Coding Interview (Meta L6) — 2-page print cheat sheet

Goal: never blank on patterns, templates, edge cases, or communication signals.

1) Universal problem-solving script (use every time)

- Restate problem + constraints; confirm input/output types; clarify duplicates/ordering/ties.
- Ask for bounds: n, value ranges, memory/time limits; streaming? mutability? recursion depth?
- Do 1-2 examples by hand (including tricky/edge).
- Propose solution(s): baseline then optimal; pick one and justify.
- State complexity **before** coding: time + space; mention dominating term.
- Code in small chunks; narrate invariants; avoid cleverness unless needed.
- Validate with tests: happy path + edges + adversarial; then final complexity recap.

2) Edge-case & bug checklist (quick scan before you hit run)

- **Empty/single:** n=0/1; None; missing keys; empty string.
- **Bounds:** off-by-one, inclusive/exclusive ranges, k=0/k>n, negative numbers.
- **Overflow:** sum of large ints; use Python's bigint or check bounds; in other langs watch int limits.
- **Duplicates:** stable ordering? multiple identical values? set vs multiset behavior.
- **Cycles:** graphs/linked lists; visited handling; parent pointers; self-loops.
- **Mutability:** modifying list while iterating; aliasing; copy vs reference; default mutable args.
- **Termination:** while loops; recursion base cases; pointer movement always progresses.
- **Floats:** precision issues; use abs(a-b) < eps; avoid == on floats.
- **Complexity:** hidden O(n²) (nested loops, list pop(0), string concat, repeated slicing).

3) Pattern picker (map problem -> tool)

- **Hash map/set:** membership, counts, complements, first occurrence, de-dup.
- **Two pointers:** sorted arrays, remove/partition, palindromes, pair sums.
- **Sliding window:** subarray/substring with constraints (≤k, exactly k, at most k).
- **Monotonic stack:** next greater/smaller, histogram, span, remove k digits.
- **Heap:** top-k, k-way merge, streaming median, schedule/interval greediness.
- **Binary search:** sorted + "first/last true" on monotonic predicate; answer-space search.
- **BFS/DFS:** grid/graph connectivity, shortest path unweighted (BFS), components, topo.
- **DP:** optimal substructure; knapsack; sequences; grid paths; string edit-like.
- **Union-Find:** dynamic connectivity, grouping, cycle detection, Kruskal.
- **Intervals:** merge, sweep line, meeting rooms, overlaps; sort by start/end.

4) Meta favorites (high-frequency patterns)

- **Trees:** LCA, serialize/deserialize, path sums, BST validation, iterators.
- **Graphs:** clone graph, detect cycle, shortest path, number of islands, course schedule.
- **Arrays:** subarray sum, product except self, merge intervals, meeting rooms.
- **Strings:** valid parentheses, decode ways, word break, regex/wildcard.
- **Design:** LRU/LFU cache, iterator patterns, random with weights.
- **Follow-ups:** optimize space, handle streaming, scale to distributed, add concurrency.

5) Core invariants to say out loud (signal seniority)

- **Window invariant:** current window always satisfies condition; when violated, shrink left until restored.
- **Two pointers:** pointers move monotonically; each element processed O(1) times => O(n).
- **Stack:** stack maintains monotonic property; each element pushed/popped once => O(n).
- **BFS:** first time you pop a node = shortest distance in unweighted graph.
- **Dijkstra:** min-heap pop finalizes shortest path when edges non-negative.
- **Binary search:** maintain [lo, hi) (or inclusive) + monotonic predicate; prove termination.
- **DP:** define state + transition + base cases + iteration order; confirm no future deps.

6) Complexity cheats (common ops)

list append O(1) amort; pop() O(1); pop(0) O(n); sort O(n log n); dict/set avg O(1); heap push/pop O(log n); deque popleft O(1); bisect O(log n); union-find ~ inverse Ackermann.

7) L6 signals in coding round (what distinguishes you)

- **Drive:** you lead with structure and ask constraints early (no thrash).
- **Clarity:** crisp explanation + invariants; you can prove correctness informally.
- **Robustness:** edge cases, input validation, careful about complexity landmines.
- **Quality:** readable code, good names, small helpers, minimal bugs.
- **Speed:** you converge quickly: identify pattern, state plan, then implement.

8) Handling follow-ups gracefully (L6 must-have)

- **'Can you do better?'** → state current complexity, propose next approach (hash/sort/heap), explain tradeoff.
- **'What if input is huge?'** → streaming, external sort, approximate (bloom/HLL), chunking, MapReduce framing.
- **'What if concurrent?'** → locks, CAS, thread-safe structures, immutability; mention race conditions.
- **'What about scale?'** → partitioning/sharding, caching, async, batching; state bottlenecks.
- **Don't panic:** 'Good question—let me think.' Pause, state constraints, propose approach.

Python templates (minimal, interview-friendly)

Sliding window (at most k / generic constraint)

```
l = 0
for r, x in enumerate(arr):
    add(x)
    while not ok():   # window violates constraint
        remove(arr[l]); l += 1
    ans = best(ans, r - l + 1)
```

Tip: for "exactly k distinct" => f(at_most(k)) - f(at_most(k-1)).

Binary search on answer (first True)

```
lo, hi = low_bound, high_bound  # hi exclusive
while lo < hi:
    mid = (lo + hi) // 2
    if feasible(mid):
        hi = mid
    else:
        lo = mid + 1
return lo
```

Always define feasible(mid) monotonic; state invariants for [lo, hi).

Graph/Tree traversal

```
from collections import deque

# BFS shortest path (unweighted)
q = deque([src])
dist = {src: 0}
while q:
    u = q.popleft()
    for v in nbrs[u]:
        if v not in dist:
            dist[v] = dist[u] + 1
            q.append(v)
# DFS iterative
stack = [src]; seen = set([src])
while stack:
    u = stack.pop()
    for v in nbrs[u]:
        if v not in seen:
            seen.add(v)
            stack.append(v)
```

Topological sort (Kahn)

```
from collections import deque
indeg = {u: 0 for u in nodes}
for u in nodes:
    for v in nbrs[u]: indeg[v] += 1
q = deque([u for u in nodes if indeg[u] == 0])
order = []
while q:
    u = q.popleft(); order.append(u)
    for v in nbrs[u]:
        indeg[v] -= 1
        if indeg[v] == 0: q.append(v)
```

If len(order) < n => cycle.

Heap patterns

```
import heapq

# top-k largest via min-heap size k
h = []
for x in nums:
    heapq.heappush(h, x)
    if len(h) > k: heapq.heappop(h)
# h holds k largest
```

For max-heap: push -x; or store (-key, item).

Union-Find (DSU)

```
parent = {x: x for x in items}
rank = {x: 0 for x in items}

def find(x):
    while parent[x] != x:
        parent[x] = parent[parent[x]]
        x = parent[x]
    return x

def union(a, b):
```

```
    ra, rb = find(a), find(b)
    if ra == rb: return False
    if rank[ra] < rank[rb]: ra, rb = rb, ra
    parent[rb] = ra
    if rank[ra] == rank[rb]: rank[ra] += 1
    return True
```

DP memoization (avoid recomputation)

```
from functools import lru_cache

@lru_cache(None)
def dp(i, state):
    if i == n: return base
    ans = ...
    return ans
```

State must be hashable; watch recursion depth; consider iterative DP if deep.

Backtracking (permutations/subsets/combinations)

```
def backtrack(path, choices):
    if done(path):
        ans.append(path[:]); return
    for i, c in enumerate(choices):
        if not valid(c): continue
        path.append(c)
        backtrack(path, choices[i+1:])  # or choices[:i]+choices[i+1:] for perms
        path.pop()

ans = []; backtrack([], items)
```

For permutations: use 'used' set or swap in-place. For pruning: skip invalid early.

Prefix sum (range queries O(1))

```
pre = [0]*(n+1)
for i, x in enumerate(arr):
    pre[i+1] = pre[i] + x
# sum [l..r) = pre[r] - pre[l]

# 2D prefix sum:
pre[i][j] = val + pre[i-1][j] + pre[i][j-1] - pre[i-1][j-1]
# query (r1,c1,r2,c2) = pre[r2][c2] - pre[r1-1][c2] - pre[r2][c1-1] + pre[r1-1][c1-1]
```

Interval merge / sweep line

```
intervals.sort(key=lambda x: x[0])
merged = []
for s, e in intervals:
    if merged and s <= merged[-1][1]:
        merged[-1][1] = max(merged[-1][1], e)
    else:
        merged.append([s, e])

# Meeting rooms II (min rooms): sweep line
events = []
for s, e in intervals:
    events.append((s, 1))   # start
    events.append((e, -1))  # end
events.sort()
max_rooms = cur = 0
for _, delta in events:
    cur += delta
    max_rooms = max(max_rooms, cur)
```

Linked list essentials

```
# Reverse in place
def reverse(head):
    prev = None
    while head:
        nxt = head.next
        head.next = prev
        prev = head
        head = nxt
    return prev

# Detect cycle (Floyd)
def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow, fast = slow.next, fast.next.next
        if slow == fast: return True
    return False

# Find middle (slow at mid when fast at end)
```

Quick select (k-th smallest, avg O(n))

```
import random
def quickselect(arr, k):  # k is 0-indexed
    if len(arr) == 1: return arr[0]
    pivot = random.choice(arr)
    lo = [x for x in arr if x < pivot]
    eq = [x for x in arr if x == pivot]
    hi = [x for x in arr if x > pivot]
    if k < len(lo): return quickselect(lo, k)
    elif k < len(lo) + len(eq): return pivot
    else: return quickselect(hi, k - len(lo) - len(eq))
```

Use random pivot to avoid worst case. For top-k, partition around k.

Python standard library reminders (fast access)

- **collections:** deque, Counter, defaultdict
- **heapq:** heappush/heappop/heappushpop/nlargest/nsmallest
- **bisect:** bisect_left/right for insertion points
- **itertools:** accumulate, product, combinations
- **math:** inf, gcd, isqrt

Final 60-second close (use if time)

Recap approach + invariant. Walk through 1 edge case. State final time/space. Mention any alternative solution and why you didn't pick it.

Coding Addendum — common algorithms & 'from scratch' snippets

Use when the problem screams for it. Keep explanations tight: invariant + why it's optimal + complexity.

1) Shortest paths (graph/grid)

Dijkstra (non-negative weights)

```
import heapq
INF = 10**18
dist = {src: 0}
pq = [(0, src)]
while pq:
    d, u = heapq.heappop(pq)
    if d != dist.get(u, INF):
        continue
    if u == tgt: break
    for v, w in nbrs[u]:
        nd = d + w
        if nd < dist.get(v, INF):
            dist[v] = nd
            heapq.heappush(pq, (nd, v))
```

Tip: for grid, neighbors are 4/8 dirs; for path reconstruct keep parent[v]=u.

0-1 BFS (weights are 0/1)

```
from collections import deque
INF = 10**18
dist = {src: 0}
dq = deque([src])
while dq:
    u = dq.popleft()
    for v, w in nbrs[u]:      # w in {0,1}
        nd = dist[u] + w
        if nd < dist.get(v, INF):
            dist[v] = nd
            if w == 0: dq.appendleft(v)
            else: dq.append(v)
```

A* (grid/pathfinding; admissible heuristic)

```
import heapq, math
def h(u):  # admissible: never overestimates
    return abs(u.x - tgt.x) + abs(u.y - tgt.y)  # manhattan for 4-neigh

INF = 10**18
g = {src: 0}
pq = [(h(src), 0, src)]  # (f=g+h, g, node)
parent = {src: None}
while pq:
    f, gu, u = heapq.heappop(pq)
    if gu != g.get(u, INF):
        continue
    if u == tgt: break
    for v, w in nbrs[u]:
        nv = gu + w
        if nv < g.get(v, INF):
            g[v] = nv
            parent[v] = u
            heapq.heappush(pq, (nv + h(v), nv, v))
```

Notes: heuristic must be admissible (and ideally consistent). A* reduces to Dijkstra if h=0.

Bellman-Ford (handles negative weights; detects neg cycles)

```
INF = 10**18
dist = [INF]*n
dist[src] = 0
for _ in range(n-1):
    changed = False
    for u, v, w in edges:
        if dist[u] != INF and dist[u] + w < dist[v]:
            dist[v] = dist[u] + w
            changed = True
    if not changed: break

# one more pass => negative cycle reachable
neg_cycle = any(dist[u] != INF and dist[u] + w < dist[v] for u, v, w in edges)
```

Use only when needed; O(VE).

2) Minimum spanning tree (MST)

Prim (dense-ish graphs)

```
import heapq
seen = set([start])
pq = []
for v, w in nbrs[start]:
    heapq.heappush(pq, (w, start, v))
mst = []
while pq and len(seen) < n:
    w, u, v = heapq.heappop(pq)
    if v in seen:
        continue
    seen.add(v); mst.append((u, v, w))
    for x, wx in nbrs[v]:
        if x not in seen:
            heapq.heappush(pq, (wx, v, x))
```

Kruskal + DSU is great for sparse graphs (you already have DSU on page 2).

3) Heap from scratch (min-heap)

```
# 0-indexed array heap
def heappush(h, x):
    h.append(x)
    i = len(h) - 1
    while i > 0:
        p = (i - 1) // 2
        if h[p] <= h[i]: break
        h[p], h[i] = h[i], h[p]
        i = p

def heappop(h):
    x = h[0]
    last = h.pop()
    if h:
        h[0] = last
        i = 0
        n = len(h)
        while True:
            l = 2*i + 1; r = l + 1
            if l >= n: break
            m = l
            if r < n and h[r] < h[l]: m = r
            if h[i] <= h[m]: break
            h[i], h[m] = h[m], h[i]
            i = m
    return x
```

Use to show fundamentals if asked; otherwise prefer heapq (less bug risk).

4) Range queries (BIT / Segment Tree) + Trie

Fenwick / BIT (prefix sums)

```
# 1-indexed BIT
bit = [0]*(n+1)
def add(i, delta):
    i += 1
    while i <= n:
        bit[i] += delta
        i += i & -i
def sum_(i):  # sum [0..i)
    s = 0
    while i > 0:
        s += bit[i]
        i -= i & -i
    return s
# range sum [l..r): sum_(r) - sum_(l)
```

Segment tree is for min/max/gcd or non-invertible ops; BIT is simpler for sums.

Monotonic stack (next greater/smaller)

```
# Next greater element to right
nge = [-1] * n
stack = []  # indices, decreasing values
for i, x in enumerate(arr):
    while stack and arr[stack[-1]] < x:
        nge[stack.pop()] = x
    stack.append(i)
# For next smaller: change < to >
# For left: iterate backwards or use different approach
```

Trie (prefix search)

```
class Node:
    __slots__ = ("next", "end")
    def __init__(self):
        self.next = {}
        self.end = False

root = Node()
def insert(s):
    cur = root
    for ch in s:
        cur = cur.next.setdefault(ch, Node())
    cur.end = True
```

5) String algorithms (when brute force times out)

KMP prefix function (pattern search)

```
# pi[i] = length of longest proper prefix == suffix for s[:i+1]
pi = [0]*m
j = 0
for i in range(1, m):
    while j > 0 and p[i] != p[j]:
        j = pi[j-1]
    if p[i] == p[j]: j += 1
    pi[i] = j

# scan text
j = 0
for i, ch in enumerate(t):
    while j > 0 and ch != p[j]:
        j = pi[j-1]
    if ch == p[j]: j += 1
    if j == m:
        return i - m + 1
```

Alternative: Rabin-Karp for average-case hashing; KMP is deterministic O(n+m).

6) Systems-y coding: LRU cache (common ask)

```
from collections import OrderedDict

class LRUCache:
    def __init__(self, cap):
        self.cap = cap
        self.od = OrderedDict()

    def get(self, k):
        if k not in self.od: return -1
        self.od.move_to_end(k)
        return self.od[k]

    def put(self, k, v):
        if k in self.od: self.od.move_to_end(k)
        self.od[k] = v
        if len(self.od) > self.cap:
            self.od.popitem(last=False)
```

If they demand “from scratch”, implement doubly-linked list + hashmap.

Last-moment reminders

- If a graph has **negative weights**: Dijkstra is wrong (use Bellman-Ford / SPFA-ish; or reframe).
- If asking for **shortest path on grid**: BFS (unweighted), 0-1 BFS (0/1), Dijkstra (weighted), A* (with heuristic).
- For **state-space search**: define state, transitions, and visited key; beware exponential blowups.
- When implementing from scratch: keep invariants, write tiny helper functions, test small cases.