DSA Patterns Cheat Sheet

12 patterns that solve most interview and OA questions. Python templates, all tested.

Chirag Kashyap
chiragdev.org · @code.withchirag

How to use: read the question, match it to a **"Use it when"** line, start from that template, then solve the 4 practice problems for that pattern (LeetCode numbers in brackets). Don't memorise code: learn *when* each pattern fits.

01 Two pointers

Time O(n) · Space O(1)

USE IT WHEN

- Array is **sorted** (or you can sort it)
- Find a pair / triplet with a target sum
- Compare from both ends: palindrome, reverse, merge

PRACTICE

- Two Sum II (167)
- Valid Palindrome (125)
- 3Sum (15)
- Container With Most Water (11)

```
def pair_sum(a, target):          # a is sorted
    l, r = 0, len(a) - 1
    while l < r:
        s = a[l] + a[r]
        if s == target:
            return l, r
        if s < target:
            l += 1
        else:
            r -= 1
    return -1, -1
```

02 Sliding window

Time O(n) · Space O(k)

USE IT WHEN

- **Contiguous** subarray or substring
- Longest / shortest window that satisfies a rule
- "at most K distinct", "no repeats", fixed size k

PRACTICE

- Longest Substring Without Repeating (3)
- Max Sum Subarray of size K
- Minimum Window Substring (76)
- Permutation in String (567)

```
def longest_unique(s):
    seen, l, best = {}, 0, 0
    for r, ch in enumerate(s):
        if ch in seen and seen[ch] >= l:
            l = seen[ch] + 1       # shrink past the repeat
        seen[ch] = r
        best = max(best, r - l + 1)
    return best
```

03 Prefix sum + hashmap

Time O(n) · Space O(n)

USE IT WHEN

- **Sum of a range** asked many times
- Count subarrays with sum = K (works with negatives)
- Equal 0s and 1s, divisible by K

PRACTICE

- Subarray Sum Equals K (560)
- Range Sum Query (303)
- Contiguous Array (525)
- Product of Array Except Self (238)

```
def subarrays_with_sum(a, k):
    count, run = 0, 0
    freq = {0: 1}                  # empty prefix
    for x in a:
        run += x
        count += freq.get(run - k, 0)
        freq[run] = freq.get(run, 0) + 1
    return count
```

04 Binary search (on the answer too)

Time O(log n) · Space O(1)

USE IT WHEN

- Sorted input, or a yes/no check that flips **once**
- "Minimum X such that ..." / "maximum X such that ..."
- Search space is a number range, not an array

PRACTICE

- Binary Search (704)
- Search in Rotated Array (33)
- Koko Eating Bananas (875)
- Ship Packages in D Days (1011)

```
def first_true(lo, hi, ok):       # ok() is False...False True...True
    while lo < hi:
        mid = (lo + hi) // 2
        if ok(mid):
            hi = mid
        else:
            lo = mid + 1
    return lo
```

05 Monotonic stack

Time O(n) · Space O(n)

USE IT WHEN

- **Next greater / smaller** element
- Span problems: stock span, daily temperatures
- Largest rectangle, valid brackets

PRACTICE

- Next Greater Element (496)
- Daily Temperatures (739)
- Valid Parentheses (20)
- Largest Rectangle in Histogram (84)

```
def next_greater(a):
    ans, stack = [-1] * len(a), []    # stack holds indexes
    for i, x in enumerate(a):
        while stack and a[stack[-1]] < x:
            ans[stack.pop()] = x
        stack.append(i)
    return ans
```

06 Recursion & backtracking

Time O(2ⁿ) / O(n!) · Space O(n)

USE IT WHEN

- Generate **all** subsets, permutations, combinations
- Choose → explore → un-choose
- n is small (≤ 20)

PRACTICE

- Subsets (78)
- Permutations (46)
- Combination Sum (39)
- N-Queens (51)

```
def subsets(nums):
    out, path = [], []
    def go(i):
        if i == len(nums):
            out.append(path[:])
            return
        path.append(nums[i]); go(i + 1)   # take
        path.pop();           go(i + 1)   # skip
    go(0)
    return out
```

07 BFS (shortest path, unweighted)

Time O(V + E) · Space O(V)

USE IT WHEN

- **Minimum steps / moves** on a grid or graph
- Level-order traversal of a tree
- Spread from many sources at once (rotting oranges)

PRACTICE

- Binary Tree Level Order (102)
- Rotting Oranges (994)
- Shortest Path in Binary Matrix (1091)
- Word Ladder (127)

```
from collections import deque
def shortest_steps(graph, start, goal):
    dist = {start: 0}
    q = deque([start])
    while q:
        u = q.popleft()
        if u == goal:
            return dist[u]
        for v in graph[u]:
            if v not in dist:
                dist[v] = dist[u] + 1
                q.append(v)
    return -1
```

08 DFS (flood fill, components)

Time O(V + E) · Space O(V)

USE IT WHEN

- Count **connected** regions or islands
- Explore every path, detect cycles
- Tree height, path sums

PRACTICE

- Number of Islands (200)
- Max Depth of Binary Tree (104)
- Clone Graph (133)
- Course Schedule (207)

```
def count_islands(grid):
    rows, cols, seen = len(grid), len(grid[0]), set()
    def dfs(r, c):
        if not (0 <= r < rows and 0 <= c < cols): return
        if grid[r][c] != "1" or (r, c) in seen: return
        seen.add((r, c))
        for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
            dfs(r + dr, c + dc)
    islands = 0
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == "1" and (r, c) not in seen:
                dfs(r, c); islands += 1
    return islands
```

09 Heap / top-K

Time O(n log k) · Space O(k)

USE IT WHEN

- **K largest / smallest / most frequent**
- Always need the current min or max
- Merge K sorted lists

PRACTICE

- Kth Largest Element (215)
- Top K Frequent Elements (347)
- Merge K Sorted Lists (23)
- Find Median from Data Stream (295)

```
import heapq
def top_k(nums, k):
    heap = []                       # min-heap of size k
    for x in nums:
        heapq.heappush(heap, x)
        if len(heap) > k:
            heapq.heappop(heap)
    return sorted(heap, reverse=True)
```

10 Greedy (intervals)

Time O(n log n) · Space O(1)

USE IT WHEN

- Sort, then take the **locally best** choice
- Intervals: merge, max non-overlapping, meeting rooms
- Jump game, coin change with standard coins

PRACTICE

- Non-overlapping Intervals (435)
- Merge Intervals (56)
- Jump Game (55)
- Assign Cookies (455)

```
def max_non_overlapping(intervals):
    intervals.sort(key=lambda iv: iv[1])   # earliest end first
    count, end = 0, float("-inf")
    for s, e in intervals:
        if s >= end:
            count, end = count + 1, e
    return count
```

11 Dynamic programming: 1D

Time O(n) · Space O(1)–O(n)

USE IT WHEN

- "Number of ways", "min cost", "max profit"
- Answer for i depends on i−1, i−2 …
- Recursion with repeated sub-calls → add memo

PRACTICE

- Climbing Stairs (70)
- House Robber (198)
- Coin Change (322)
- Longest Increasing Subsequence (300)

```
def climb_stairs(n):
    a, b = 1, 1                     # ways(0), ways(1)
    for _ in range(n - 1):
        a, b = b, a + b
    return b
```

12 Dynamic programming: 2D

Time O(n·m) · Space O(n·m)

USE IT WHEN

- **Two strings** or a grid
- dp[i][j] = answer for the first i and first j
- Match → diagonal; no match → best of top / left

PRACTICE

- Longest Common Subsequence (1143)
- Edit Distance (72)
- Unique Paths (62)
- 0/1 Knapsack

```
def lcs(x, y):
    dp = [[0] * (len(y) + 1) for _ in range(len(x) + 1)]
    for i in range(1, len(x) + 1):
        for j in range(1, len(y) + 1):
            if x[i - 1] == y[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[-1][-1]
```