Comprehensive Guide to Coding Interview Patterns in Python

Your Name Here

September 28, 2025

Contents

1 Introduction

This document is a comprehensive guide to the most common coding interview patterns and templates in Python. The goal is to provide a clean, printable reference that you can use to practice until these patterns become muscle memory. Mastering these patterns is crucial for success in technical interviews at top tech companies.

2 Pattern Recognition Guide

Here's a quick guide to help you recognize which pattern to apply based on the problem description:

- **Two Pointers:** Problems involving sorted arrays or linked lists where you need to find a pair of elements that meet a certain condition. Also useful for finding palindromes and detecting cycles.
- **Sliding Window:** Problems that ask for the longest/shortest subarray/substring, or a subarray/substring with a certain property.
- **BFS (Breadth-First Search):** Problems involving finding the shortest path in an unweighted graph or traversing a tree level by level.
- **DFS (Depth-First Search):** Problems involving traversing a graph or tree, checking for connectivity, or finding all paths.
- **Topological Sort:** Problems involving dependencies or ordering of tasks, like course schedules.
- **Binary Search:** Problems on sorted data structures (arrays, matrices) where you need to find a specific element or a range of elements.
- **Dynamic Programming:** Optimization problems (e.g., maximize/minimize something) or counting problems that can be broken down into overlapping subproblems.
- **Backtracking:** Problems that require generating all possible solutions, like permutations, combinations, or solving puzzles like Sudoku.
- **Heap (Priority Queue):** Problems that involve finding the top 'k' elements, medians, or scheduling.
- **Union-Find:** Problems involving connected components in a graph or network, or checking for cycles.
- **Trie:** Problems involving string prefixes, searching for words, or autocomplete features.

3 Core Patterns and Templates

3.1 Depth-First Search (DFS)

3.1.1 Recursive DFS Template

```
from typing import List, Dict, Set

def dfs_recursive(graph: Dict[int, List[int]], start_node: int):
    visited = set()

    def dfs_util(node: int):
        if node in visited:
            return
        visited.add(node)
        print(f"Visiting node: {node}")  # Process node

        for neighbor in graph.get(node, []):
            if neighbor not in visited:
                dfs_util(neighbor)

    dfs_util(start_node)

# Example Usage:
# graph = {0: [1, 2], 1: [2], 2: [0, 3], 3: [3]}
# dfs_recursive(graph, 2)
```

3.1.2 Iterative DFS Template

```
from typing import List, Dict, Set

def dfs_iterative(graph: Dict[int, List[int]], start_node: int):
    visited = set()
    stack = [start_node]

    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            print(f"Visiting node: {node}")  # Process node

            # Add neighbors to the stack in reverse order to visit them in order
            for neighbor in reversed(graph.get(node, [])):
                if neighbor not in visited:
```

```
                    stack.append(neighbor)

# Example Usage:
# graph = {0: [1, 2], 1: [2], 2: [0, 3], 3: [3]}
# dfs_iterative(graph, 2)
```

3.2 Breadth-First Search (BFS)

```
from typing import List, Dict, Set
from collections import deque

def bfs(graph: Dict[int, List[int]], start_node: int):
    visited = set()
    queue = deque([start_node])
    visited.add(start_node)

    while queue:
        node = queue.popleft()
        print(f"Visiting node: {node}")  # Process node

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

# Example Usage:
# graph = {0: [1, 2], 1: [2], 2: [0, 3], 3: [3]}
# bfs(graph, 2)
```

3.3 Topological Sort (Kahn's Algorithm)

```
from typing import List, Dict
from collections import deque

def topological_sort(graph: Dict[int, List[int]], num_nodes: int) -> List[int]:
    in_degree = {i: 0 for i in range(num_nodes)}
    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])
    sorted_order = []
```

```
    while queue:
        node = queue.popleft()
        sorted_order.append(node)

        for neighbor in graph.get(node, []):
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    if len(sorted_order) == num_nodes:
        return sorted_order
    else:
        return []  # Graph has a cycle

# Example Usage:
# num_nodes = 6
# graph = {0: [1, 2], 1: [3], 2: [3, 4], 3: [5], 4: [5], 5: []}
# print(topological_sort(graph, num_nodes))
```

3.4 Two Pointers

3.4.1 Opposite Ends

```
from typing import List

def two_pointers_opposite(arr: List[int], target: int) -> List[int]:
    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
        else:
            right -= 1
    return [-1, -1]

# Example Usage (for sorted array):
# arr = [2, 7, 11, 15]
# target = 9
# print(two_pointers_opposite(arr, target))
```

3.4.2 Fast and Slow Pointers (Cycle Detection)

```
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

def has_cycle(head: ListNode) -> bool:
    if not head:
        return False
    slow, fast = head, head.next
    while fast and fast.next:
        if slow == fast:
            return True
        slow = slow.next
        fast = fast.next.next
    return False

# Example Usage:
# node1 = ListNode(3)
# node2 = ListNode(2)
# node3 = ListNode(0)
# node4 = ListNode(-4)
# node1.next = node2
# node2.next = node3
# node3.next = node4
# node4.next = node2  # Cycle
# print(has_cycle(node1))
```

3.5 Sliding Window

3.5.1 Fixed Size Window

```
from typing import List

def fixed_sliding_window(arr: List[int], k: int) -> int:
    # Example: Find max sum of a subarray of size k
    if len(arr) < k:
        return 0

    current_sum = sum(arr[:k])
    max_sum = current_sum

    for i in range(k, len(arr)):
```

```
        current_sum = current_sum - arr[i-k] + arr[i]
        max_sum = max(max_sum, current_sum)

    return max_sum

# Example Usage:
# arr = [1, 4, 2, 10, 2, 3, 1, 0, 20]
# k = 4
# print(fixed_sliding_window(arr, k))
```

3.5.2 Variable Size Window

```
from typing import List

def variable_sliding_window(arr: List[int], target: int) -> int:
    # Example: Find length of smallest subarray with sum >= target
    min_length = float('inf')
    current_sum = 0
    window_start = 0

    for window_end in range(len(arr)):
        current_sum += arr[window_end]

        while current_sum >= target:
            min_length = min(min_length, window_end - window_start + 1)
            current_sum -= arr[window_start]
            window_start += 1

    return min_length if min_length != float('inf') else 0

# Example Usage:
# arr = [2, 3, 1, 2, 4, 3]
# target = 7
# print(variable_sliding_window(arr, target))
```

3.6 Binary Search

3.6.1 Standard Binary Search

```
from typing import List

def binary_search(arr: List[int], target: int) -> int:
    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

# Example Usage:
# arr = [2, 5, 7, 8, 11, 12]
# target = 13
# print(binary_search(arr, target))
```

3.6.2 First and Last Occurrence

```
from typing import List

def find_first(arr: List[int], target: int) -> int:
    left, right = 0, len(arr) - 1
    result = -1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            result = mid
            right = mid - 1
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return result

def find_last(arr: List[int], target: int) -> int:
    left, right = 0, len(arr) - 1
    result = -1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            result = mid
            left = mid + 1
        elif arr[mid] < target:
            left = mid + 1
```

```
        else:
            right = mid - 1
    return result

# Example Usage:
# arr = [5, 7, 7, 8, 8, 10]
# target = 8
# print(f"First occurrence: {find_first(arr, target)}")
# print(f"Last occurrence: {find_last(arr, target)}")
```

3.7 Dynamic Programming

3.7.1 1D DP Template (e.g., Fibonacci)

```
def fib_dp(n: int) -> int:
    if n <= 1:
        return n
    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]

# Example Usage:
# print(fib_dp(10))
```

3.7.2 2D DP Template (e.g., Unique Paths)

```
def unique_paths(m: int, n: int) -> int:
    dp = [[0] * n for _ in range(m)]

    for i in range(m):
        dp[i][0] = 1
    for j in range(n):
        dp[0][j] = 1

    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]

# Example Usage:
```

```
# print(unique_paths(3, 7))
```

3.8 Backtracking

3.8.1 General Template

```
from typing import List

def backtrack(result: List, current_path: List, other_params):
    if is_solution(current_path, other_params):
        result.append(list(current_path))
        return

    for choice in get_choices(current_path, other_params):
        if is_valid(choice, current_path, other_params):
            current_path.append(choice)
            backtrack(result, current_path, other_params)
            current_path.pop() # Backtrack

# Note: is_solution, get_choices, and is_valid are helper functions
# that you would define based on the specific problem.
```

3.8.2 Permutations

```
from typing import List

def permutations(nums: List[int]) -> List[List[int]]:
    result = []

    def backtrack(start: int):
        if start == len(nums):
            result.append(list(nums))
            return

        for i in range(start, len(nums)):
            nums[start], nums[i] = nums[i], nums[start]
            backtrack(start + 1)
            nums[start], nums[i] = nums[i], nums[start] # Backtrack

    backtrack(0)
    return result

# Example Usage:
```

```
# print(permutations([1, 2, 3]))
```

3.9 Heap Operations

```
import heapq
from typing import List

def heap_operations(nums: List[int], k: int) -> List[int]:
    # Min-heap
    min_heap = []
    for num in nums:
        heapq.heappush(min_heap, num)

    smallest = [heapq.heappop(min_heap) for _ in range(len(min_heap))]

    # Max-heap (emulated with negative numbers)
    max_heap = []
    for num in nums:
        heapq.heappush(max_heap, -num)

    largest = [-heapq.heappop(max_heap) for _ in range(len(max_heap))]

    # Find k-th largest element
    k_largest_heap = nums[:k]
    heapq.heapify(k_largest_heap)
    for i in range(k, len(nums)):
        if nums[i] > k_largest_heap[0]:
            heapq.heapreplace(k_largest_heap, nums[i])

    return k_largest_heap[0]

# Example Usage:
# nums = [3, 2, 1, 5, 6, 4]
# k = 2
# print(f"K-th largest element: {heap_operations(nums, k)}")
```

3.10 Union-Find (Disjoint Set Union)

```
from typing import List

class UnionFind:
    def __init__(self, size: int):
        self.parent = list(range(size))
```

```
        self.rank = [0] * size

    def find(self, i: int) -> int:
        if self.parent[i] == i:
            return i
        self.parent[i] = self.find(self.parent[i])  # Path compression
        return self.parent[i]

    def union(self, i: int, j: int) -> bool:
        root_i = self.find(i)
        root_j = self.find(j)
        if root_i != root_j:
            # Union by rank
            if self.rank[root_i] > self.rank[root_j]:
                self.parent[root_j] = root_i
            elif self.rank[root_i] < self.rank[root_j]:
                self.parent[root_i] = root_j
            else:
                self.parent[root_j] = root_i
                self.rank[root_i] += 1
            return True
        return False

# Example Usage:
# uf = UnionFind(10)
# uf.union(1, 2)
# uf.union(2, 5)
# print(uf.find(1) == uf.find(5))
```

3.11 Trie (Prefix Tree)

```
from typing import Dict

class TrieNode:
    def __init__(self):
        self.children: Dict[str, TrieNode] = {}
        self.is_end_of_word = False

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

    def insert(self, word: str):
```

```
        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: str) -> bool:
        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: str) -> bool:
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True

# Example Usage:
# trie = Trie()
# trie.insert("apple")
# print(trie.search("apple"))
# print(trie.starts_with("app"))
```

4 Special Algorithms

4.1 Kadane's Algorithm (Max Subarray Sum)

```
from typing import List

def kadanes_algorithm(nums: List[int]) -> int:
    max_so_far = -float('inf')
    max_ending_here = 0

    for num in nums:
        max_ending_here += num
        if max_so_far < max_ending_here:
```

```
            max_so_far = max_ending_here
        if max_ending_here < 0:
            max_ending_here = 0

    return max_so_far

# Example Usage:
# nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
# print(kadanes_algorithm(nums))
```

4.2 Prefix Sum

```
from typing import List

class PrefixSum:
    def __init__(self, nums: List[int]):
        self.prefix = [0] * (len(nums) + 1)
        for i in range(len(nums)):
            self.prefix[i+1] = self.prefix[i] + nums[i]

    def range_sum(self, left: int, right: int) -> int:
        return self.prefix[right + 1] - self.prefix[left]

# Example Usage:
# nums = [-2, 0, 3, -5, 2, -1]
# ps = PrefixSum(nums)
# print(ps.range_sum(0, 2))
```

4.3 Monotonic Stack

```
from typing import List

def monotonic_stack(nums: List[int]) -> List[int]:
    # Example: Find next greater element for each element
    stack = []
    result = [-1] * len(nums)

    for i in range(len(nums) - 1, -1, -1):
        while stack and stack[-1] <= nums[i]:
            stack.pop()
        if stack:
            result[i] = stack[-1]
        stack.append(nums[i])
```

```
    return result

# Example Usage:
# nums = [4, 5, 2, 10]
# print(monotonic_stack(nums))
```

4.4 Cyclic Sort

```
from typing import List

def cyclic_sort(nums: List[int]):
    # For arrays containing numbers from 1 to n
    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

# Example Usage:
# nums = [3, 1, 5, 4, 2]
# print(cyclic_sort(nums))
```

5 Study Plan and Routine

5.1 4-Week Study Schedule

- **Week 1: Foundations**
  - Arrays, Strings, Linked Lists
  - Two Pointers, Sliding Window
  - Basic Sorting Algorithms
  - Time/Space Complexity Analysis
- **Week 2: Trees and Graphs**
  - Tree Traversals (In-order, Pre-order, Post-order)
  - DFS, BFS
  - Topological Sort

  - Union-Find

- **Week 3: Advanced Topics**
  - Heaps (Priority Queues)
  - Tries
  - Dynamic Programming (1D and 2D)
  - Backtracking

- **Week 4: Review and Mock Interviews**
  - Review all patterns
  - Practice medium/hard LeetCode problems
  - Do mock interviews (with peers or on platforms)
  - Focus on communication and problem-solving process

5.2 Daily Practice Routine

1. **Warm-up (15-20 mins):** Solve one easy LeetCode problem to get your mind working.

2. **Pattern Practice (60-90 mins):**
   - Pick a pattern for the day.
   - Write the template from memory.
   - Solve 2-3 medium problems related to that pattern.
   - Focus on understanding the solution and trade-offs.

3. **Review (15-20 mins):**
   - Review a problem you solved a few days ago.
   - Explain the solution out loud.
   - This helps with long-term retention.

6 Time and Space Complexity Notes

A quick reference for common complexities:

- **O(1) - Constant:** Accessing an element in an array or hash map.
- **O(log n) - Logarithmic:** Binary search, operations on balanced binary search trees.
- **O(n) - Linear:** Iterating through a list, linear search.

- **O(n log n) - Log-Linear:** Efficient sorting algorithms (Merge Sort, Quick Sort), heap operations.
- **O(n^2) - Quadratic:** Nested loops (e.g., brute-force search for pairs), inefficient sorting algorithms (Bubble Sort).
- **O(2^n) - Exponential:** Recursive solutions that solve a problem of size n by solving two subproblems of size n-1 (e.g., recursive Fibonacci without memoization).
- **O(n!) - Factorial:** Generating all permutations of a set.