns

15. Merge Sort Pattern

Use Cases: Divide and conquer, counting inversions, external sorting **Examples**: Merge sort, Count inversions, Merge k sorted lists

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)
```

16. Quick Select Pattern

Use Cases: Finding kth element, median finding **Examples**: Kth largest element, Top K frequent elements

```python
def quick_select(arr, k):
    if len(arr) == 1:
        return arr[0]

    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    right = [x for x in arr if x > pivot]

    if k <= len(left):
        return quick_select(left, k)
    elif k > len(arr) - len(right):
        return quick_select(right, k - (len(arr) - len(right)))
    else:
        return pivot
```

Heap Patterns

17. Heap (Priority Queue)

Use Cases: Top K problems, merge operations, scheduling **Examples**: Kth largest, Merge k sorted lists, Task scheduler

```python
import heapq

def top_k_elements(arr, k):
    heap = []
    for num in arr:
        heapq.heappush(heap, num)
        if len(heap) > k:
            heapq.heappop(heap)
    return heap
```

Advanced Patterns

18. Union-Find (Disjoint Set)

Use Cases: Connected components, cycle detection, dynamic connectivity **Examples**: Number of islands, Redundant connection

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])
        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
        if self.rank[px] == self.rank[py]:
            self.rank[px] += 1
        return True
```

19. Trie (Prefix Tree)

Use Cases: String prefix problems, autocomplete, word search **Examples**: Implement trie, Word search II, Autocomplete

python

```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
```

20. Segment Tree

Use Cases: Range queries, range updates, interval problems **Examples**: Range sum query, Range minimum query

python

```python
class SegmentTree:
    def __init__(self, arr):
        self.n = len(arr)
        self.tree = [0] * (4 * self.n)
        self.build(arr, 0, 0, self.n - 1)

    def build(self, arr, node, start, end):
        if start == end:
            self.tree[node] = arr[start]
        else:
            mid = (start + end) // 2
            self.build(arr, 2*node+1, start, mid)
            self.build(arr, 2*node+2, mid+1, end)
            self.tree[node] = self.tree[2*node+1] + self.tree[2*node+2]
```

Pattern Selection Guide

Problem Type → Pattern Mapping

Problem Type	Recommended Patterns
Array/String with conditions	Two Pointers, Sliding Window
Subarray/Substring problems	Sliding Window, Prefix Sum
Tree problems	DFS, BFS, Binary Search on Trees
Graph problems	DFS, BFS, Topological Sort, Union-Find
Search problems	Binary Search, DFS, BFS
Optimization problems	Dynamic Programming, Greedy
Combinatorial problems	Backtracking, Dynamic Programming
Top K problems	Heap, Quick Select
Range query problems	Segment Tree, Binary Indexed Tree
String matching	KMP, Trie, Dynamic Programming

Time Complexity Quick Reference

Pattern	Time Complexity	Space Complexity
Two Pointers	O(n)	O(1)
Sliding Window	O(n)	O(1)
Binary Search	O(log n)	O(1)
DFS/BFS	O(V + E)	O(V)
Dynamic Programming	$O(n^2)$ typical	O(n) to $O(n^2)$
Backtracking	$O(2^n)$ typical	O(n)
Heap operations	O(log n)	O(n)
Union-Find	O(α(n))	O(n)

Key Tips for Pattern Recognition

1. **Array/String + Two elements** → Two Pointers
2. **Subarray/Substring + condition** → Sliding Window
3. **Tree + path/traversal** → DFS/BFS
4. **Graph + connectivity** → Union-Find
5. **Sorted array + search** → Binary Search
6. **Optimization + choices** → Dynamic Programming
7. **Generate all solutions** → Backtracking
8. **Top K + comparison** → Heap
9. **Prefix/Range queries** → Prefix Sum/Segment Tree
10. **String patterns** → Trie/KMP

Remember: Most coding problems can be solved by combining 2-3 of these patterns. Practice recognizing which pattern(s) fit the problem constraints and requirements.