Coding Interview Patterns

Python Templates for Muscle Memory

Version: September 28, 2025

This printable reference collects the most frequently used patterns in modern software engineering interviews. Each template is concise, type hinted, and commented so you can rewrite it from memory.[1]

Contents

[1] Patterns consolidated from widely used interview references such as Educative's "Grokking the Coding Interview" and Tech Interview Handbook.

How to Use This Guide

- **Daily reps**: Pick 3–5 patterns, close the guide, and handwrite/type them from memory.
- **Say it out loud**: For each line, state what it does and the complexity.
- **Space repetition**: Rotate patterns; increase difficulty by removing comments.
- **Drills**: After templating, solve 1–2 problems per pattern (e.g., from LeetCode).

Core Patterns & Complete Python Templates

Depth-First Search (DFS)

Recursive DFS (Graph/Tree)

Listing 1: Recursive DFS

```python
from typing import List, Dict, Set

def dfs_recursive(graph: Dict[int, List[int]], start: int) -> List[int]:
    visited: Set[int] = set()
    order: List[int] = []

    def dfs(u: int) -> None:
        if u in visited:
            return
        visited.add(u)
        order.append(u)
        for v in graph.get(u, []):
            dfs(v)

    dfs(start)
    return order
```

Iterative DFS (Using Stack)

Listing 2: Iterative DFS

```python
from typing import List, Dict

def dfs_iterative(graph: Dict[int, List[int]], start: int) -> List[int]:
    stack: List[int] = [start]
    visited = set()
    order: List[int] = []

    while stack:
        u = stack.pop()
        if u in visited:
            continue
        visited.add(u)
        order.append(u)
        # push neighbors in reverse to mimic recursive order if needed
        for v in reversed(graph.get(u, [])):
```

```python
            if v not in visited:
                stack.append(v)
    return order
```

Breadth-First Search (BFS)

Listing 3: BFS for shortest layers in unweighted graphs

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

def bfs(graph: Dict[int, List[int]], start: int) -> List[int]:
    q = deque([start])
    visited = {start}
    order: List[int] = []

    while q:
        u = q.popleft()
        order.append(u)
        for v in graph.get(u, []):
            if v not in visited:
                visited.add(v)
                q.append(v)
    return order
```

Topological Sort (Kahn's Algorithm)

Listing 4: Kahn's algorithm for DAGs

```python
from collections import deque, defaultdict
from typing import List, Tuple

def topo_sort_kahn(n: int, edges: List[Tuple[int, int]]) -> List[int]:
    indeg = [0] * n
    g = defaultdict(list)
    for u, v in edges:
        g[u].append(v)
        indeg[v] += 1
    q = deque([i for i in range(n) if indeg[i] == 0])
    order: List[int] = []
    while q:
        u = q.popleft()
        order.append(u)
        for v in g[u]:
            indeg[v] -= 1
            if indeg[v] == 0:
                q.append(v)
    return order  # if len(order) < n, there is a cycle
```

Two Pointers

Opposite Ends (Sorted array/string)

Listing 5: Two pointers from both ends

```python
from typing import List

def two_sum_sorted(nums: List[int], target: int) -> List[int]:
    i, j = 0, len(nums) - 1
    while i < j:
        s = nums[i] + nums[j]
        if s == target:
            return [i, j]
        if s < target:
            i += 1
        else:
            j -= 1
    return []
```

Fast/Slow (Cycle detection)

Listing 6: Floyd's Tortoise and Hare

```python
from typing import Optional

class ListNode:
    def __init__(self, val: int = 0, nxt: 'Optional[ListNode]' = None):
        self.val = val
        self.next = nxt

def has_cycle(head: Optional[ListNode]) -> bool:
    slow, fast = head, head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow is fast:
            return True
    return False
```

Sliding Window

Fixed Size

Listing 7: Max sum over a window of size k

```python
from typing import List

def max_window_sum(nums: List[int], k: int) -> int:
    cur = sum(nums[:k])
    best = cur
    for i in range(k, len(nums)):
        cur += nums[i] - nums[i - k]
        best = max(best, cur)
    return best
```

Variable Size (Longest substring with at most K distinct)

Listing 8: Generic variable-size window

```python
from collections import Counter
from typing import Dict

def longest_at_most_k_distinct(s: str, k: int) -> int:
    count: Dict[str, int] = Counter()
    left = 0
    best = 0
    for right, ch in enumerate(s):
        count[ch] += 1
        while len(count) > k:
            left_ch = s[left]
            count[left_ch] -= 1
            if count[left_ch] == 0:
                del count[left_ch]
            left += 1
        best = max(best, right - left + 1)
    return best
```

Binary Search

Standard (Exact Match)

Listing 9: Binary search on sorted array

```python
from typing import List

def binary_search(nums: List[int], target: int) -> int:
    lo, hi = 0, len(nums) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if nums[mid] == target:
            return mid
        if nums[mid] < target:
            lo = mid + 1
        else:
            hi = mid - 1
    return -1
```

First/Last Occurrence (Lower/Upper Bound)

Listing 10: Lower and upper bound

```python
from typing import List, Tuple

def lower_upper_bound(nums: List[int], target: int) -> Tuple[int, int]:
    # first index >= target
    lo, hi = 0, len(nums)
    while lo < hi:
        mid = (lo + hi) // 2
        if nums[mid] < target:
```

```python
            lo = mid + 1
        else:
            hi = mid
    first = lo if lo < len(nums) and nums[lo] == target else -1

    # first index > target
    lo, hi = 0, len(nums)
    while lo < hi:
        mid = (lo + hi) // 2
        if nums[mid] <= target:
            lo = mid + 1
        else:
            hi = mid
    last = lo - 1 if lo - 1 >= 0 and nums[lo - 1] == target else -1
    return first, last
```

Dynamic Programming

1D DP (Bottom-Up)

Listing 11: Classic 1D DP (e.g., climb stairs)

```python
from typing import List

def dp_1d(n: int) -> int:
    if n <= 1:
        return 1
    dp = [0] * (n + 1)
    dp[0] = dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]
```

2D DP (Grid Paths with Obstacles)

Listing 12: 2D DP grid template

```python
from typing import List

def dp_2d(grid: List[List[int]]) -> int:
    m, n = len(grid), len(grid[0])
    dp = [[0] * n for _ in range(m)]
    dp[0][0] = 1 if grid[0][0] == 0 else 0
    for i in range(m):
        for j in range(n):
            if grid[i][j] == 1 or (i == 0 and j == 0):
                continue
            top = dp[i - 1][j] if i > 0 else 0
            left = dp[i][j - 1] if j > 0 else 0
            dp[i][j] = top + left
    return dp[m - 1][n - 1]
```

Backtracking

General Template

Listing 13: Backtracking skeleton

```
from typing import List

def backtrack_template(choices: List[int]) -> List[List[int]]:
    path: List[int] = []
    res: List[List[int]] = []

    def backtrack(start: int = 0) -> None:
        # record if path is a complete solution
        res.append(path.copy())
        for i in range(start, len(choices)):
            # choose
            path.append(choices[i])
            # explore
            backtrack(i + 1)  # or backtrack(start) for permutations with used[]
            # un-choose
            path.pop()

    backtrack(0)
    return res
```

Permutations

Listing 14: flag]Permutations using used[] flag

```
from typing import List

def permutations(nums: List[int]) -> List[List[int]]:
    res: List[List[int]] = []
    used = [False] * len(nums)
    path: List[int] = []

    def dfs() -> None:
        if len(path) == len(nums):
            res.append(path.copy())
            return
        for i, x in enumerate(nums):
            if used[i]:
                continue
            used[i] = True
            path.append(x)
            dfs()
            path.pop()
            used[i] = False

    dfs()
    return res
```

Heap Operations (heapq)

Listing 15: Min-heap, max-heap idioms, k-way merge

```python
import heapq
from typing import Iterable, List, Tuple

# Min-heap
h: List[int] = []
for x in [5, 1, 4]:
    heapq.heappush(h, x)
smallest = heapq.heappop(h)

# Max-heap via negation
h_max: List[int] = []
for x in [5, 1, 4]:
    heapq.heappush(h_max, -x)
max_val = -heapq.heappop(h_max)

# Heapify existing list
arr = [7, 2, 6]
heapq.heapify(arr)  # in-place, O(n)

# Merge k sorted lists
def merge_k_sorted(lists: List[List[int]]) -> List[int]:
    out: List[int] = []
    heap: List[Tuple[int, int, int]] = []  # (val, list_idx, elem_idx)
    for i, lst in enumerate(lists):
        if lst:
            heapq.heappush(heap, (lst[0], i, 0))
    while heap:
        val, i, j = heapq.heappop(heap)
        out.append(val)
        if j + 1 < len(lists[i]):
            heapq.heappush(heap, (lists[i][j + 1], i, j + 1))
    return out
```

Union-Find (Disjoint Set Union)

Listing 16: Path compression + union by rank

```python
from typing import List

class DSU:
    def __init__(self, n: int):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x: int) -> int:
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # path compression
        return self.parent[x]

    def union(self, x: int, y: int) -> bool:
        rx, ry = self.find(x), self.find(y)
        if rx == ry:
```

```
            return False
        if self.rank[rx] < self.rank[ry]:
            self.parent[rx] = ry
        elif self.rank[rx] > self.rank[ry]:
            self.parent[ry] = rx
        else:
            self.parent[ry] = rx
            self.rank[rx] += 1
        return True
```

Trie (Prefix Tree)

Listing 17: Trie with insert/search/prefix

```
from typing import Dict

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

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

    def insert(self, word: str) -> None:
        node = self.root
        for ch in word:
            if ch not in node.children:
                node.children[ch] = TrieNode()
            node = node.children[ch]
        node.end = True

    def search(self, word: str) -> bool:
        node = self.root
        for ch in word:
            if ch not in node.children:
                return False
            node = node.children[ch]
        return node.end

    def starts_with(self, prefix: str) -> bool:
        node = self.root
        for ch in prefix:
            if ch not in node.children:
                return False
            node = node.children[ch]
        return True
```

Special Algorithms

Kadane's Algorithm (Maximum Subarray)

Listing 18: Kadane in O(n)

```python
from typing import List

def kadane(nums: List[int]) -> int:
    best = cur = nums[0]
    for x in nums[1:]:
        cur = max(x, cur + x)
        best = max(best, cur)
    return best
```

Prefix Sum

Listing 19: Prefix sums and range queries

```python
from typing import List

def prefix_sums(nums: List[int]) -> List[int]:
    ps = [0]
    for x in nums:
        ps.append(ps[-1] + x)
    return ps  # ps[i] = sum(nums[:i])

# sum of nums[l:r] (l inclusive, r exclusive)
def range_sum(ps: List[int], l: int, r: int) -> int:
    return ps[r] - ps[l]
```

Monotonic Stack

Listing 20: Next greater element (increasing stack)

```python
from typing import List

def next_greater(nums: List[int]) -> List[int]:
    res = [-1] * len(nums)
    stack: List[int] = []  # indices, stack maintains decreasing values
    for i, x in enumerate(nums):
        while stack and nums[stack[-1]] < x:
            j = stack.pop()
            res[j] = x
        stack.append(i)
    return res
```

Cyclic Sort (Arrays with 1..n)

Listing 21: Place numbers at correct indices

```python
from typing import List

def cyclic_sort(nums: List[int]) -> None:
    i = 0
    n = len(nums)
    while i < n:
        j = nums[i] - 1
```

```python
        if 0 <= j < n and nums[i] != nums[j]:
            nums[i], nums[j] = nums[j], nums[i]
        else:
            i += 1
# in-place; useful for finding missing/duplicate numbers in 1..n
```

Pattern Recognition Guide (When to Use What)

- **Two Pointers**: Sorted arrays/strings; pair sums; palindrome checks; removing duplicates in-place.
- **Sliding Window**: Substrings/subarrays asking for longest/shortest/# satisfying constraints; streaming data.
- **Binary Search**: Sorted arrays; monotonic predicates ("can we?" decision problems); answer-search on range.
- **BFS**: Shortest path in unweighted graphs; nearest/levels; minimum steps.
- **DFS**: Connectivity; components; recursion on trees; cycle detection (directed via color/stack).
- **Topological Sort**: Prerequisites; ordering tasks; detect cycles in DAG.
- **Union-Find**: Dynamic connectivity; components after unions; cycle detection in undirected graphs; Kruskal MST.
- **Trie**: Prefix queries; autocomplete; word break; dictionary problems.
- **DP (1D/2D)**: Overlapping subproblems with optimal substructure; sequences; knapsack; edit distance; grid paths.
- **Backtracking**: Generate all valid configurations under constraints (permutations, subsets, N-Queens, parentheses).
- **Heap**: Top-k / k-way merge / running median; scheduling by priority.
- **Kadane/Prefix Sum**: Max subarray; quick range sums; difference arrays.
- **Monotonic Stack**: Next greater/smaller; histogram area; sliding window min/max (deque).
- **Cyclic Sort**: Arrays containing $1..n$ to find missing/duplicate numbers quickly.

Time & Space Complexity Notes

- DFS/BFS: $O(V + E)$ time, $O(V)$ space (visited + recursion/queue).
- Topo Sort (Kahn): $O(V + E)$ time, $O(V)$ space.
- Two Pointers/Sliding Window: Usually $O(n)$ time, $O(1)$ or $O(K)$ space.
- Binary Search: $O(\log n)$ time, $O(1)$ space.
- DP: Depends on states; 1D often $O(n)$, 2D often $O(mn)$; can optimize space by rolling arrays.
- Backtracking: Exponential in solutions; prune aggressively.

- Heaps: Push/pop $O(\log n)$; heapify $O(n)$; k-way merge $O(N \log k)$.
- Union-Find: $\alpha(n)$ inverse Ackermann for amortized almost-constant ops.
- Trie: Insert/search $O(L)$ where L is word length; space proportional to stored characters.
- Monotonic Stack/Deque: Each element in/out once $\Rightarrow O(n)$ time, $O(n)$ space.

Daily Practice Routine (45–90 minutes)

1. **Warm-up (5 min)**: Rewrite 1–2 templates from memory (rotate daily).
2. **Focused Drill (25–45 min)**: Solve 2 problems of the day's pattern. After each, restate the approach and complexity.
3. **Spaced Repetition (10 min)**: Flash-review 3 older problems; explain aloud without coding.
4. **Retrospective (5 min)**: Note mistakes and update your trigger words for pattern recognition.

4-Week Study Schedule

Week 1: Fundamentals

Two Pointers, Sliding Window, Binary Search, Prefix Sum, Kadane.

- Day 1–2: Two Pointers (pairs, palindromes, dedupe)
- Day 3–4: Sliding Window (fixed/variable, substrings)
- Day 5: Binary Search (bounds, predicate search)
- Day 6: Prefix Sums + Kadane
- Day 7: Mixed review quiz + rewrite 5 templates

Week 2: Graphs & Sets

DFS (rec/iter), BFS, Topological Sort, Union-Find.

- Day 8–9: DFS/BFS on trees/graphs; shortest paths (unweighted)
- Day 10: Cycle detection; components; bipartite check
- Day 11: Topological Sort (Kahn) + prerequisites problems
- Day 12: Union-Find (dynamic connectivity, Kruskal-style)
- Day 13: Heaps (top-k, k-way merge)
- Day 14: Review + mock

Week 3: DP & Backtracking

1D/2D DP, classic sequences, and generation problems.

- Day 15–16: 1D DP (climb stairs, house robber, coin change)

- Day 17–18: 2D DP (grid paths, edit distance, LCS)
- Day 19: State compression / rolling arrays
- Day 20: Backtracking (subsets, permutations, combos)
- Day 21: Review + speed drills

Week 4: Mastery & Patterns Mix

Tries, Monotonic Stack, Cyclic Sort, mixed sets, and timed mocks.

- Day 22: Trie (prefix queries, word search)
- Day 23: Monotonic Stack (NGE, histogram, daily temps)
- Day 24: Cyclic Sort (missing/duplicate in 1..n)
- Day 25–26: Mixed sets under time pressure
- Day 27: Full mock (70 min) + analysis
- Day 28: Final review + rewrite all templates once

Tips for Muscle Memory

- Build mental *triggers*: e.g., "longest/shortest subarray/substring" $\Rightarrow$ sliding window.
- Keep idioms ready: bounds patterns for binary search; deque for window min/max; DSU for connectivity.
- Practice explaining: interviewers value clarity and tradeoffs as much as code.
- Track your misses: create a "red list" of patterns you consistently forget.

This reference is intentionally compact. Recopy it until it becomes automatic. Good luck!