FAANG Coding Interview Patterns & Templates

Python Reference Guide

Master These Patterns for Interview Success

GOAL: Practice these templates until they become muscle memory
COVERAGE: 12 Essential Patterns + Recognition Guide
TIMELINE: 4-Week Intensive Study Plan
TARGET: FAANG/MAANG Interview Readiness

Based on 2024 FAANG Interview Analysis

September 28, 2025

Contents

1 Quick Pattern Recognition Guide

Pattern	Key Indicators	When to Use
Two Pointers	Sorted array, pairs, palindromes	Finding pairs with target sum, removing duplicates
Sliding Window	Subarray/substring, contiguous	Max/min subarray, longest substring problems
DFS	Tree/graph traversal, explore all paths	Path finding, connectivity, topological sort
BFS	Shortest path, level-order	Minimum steps, level traversal
Binary Search	Sorted data, search space	Finding element, first/last occurrence
Dynamic Programming	Optimal substructure, overlapping subproblems	Optimization, counting, decision problems
Backtracking	Generate all combinations/permutations	N-Queens, Sudoku, subset generation
Heap	Top K, merge K, priority	Finding extremes, scheduling
Union-Find	Connected components, cycles	Graph connectivity, MST
Trie	Prefix matching, word games	Autocomplete, word search

2 Core Algorithm Patterns

2.1 1. Depth-First Search (DFS)

DFS - Tree Traversal

Use Case: Tree/graph traversal, path finding, cycle detection
Time: O(V + E) for graphs, O(n) for trees
Space: O(h) where h is height/depth

2.1.1 Recursive DFS Template

```python
def dfs_recursive(root):
    if not root:
        return  # Base case

    # Process current node
    print(root.val)

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

# With result collection
def dfs_collect_paths(root, path=[], all_paths=[]):
    if not root:
        return

    path.append(root.val)

    # Leaf node - save path
    if not root.left and not root.right:
        all_paths.append(path[:])  # Copy path

    dfs_collect_paths(root.left, path, all_paths)
    dfs_collect_paths(root.right, path, all_paths)

    path.pop()  # Backtrack
    return all_paths
```

2.1.2 Iterative DFS Template

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

    stack = [root]
    while stack:
        node = stack.pop()
        print(node.val)  # Process node

        # Add children (right first for left-to-right order)
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)

# Graph DFS with visited set
def dfs_graph(graph, start):
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            print(node)  # Process node

            # Add neighbors
            for neighbor in graph[node]:
                if neighbor not in visited:
                    stack.append(neighbor)
```

2.2 2. Breadth-First Search (BFS)

BFS - Level-by-Level Traversal

Use Case: Shortest path, level-order traversal, minimum steps
Time: O(V + E) for graphs, O(n) for trees
Space: O(w) where w is maximum width

2.2.1 Standard BFS Template

```python
from collections import deque

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

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

    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)

        result.append(level)
    return result
```

```python
# Simple BFS without levels
def bfs_simple(root):
    if not root:
        return

    queue = deque([root])
    while queue:
        node = queue.popleft()
        print(node.val)  # Process node

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

2.2.2 BFS for Shortest Path

```python
def bfs_shortest_path(graph, start, target):
    queue = deque([(start, 0)])  # (node, distance)
    visited = set([start])

    while queue:
        node, dist = queue.popleft()

        if node == target:
            return dist

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, dist + 1))

    return -1  # Target not reachable
```

2.3 3. Two Pointers

Two Pointers - Efficient Array/String Processing

Use Case: Sorted arrays, palindromes, pairs with target sum
Time: O(n) typically
Space: O(1) space optimization

2.3.1 Opposite Ends Template

```python
def two_sum_sorted(nums, target):
    left, right = 0, len(nums) - 1

    while left < right:
        current_sum = nums[left] + nums[right]

        if current_sum == target:
            return [left, right]
        elif current_sum < target:
            left += 1
        else:
            right -= 1

    return []

def is_palindrome(s):
    left, right = 0, len(s) - 1

    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1
```

```

    return True

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

    while left < right:
        arr[left], arr[right] = arr[right], arr[left]
        left += 1
        right -= 1
```

2.3.2 Fast and Slow Pointers

```
def has_cycle(head):
    if not head or not head.next:
        return False

    slow = fast = head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

        if slow == fast:
            return True

    return False

def find_middle(head):
    slow = fast = head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    return slow

def remove_nth_from_end(head, n):
    dummy = ListNode(0)
    dummy.next = head
    slow = fast = dummy

    # Move fast n+1 steps ahead
    for _ in range(n + 1):
        fast = fast.next

    # Move both until fast reaches end
    while fast:
        slow = slow.next
        fast = fast.next

    # Remove nth node
    slow.next = slow.next.next
    return dummy.next
```

2.4 4. Sliding Window

Sliding Window - Contiguous Subarray/Substring

Use Case: Maximum/minimum subarray, longest substring
Time: O(n) single pass
Space: O(k) for tracking window contents

2.4.1 Fixed Size Window

```
def max_sum_subarray(nums, k):
    if len(nums) < k:
```

```python
        return 0

    # Calculate first window
    window_sum = sum(nums[:k])
    max_sum = window_sum

    # Slide window
    for i in range(k, len(nums)):
        window_sum = window_sum - nums[i - k] + nums[i]
        max_sum = max(max_sum, window_sum)

    return max_sum

def find_averages(nums, k):
    result = []
    window_sum = 0

    for i in range(len(nums)):
        window_sum += nums[i]

        if i >= k - 1:
            result.append(window_sum / k)
            window_sum -= nums[i - k + 1]

    return result
```

2.4.2 Variable Size Window

```python
def longest_substring_k_distinct(s, k):
    if k == 0:
        return 0

    char_count = {}
    left = 0
    max_length = 0

    for right in range(len(s)):
        # Expand window
        char_count[s[right]] = char_count.get(s[right], 0) + 1

        # Contract window if needed
        while len(char_count) > k:
            char_count[s[left]] -= 1
            if char_count[s[left]] == 0:
                del char_count[s[left]]
            left += 1

        max_length = max(max_length, right - left + 1)

    return max_length

def min_window_substring(s, t):
    if not s or not t:
        return ""

    t_count = {}
    for char in t:
        t_count[char] = t_count.get(char, 0) + 1

    left = 0
    min_len = float('inf')
    min_start = 0
    matched = 0
    window_count = {}

    for right in range(len(s)):
        char = s[right]
        window_count[char] = window_count.get(char, 0) + 1

        if char in t_count and window_count[char] == t_count[char]:
            matched += 1

```

```python
        while matched == len(t_count):
            if right - left + 1 < min_len:
                min_len = right - left + 1
                min_start = left

            left_char = s[left]
            window_count[left_char] -= 1
            if left_char in t_count and window_count[left_char] < t_count[left_char]:
                matched -= 1
            left += 1

    return s[min_start:min_start + min_len] if min_len != float('inf') else ""
```

2.5 5. Binary Search

Binary Search - Efficient Search in Sorted Space

Use Case: Finding element, first/last occurrence, search space
Time: O(log n)
Space: O(1) iterative, O(log n) recursive

2.5.1 Standard Binary Search

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

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

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

    return -1

def binary_search_recursive(nums, target, left=0, right=None):
    if right is None:
        right = len(nums) - 1

    if left > right:
        return -1

    mid = left + (right - left) // 2

    if nums[mid] == target:
        return mid
    elif nums[mid] < target:
        return binary_search_recursive(nums, target, mid + 1, right)
    else:
        return binary_search_recursive(nums, target, left, mid - 1)
```

2.5.2 Find First and Last Occurrence

```python
def find_first_occurrence(nums, target):
    left, right = 0, len(nums) - 1
    result = -1

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

        if nums[mid] == target:
            result = mid
            right = mid - 1  # Continue searching left
        elif nums[mid] < target:
```

```python
            left = mid + 1
        else:
            right = mid - 1

    return result

def find_last_occurrence(nums, target):
    left, right = 0, len(nums) - 1
    result = -1

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

        if nums[mid] == target:
            result = mid
            left = mid + 1  # Continue searching right
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return result

def search_range(nums, target):
    return [find_first_occurrence(nums, target),
            find_last_occurrence(nums, target)]
```

2.5.3 Binary Search on Answer

```python
def find_peak_element(nums):
    left, right = 0, len(nums) - 1

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

        if nums[mid] > nums[mid + 1]:
            right = mid
        else:
            left = mid + 1

    return left

def search_rotated_sorted_array(nums, target):
    left, right = 0, len(nums) - 1

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

        if nums[mid] == target:
            return mid

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

    return -1
```

2.6 6. Dynamic Programming

Dynamic Programming - Optimal Substructure

Use Case: Optimization, counting, decision problems
Time: Varies (often $O(n^2)$ or $O(n*m)$)
Space: $O(n)$ to $O(n*m)$ depending on dimensions

2.6.1 1D DP Template

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

    # Bottom-up approach
    dp = [0] * (n + 1)
    dp[1] = 1

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

    return dp[n]

# Space optimized
def fibonacci_optimized(n):
    if n <= 1:
        return n

    prev2, prev1 = 0, 1

    for i in range(2, n + 1):
        current = prev1 + prev2
        prev2, prev1 = prev1, current

    return prev1

def climb_stairs(n):
    if n <= 2:
        return n

    dp = [0] * (n + 1)
    dp[1], dp[2] = 1, 2

    for i in range(3, n + 1):
        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]

def house_robber(nums):
    if not nums:
        return 0
    if len(nums) == 1:
        return nums[0]

    dp = [0] * len(nums)
    dp[0] = nums[0]
    dp[1] = max(nums[0], nums[1])

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

    return dp[-1]
```

2.6.2 2D DP Template

```python
def unique_paths(m, n):
    # Create DP table
    dp = [[1 for _ in range(n)] for _ in range(m)]
```

```python

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

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

def longest_common_subsequence(text1, text2):
    m, n = len(text1), len(text2)
    dp = [[0 for _ in range(n + 1)] for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if text1[i-1] == text2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[m][n]

def min_path_sum(grid):
    m, n = len(grid), len(grid[0])

    # Initialize first row and column
    for i in range(1, m):
        grid[i][0] += grid[i-1][0]

    for j in range(1, n):
        grid[0][j] += grid[0][j-1]

    # Fill the DP table
    for i in range(1, m):
        for j in range(1, n):
            grid[i][j] += min(grid[i-1][j], grid[i][j-1])

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

2.7 7. Backtracking

Backtracking - Explore All Possibilities

Use Case: Permutations, combinations, N-Queens, Sudoku
Time: Exponential O(2^n)$orO(n!)$
Space:$O(depth)forrecursionstack$

2.7.1 General Backtracking Template

```python
def backtrack_template(candidates, target, path=[], result=[]):
    # Base case - solution found
    if is_valid_solution(path, target):
        result.append(path[:])  # Make a copy
        return

    # Explore all possibilities
    for i, candidate in enumerate(candidates):
        # Skip invalid candidates
        if not is_valid_candidate(candidate, path):
            continue

        # Make choice
        path.append(candidate)

        # Recurse with remaining candidates
        backtrack_template(candidates[i+1:], target, path, result)

        # Undo choice (backtrack)
        path.pop()

    return result
```

```python

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

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

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

2.7.2 Combinations and Subsets

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

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

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

def combination_sum(candidates, target):
    def backtrack(start, path, remaining):
        if remaining == 0:
            result.append(path[:])
            return

        for i in range(start, len(candidates)):
            if candidates[i] > remaining:
                break

            path.append(candidates[i])
            # Can reuse same element
            backtrack(i, path, remaining - candidates[i])
            path.pop()

    result = []
    candidates.sort()
    backtrack(0, [], target)
    return result

def letter_combinations(digits):
    if not digits:
        return []

    phone = {
        '2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl',
        '6': 'mno', '7': 'pqrs', '8': 'tuv', '9': 'wxyz'
    }

    def backtrack(index, path):
        if index == len(digits):
            result.append("".join(path))
            return

        for letter in phone[digits[index]]:
            path.append(letter)
            backtrack(index + 1, path)
            path.pop()
```

```python

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

2.8 8. Heap Operations

Heap - Priority Queue for Top K Problems

Use Case: Top K elements, merge K sorted, scheduling
Time: O(log n) insert/delete, O(1) peek
Space: O(n) for heap storage

2.8.1 Min/Max Heap Templates

```python
import heapq

def find_kth_largest(nums, k):
    # Use min heap of size k
    min_heap = []

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

    return min_heap[0]

def find_k_largest_elements(nums, k):
    # Min heap approach
    min_heap = []

    for num in nums:
        if len(min_heap) < k:
            heapq.heappush(min_heap, num)
        elif num > min_heap[0]:
            heapq.heapreplace(min_heap, num)

    return sorted(min_heap, reverse=True)

# For max heap, negate values
def max_heap_operations():
    max_heap = []

    # Insert (negate for max heap)
    heapq.heappush(max_heap, -value)

    # Extract max (negate result)
    max_val = -heapq.heappop(max_heap)

    # Peek max
    max_val = -max_heap[0]
```

2.8.2 Advanced Heap Applications

```python
def merge_k_sorted_lists(lists):
    import heapq

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

    dummy = ListNode(0)
    current = dummy

```

```python
    while min_heap:
        val, list_idx, node = heapq.heappop(min_heap)
        current.next = node
        current = current.next

        # Add next element from same list
        if node.next:
            heapq.heappush(min_heap, (node.next.val, list_idx, node.next))

    return dummy.next

def top_k_frequent_elements(nums, k):
    from collections import Counter
    import heapq

    count = Counter(nums)

    # Min heap of size k
    min_heap = []
    for num, freq in count.items():
        heapq.heappush(min_heap, (freq, num))
        if len(min_heap) > k:
            heapq.heappop(min_heap)

    return [num for freq, num in min_heap]

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

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

        # Balance: move largest from small to large
        heapq.heappush(self.large, -heapq.heappop(self.small))

        # Ensure small has more or equal elements
        if len(self.large) > len(self.small):
            heapq.heappush(self.small, -heapq.heappop(self.large))

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

2.9 9. Union-Find (Disjoint Set)

Union-Find - Connected Components

Use Case: Graph connectivity, cycle detection, MST
Time: O((n)) amortized per operation
Space: O(n) for parent and rank arrays

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

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

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)
```

```python

        if root_x == root_y:
            return False  # Already connected

        # Union by rank
        if self.rank[root_x] < self.rank[root_y]:
            self.parent[root_x] = root_y
        elif self.rank[root_x] > self.rank[root_y]:
            self.parent[root_y] = root_x
        else:
            self.parent[root_y] = root_x
            self.rank[root_x] += 1

        self.components -= 1
        return True

    def connected(self, x, y):
        return self.find(x) == self.find(y)

    def count_components(self):
        return self.components

# Applications
def number_of_islands(grid):
    if not grid:
        return 0

    m, n = len(grid), len(grid[0])
    uf = UnionFind(m * n)
    islands = 0

    for i in range(m):
        for j in range(n):
            if grid[i][j] == '1':
                islands += 1
                # Check 4 directions
                for di, dj in [(0,1), (1,0), (0,-1), (-1,0)]:
                    ni, nj = i + di, j + dj
                    if (0 <= ni < m and 0 <= nj < n and
                        grid[ni][nj] == '1'):
                        if uf.union(i*n + j, ni*n + nj):
                            islands -= 1

    return islands

def redundant_connection(edges):
    uf = UnionFind(len(edges) + 1)

    for u, v in edges:
        if not uf.union(u, v):
            return [u, v]  # This edge creates a cycle

    return []
```

2.10 10. Trie (Prefix Tree)

Trie - Efficient String Storage and Search

Use Case: Autocomplete, word search, prefix matching
Time: O(m) where m is string length
Space: O(ALPHABET_SIZE * N * M) worst case

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

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

```python

    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_of_word = 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_of_word

    def starts_with(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True

    def find_words_with_prefix(self, prefix):
        # Find the prefix node
        node = self.root
        for char in prefix:
            if char not in node.children:
                return []
            node = node.children[char]

        # DFS to find all words
        words = []
        self._dfs(node, prefix, words)
        return words

    def _dfs(self, node, path, words):
        if node.is_end_of_word:
            words.append(path)

        for char, child_node in node.children.items():
            self._dfs(child_node, path + char, words)

# Word Search II using Trie
def find_words_in_board(board, words):
    # Build trie
    trie = Trie()
    for word in words:
        trie.insert(word)

    result = set()
    m, n = len(board), len(board[0])

    def dfs(i, j, node, path):
        if node.is_end_of_word:
            result.add(path)

        if (i < 0 or i >= m or j < 0 or j >= n or
            board[i][j] not in node.children):
            return

        char = board[i][j]
        board[i][j] = '#'  # Mark visited

        # Explore 4 directions
        for di, dj in [(0,1), (1,0), (0,-1), (-1,0)]:
            dfs(i + di, j + dj, node.children[char], path + char)

        board[i][j] = char  # Restore

    # Try starting from each cell
```

```python
    for i in range(m):
        for j in range(n):
            if board[i][j] in trie.root.children:
                dfs(i, j, trie.root, "")

    return list(result)
```

2.11 11. Topological Sort

Topological Sort - Ordering with Dependencies

Use Case: Course scheduling, dependency resolution
Time: O(V + E)
Space: O(V) for in-degree array and queue

```python
from collections import defaultdict, deque

def topological_sort_kahn(num_courses, prerequisites):
    # Build graph and in-degree array
    graph = defaultdict(list)
    in_degree = [0] * num_courses

    for course, prereq in prerequisites:
        graph[prereq].append(course)
        in_degree[course] += 1

    # Initialize queue with nodes having in-degree 0
    queue = deque([i for i in range(num_courses) if in_degree[i] == 0])
    result = []

    while queue:
        node = queue.popleft()
        result.append(node)

        # Reduce in-degree of neighbors
        for neighbor in graph[node]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    # Check if topological sort is possible
    return result if len(result) == num_courses else []

def can_finish_courses(num_courses, prerequisites):
    topo_order = topological_sort_kahn(num_courses, prerequisites)
    return len(topo_order) == num_courses

# DFS-based topological sort
def topological_sort_dfs(graph):
    visited = set()
    rec_stack = set()
    result = []

    def dfs(node):
        if node in rec_stack:
            return False  # Cycle detected
        if node in visited:
            return True

        visited.add(node)
        rec_stack.add(node)

        for neighbor in graph.get(node, []):
            if not dfs(neighbor):
                return False

        rec_stack.remove(node)
        result.append(node)  # Add to result in post-order
        return True

```

```python
    for node in graph:
        if node not in visited:
            if not dfs(node):
                return []  # Cycle detected

    return result[::-1]  # Reverse for correct order
```

2.12 12. Special Algorithms

2.12.1 Kadane's Algorithm - Maximum Subarray

```python
def max_subarray_sum(nums):
    if not nums:
        return 0

    max_sum = current_sum = nums[0]

    for i in range(1, len(nums)):
        # Either extend existing subarray or start new one
        current_sum = max(nums[i], current_sum + nums[i])
        max_sum = max(max_sum, current_sum)

    return max_sum

def max_subarray_with_indices(nums):
    max_sum = current_sum = nums[0]
    start = end = temp_start = 0

    for i in range(1, len(nums)):
        if current_sum < 0:
            current_sum = nums[i]
            temp_start = i
        else:
            current_sum += nums[i]

        if current_sum > max_sum:
            max_sum = current_sum
            start = temp_start
            end = i

    return max_sum, start, end
```

2.12.2 Prefix Sum

```python
class PrefixSum:
    def __init__(self, nums):
        self.prefix = [0]
        for num in nums:
            self.prefix.append(self.prefix[-1] + num)

    def range_sum(self, i, j):
        # Sum from index i to j (inclusive)
        return self.prefix[j + 1] - self.prefix[i]

def subarray_sum_equals_k(nums, k):
    count = 0
    prefix_sum = 0
    sum_count = {0: 1}  # prefix_sum -> frequency

    for num in nums:
        prefix_sum += num
        if prefix_sum - k in sum_count:
            count += sum_count[prefix_sum - k]
        sum_count[prefix_sum] = sum_count.get(prefix_sum, 0) + 1

    return count
```

2.12.3 Monotonic Stack

```python
def next_greater_element(nums):
    result = [-1] * len(nums)
    stack = []  # Store indices

    for i in range(len(nums)):
        while stack and nums[i] > nums[stack[-1]]:
            index = stack.pop()
            result[index] = nums[i]
        stack.append(i)

    return result

def daily_temperatures(temperatures):
    result = [0] * len(temperatures)
    stack = []

    for i, temp in enumerate(temperatures):
        while stack and temp > temperatures[stack[-1]]:
            prev_index = stack.pop()
            result[prev_index] = i - prev_index
        stack.append(i)

    return result

def largest_rectangle_in_histogram(heights):
    stack = []
    max_area = 0

    for i, h in enumerate(heights + [0]):  # Add sentinel
        while stack and h < heights[stack[-1]]:
            height = heights[stack.pop()]
            width = i if not stack else i - stack[-1] - 1
            max_area = max(max_area, height * width)
        stack.append(i)

    return max_area
```

2.12.4 Cyclic Sort

```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
    return nums

def find_missing_number(nums):
    # Array contains n numbers in range [0, n]
    i = 0
    n = len(nums)

    while i < n:
        if nums[i] < n and nums[i] != nums[nums[i]]:
            nums[nums[i]], nums[i] = nums[i], nums[nums[i]]
        else:
            i += 1

    # Find the missing number
    for i in range(n):
        if nums[i] != i:
            return i

    return n

def find_all_duplicates(nums):
    duplicates = []
```

```python

    for i in range(len(nums)):
        # Use array indices to mark presence
        num = abs(nums[i])
        if nums[num - 1] < 0:
            duplicates.append(num)
        else:
            nums[num - 1] *= -1

    return duplicates
```

3 Pattern Recognition Guide

Complexity Analysis

Time Complexity Quick Reference:

- O(1) - Hash table access, array index
- O(log n) - Binary search, heap operations
- O(n) - Single pass through array, BFS/DFS
- O(n log n) - Merge sort, heap sort
- O(n^2) - Nested loops, bubble sort
- O($2^n) - Recursivealgorithmswithoutmemoization$

3.1 Decision Tree for Pattern Selection

1. **Array/String Problems:**
    - Sorted array → Binary Search
    - Two elements with condition → Two Pointers
    - Contiguous subarray/substring → Sliding Window
    - All subarrays → Prefix Sum or DP
2. **Tree/Graph Problems:**
    - Path finding → DFS
    - Shortest path → BFS
    - Level-order traversal → BFS
    - Connected components → Union-Find or DFS
3. **Optimization Problems:**
    - Optimal substructure → Dynamic Programming
    - Multiple choices at each step → Backtracking
    - Greedy choice property → Greedy Algorithm
4. **Priority/Ordering Problems:**
    - Top K elements → Heap
    - Streaming data → Heap
    - Dependencies → Topological Sort
5. **String Matching:**
    - Prefix operations → Trie

- Pattern matching → KMP or Rolling Hash
- Anagrams → Hash Map

Pro Tip

Red Flags for Common Patterns:

- "Maximum/Minimum subarray" → Sliding Window or Kadane's
- "All permutations/combinations" → Backtracking
- "Shortest path in unweighted graph" → BFS
- "Detect cycle" → DFS or Union-Find
- "Top K" or "K-th largest" → Heap

4 4-Week Study Schedule

4.1 Week 1: Foundation Patterns

Day 1-2: Two Pointers & Sliding Window

- Practice writing templates from memory
- Solve: Two Sum II, Container With Most Water, Longest Substring Without Repeating Characters

Day 3-4: DFS & BFS

- Master both recursive and iterative approaches
- Solve: Binary Tree Inorder Traversal, Binary Tree Level Order Traversal, Number of Islands

Day 5-7: Binary Search

- Practice standard and modified binary search
- Solve: Search in Rotated Sorted Array, Find First and Last Position, Search Insert Position

4.2 Week 2: Dynamic Programming & Backtracking

Day 8-10: 1D & 2D Dynamic Programming

- Focus on identifying optimal substructure
- Solve: Climbing Stairs, House Robber, Unique Paths, Longest Common Subsequence

Day 11-14: Backtracking

- Master the template and understand when to backtrack
- Solve: Permutations, Combinations, Subsets, N-Queens

4.3 Week 3: Advanced Data Structures

Day 15-17: Heap Operations

- Learn min/max heap operations and applications
- Solve: Kth Largest Element, Top K Frequent Elements, Merge K Sorted Lists

Day 18-19: Union-Find

- Understand path compression and union by rank
- Solve: Number of Islands II, Redundant Connection, Friend Circles

Day 20-21: Trie

- Build trie from scratch and understand applications
- Solve: Implement Trie, Word Search II, Add and Search Word

4.4 Week 4: Special Algorithms & Integration

Day 22-24: Special Algorithms

- Kadane's Algorithm, Prefix Sum, Monotonic Stack
- Solve: Maximum Subarray, Subarray Sum Equals K, Next Greater Element

Day 25-26: Topological Sort

- Both Kahn's algorithm and DFS approach
- Solve: Course Schedule, Course Schedule II, Alien Dictionary

Day 27-28: Integration & Mock Interviews

- Combine multiple patterns in complex problems
- Practice writing templates under time pressure
- Mock interview sessions

5 Daily Practice Routine

5.1 Morning Routine (30 minutes)

1. **Template Writing (10 min):** Write 3 random templates from memory
2. **Pattern Recognition (10 min):** Look at problem titles and identify patterns
3. **Complexity Analysis (10 min):** Review time/space complexity for each pattern

5.2 Evening Session (60-90 minutes)

1. **Problem Solving (45-60 min):**
   - Choose 2-3 problems focusing on current week's patterns
   - Spend 5 minutes identifying the pattern before coding
   - Write solution from scratch using templates
2. **Template Review (15-30 min):**
   - Review any templates you struggled with
   - Write them again from memory
   - Note common mistakes or variations

5.3 Weekly Goals

- **Week 1:** Master 4 foundational patterns
- **Week 2:** Add DP and backtracking to arsenal
- **Week 3:** Comfortable with advanced data structures
- **Week 4:** Integrate patterns and achieve fluency

Pro Tip

Success Metrics:

- Can write any template from memory in under 2 minutes
- Identify correct pattern within 30 seconds of reading problem
- Solve medium problems in 15-20 minutes
- Explain time/space complexity confidently

6 Progress Tracking

6.1 Template Mastery Checklist

Check off when you can write from memory in under 2 minutes:

- ☐ DFS (recursive)
- ☐ DFS (iterative)
- ☐ BFS (standard)
- ☐ BFS (level-order)
- ☐ Two Pointers (opposite ends)
- ☐ Two Pointers (fast/slow)
- ☐ Sliding Window (fixed)
- ☐ Sliding Window (variable)
- ☐ Binary Search (standard)
- ☐ Binary Search (first/last occurrence)
- ☐ 1D DP template
- ☐ 2D DP template
- ☐ Backtracking template
- ☐ Heap operations
- ☐ Union-Find (with optimizations)
- ☐ Trie implementation
- ☐ Topological Sort (Kahn's)
- ☐ Kadane's Algorithm
- ☐ Prefix Sum
- ☐ Monotonic Stack

6.2 Problem Categories to Master

Category	**Easy**	**Medium**	**Hard**
Arrays & Strings	10	15	5
Trees & Graphs	8	12	5
Dynamic Programming	5	10	5
Backtracking	3	8	3
Heaps & Priority Queues	5	8	3
Advanced Data Structures	5	10	5
Total	**36**	**63**	**26**

7 Final Tips for Success

Pro Tip

Interview Day Strategy:

1. Spend 2-3 minutes understanding the problem completely
2. Identify the pattern (this should be automatic by now)
3. Explain your approach before coding
4. Write the template structure first, then fill in details
5. Test with simple examples
6. Analyze time and space complexity

Complexity Analysis

Common Optimizations:

- Two-pass → One-pass with hash map
- Nested loops → Two pointers or hash map
- Recursion → DP with memoization
- Multiple data structures → Single optimized structure
- Extra space → In-place modifications

7.1 Red Flags to Avoid

- Jumping into coding without understanding the problem
- Not identifying the pattern before starting
- Overcomplicating simple problems
- Not testing with edge cases
- Forgetting to analyze complexity

7.2 Last-Minute Review (Day Before Interview)

1. Write all 12 main templates from memory
2. Review the pattern recognition guide
3. Practice one problem from each category
4. Get good sleep and stay confident

Remember: Consistency beats intensity.
Practice these templates daily until they become automatic!