FREE · LEARN.AIWITHPRIYABRAT.COM

DSA Patterns for Coding Interviews

The 15 patterns behind most coding interview questions: how to spot each one from the problem's wording, a reusable Python template, a worked example with its real output, the Big-O, and the classic problems to practise. Plus 4 advanced patterns, a 4-week plan and what to say in the room.

By **Priyabrat Dalbehera** · GenAI engineer

13 pages · **12** parts · **19** patterns · **59** examples run, real output · **90+** practice problems

What's inside

1 **Approach** — How to approach any coding problem — 2

2 **Arrays & strings** — Two pointers · sliding window — 3

3 **Lookups** — Prefix sums · hash maps · binary search — 4

4 **Lists & stacks** — Linked lists · stacks · monotonic stack — 5

5 **Intervals & heaps** — Intervals · heaps · top K · k-way merge — 6

6 **Trees** — DFS · BFS · BST · LCA · trie — 7

7 **Graphs** — Grids · BFS · topological sort · union-find · Dijkstra — 8

8 **Backtracking** — Subsets · permutations · N-Queens · bits — 9

9 **Dynamic programming** — 1D · 2D · knapsack · LIS · edit distance — 10

10 **Greedy & tricks** — Greedy · Kadane · cyclic sort · matrix — 11

11 **Interview day** — Complexity · edge cases · 4-week plan — 12

12 **Key insights & extras** — Key insights · monotonic deque · difference array · quickselect · Fenwick — 13

Spot it
Start with the signals table on page 2: the problem's wording points to the pattern.

Run it
Every example is plain Python 3 and was run to print the output shown (helpers: 11.8).

Practise it
Each part ends with classic problems. Solve them in the order listed.

learn.aiwithpriyabrat.com

© 2026 Priyabrat Dalbehera · Free to download and share unchanged, with credit · Not for resale

1.1 The 6-step method (say each step out loud)

1. Clarify	inputs, sizes, sorted?, duplicates?, negatives?, what to return
2. Examples	walk one normal case and one edge case by hand
3. Brute force	state it and its Big-O, even if you won't code it
4. Pattern	match the problem's signals (1.2) to a faster approach
5. Code	clean names, small helpers, narrate as you type
6. Test	dry-run your example, then edge cases; give time + space

Tip: Interviewers score communication as much as the answer. A clear brute force beats a silent, half-finished optimal solution.

1.2 Spot the pattern from the problem's signals

If you see...	Try	p.
sorted array, pair / triplet with a sum	Two pointers	3
longest / shortest contiguous subarray or substring	Sliding window	3
sum of a range, subarray sum = k	Prefix sum + hash map	4
seen before?, counts, pairs, anagrams	Hash map / set	4
sorted input, or 'minimum X such that ...'	Binary search	4
linked list cycle, middle, k-th from end	Fast & slow pointers	5
next greater / smaller, matching brackets	(Monotonic) stack	5
overlapping ranges, meetings, schedules	Intervals: sort by start	6
top / k-th largest, k closest, stream median	Heap	6
tree paths, depth, levels	DFS / BFS on trees	7
prefix lookup, autocomplete, word list	Trie	7
grid regions, shortest steps, dependencies	Graph BFS / DFS / topo sort	8
all subsets / permutations / combinations	Backtracking	9
count ways, min / max cost, overlapping choices	Dynamic programming	10
local best choice is provably safe	Greedy	11

1.3 What the input size tells you

Constraint	Target	Typical approach
$n \le 12$	$O(n!)$	permutations, brute force
$n \le 25$	$O(2^n)$	subsets, backtracking, bitmask DP
$n \le 500$	$O(n^3)$	3 nested loops, interval DP
$n \le 5{,}000$	$O(n^2)$	pairs, 2D DP
$n \le 10^6$	$O(n \log n)$	sort, heap, binary search
$n \le 10^8$	$O(n)$	one pass, two pointers, hashing
bigger	$O(\log n)$ / $O(1)$	math, binary search on the answer

Rule of thumb: about 10^8 simple operations per second in C++, 10^7 in Python.

1.4 Big-O of everyday Python operations

`list` append / pop() / index	O(1)
`list` insert(0) / pop(0) / `x in list`	O(n)
`dict` / `set` get, add, `in`	O(1) average
`deque` appendleft / popleft	O(1)
`heapq` push / pop · heapify	O(log n) · O(n)
`sorted()` / `list.sort()`	O(n log n)
`bisect` search · insort	O(log n) · O(n)
slice `a[i:j]` · string `+` in a loop	O(j−i) · $O(n^2)$ total

1.5 The Python toolkit you'll use everywhere

```python
from collections import Counter, defaultdict, deque
import heapq, bisect, math
from functools import cache      # every example assumes these

print(Counter("banana").most_common(2))
g = defaultdict(list); g["a"].append(1); print(dict(g))
q = deque([1, 2]); q.appendleft(0); q.append(3)
print(q.popleft(), list(q))
h = [5, 1, 4]; heapq.heapify(h)
print(heapq.heappop(h), heapq.nsmallest(2, [7, 3, 9]))
print(bisect.bisect_left([1, 3, 3, 5], 3))
print(math.inf > 10**18, divmod(17, 5))
```

Output

```
[('a', 3), ('n', 2)]
{'a': [1]}
0 [1, 2, 3]
1 [3, 7]
1
True (3, 2)
```

Max-heap: push `-x` and negate when you pop.

1.6 Recursion and memoization

```python
calls = 0
def fib(n):
    global calls; calls += 1
    return n if n < 2 else fib(n - 1) + fib(n - 2)
print(fib(20), calls)            # exponential

@cache
def fib_memo(n):
    return n if n < 2 else fib_memo(n - 1) + fib_memo(n - 2)
print(fib_memo(90), fib_memo.cache_info().misses)
```

Output

```
6765 21891
2880067194370816120 91
```

Every recursion needs a **base case** and a step that makes the input **smaller**. If the same arguments repeat, memoize: that's top-down DP (part 9).

1.7 Sorting tricks

```python
people = [("asha", 30), ("ravi", 25), ("meera", 30)]
print(sorted(people, key=lambda p: (-p[1], p[0])))

from functools import cmp_to_key
def largest_number(nums):        # [3, 30, 34] -> "34330"
    order = lambda a, b: (b + a > a + b) - (b + a < a + b)
    s = sorted(map(str, nums), key=cmp_to_key(order))
    return "".join(s).lstrip("0") or "0"
print(largest_number([3, 30, 34, 5, 9]))
```

Output

```
[('asha', 30), ('meera', 30), ('ravi', 25)]
9534330
```

Python's sort is stable: sort by the secondary key first, then by the primary key. Counting sort beats O(n log n) when values are small integers.

learn.aiwithpriyabrat.com

2.1 Two pointers from both ends (sorted input)

```python
def pair_with_sum(nums, target):
    lo, hi = 0, len(nums) - 1
    while lo < hi:
        s = nums[lo] + nums[hi]
        if s == target:
            return [lo, hi]
        if s < target:
            lo += 1          # need a bigger sum
        else:
            hi -= 1          # need a smaller sum
    return []

print(pair_with_sum([1, 3, 4, 6, 8, 11], 10))
```

Output

```
[2, 3]
```

O(n) time, O(1) space. Works because the array is sorted: each move safely discards one value.

2.2 Same direction: a read and a write pointer

```python
def remove_duplicates(nums):     # sorted, in place
    write = 1
    for read in range(1, len(nums)):
        if nums[read] != nums[write - 1]:
            nums[write] = nums[read]
            write += 1
    return write

a = [1, 1, 2, 3, 3, 3, 4]
k = remove_duplicates(a)
print(k, a[:k])
```

Output

```
4 [1, 2, 3, 4]
```

Same shape: move zeros to the end, remove an element, partition by a condition.

2.3 3Sum: sort, fix one, two-pointer the rest

```python
def three_sum(nums):
    nums.sort(); res = []
    for i in range(len(nums) - 2):
        if i and nums[i] == nums[i - 1]:
            continue                # skip duplicates
        lo, hi = i + 1, len(nums) - 1
        while lo < hi:
            s = nums[i] + nums[lo] + nums[hi]
            if s < 0: lo += 1
            elif s > 0: hi -= 1
            else:
                res.append([nums[i], nums[lo], nums[hi]])
                lo += 1
                while lo < hi and nums[lo] == nums[lo - 1]:
                    lo += 1
    return res

print(three_sum([-1, 0, 1, 2, -1, -4]))
```

Output

```
[[-1, -1, 2], [-1, 0, 1]]
```

$O(n^2)$. Container With Most Water and Trapping Rain Water also move the pointer at the smaller side.

2.4 Fixed-size window

```python
def max_sum_k(nums, k):
    window = best = sum(nums[:k])
    for i in range(k, len(nums)):
        window += nums[i] - nums[i - k]   # add new, drop old
        best = max(best, window)
    return best

print(max_sum_k([2, 1, 5, 1, 3, 2], 3))
```

Output

```
9
```

2.5 Variable window: grow right, shrink left

```python
def longest_unique(s):
    last, left, best = {}, 0, 0
    for right, ch in enumerate(s):
        if last.get(ch, -1) >= left:
            left = last[ch] + 1      # jump past the repeat
        last[ch] = right
        best = max(best, right - left + 1)
    return best

print(longest_unique("abcabcbb"), longest_unique("pwwkew"))
```

Output

```
3 3
```

```
# template
left = 0
for right, x in enumerate(items):
    add x to the window
    while the window is invalid:
        remove items[left]; left += 1
    update the answer with right - left + 1
```

2.6 At most K distinct (a counter in the window)

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

print(longest_k_distinct("eceba", 2))
```

Output

```
3
```

Tip: "Exactly K" = atMost(K) − atMost(K−1). Sum-based windows need non-negative numbers; with negatives use prefix sums (3.2).

2.7 Practice

Practice: Two Sum II · Valid Palindrome · Container With Most Water · 3Sum · Trapping Rain Water · Best Time to Buy and Sell Stock · Longest Repeating Character Replacement · Permutation in String · Minimum Window Substring · Sliding Window Maximum (deque)

learn.aiwithpriyabrat.com

3.1 Prefix sums: any range sum in O(1)

```
nums = [3, 1, 4, 1, 5, 9]
pre = [0]
for x in nums:
    pre.append(pre[-1] + x)
def range_sum(i, j):            # inclusive
    return pre[j + 1] - pre[i]
print(pre)
print(range_sum(1, 3), range_sum(0, 5))
```

Output

```
[0, 3, 4, 8, 9, 14, 23]
6 23
```

3.2 Subarray sum = k (works with negatives)

```
def subarray_sum(nums, k):
    seen = Counter({0: 1})       # prefix sum -> count
    total = count = 0
    for x in nums:
        total += x
        count += seen[total - k]  # earlier prefix to cut
        seen[total] += 1
    return count

print(subarray_sum([1, 2, 3, -2, 2], 3))
```

Output

```
4
```

Same idea: longest subarray with sum k (store the first index), divisible by k (store sum % k).

3.3 Hash map: one pass, look back

```
def two_sum(nums, target):
    index = {}
    for i, x in enumerate(nums):
        if target - x in index:
            return [index[target - x], i]
        index[x] = i

def group_anagrams(words):
    groups = defaultdict(list)
    for w in words:
        groups["".join(sorted(w))].append(w)
    return list(groups.values())

print(two_sum([2, 7, 11, 15], 9))
print(group_anagrams(["eat", "tea", "tan", "ate", "nat"]))
```

Output

```
[0, 1]
[['eat', 'tea', 'ate'], ['tan', 'nat']]
```

Longest Consecutive Sequence: put all in a set, start counting only where `x-1` is missing: O(n).

3.4 Binary search: one template, no off-by-one

```
def lower_bound(a, target):   # first index with a[i] >= target
    lo, hi = 0, len(a)
    while lo < hi:
        mid = (lo + hi) // 2
        if a[mid] < target:
            lo = mid + 1
        else:
            hi = mid
    return lo

a = [1, 3, 3, 5, 8]
print(lower_bound(a, 3), lower_bound(a, 4), lower_bound(a, 9))
print(bisect.bisect_left(a, 3), bisect.bisect_right(a, 3))
```

Output

```
1 3 5
1 3
```

Tip: Found if `i < len(a) and a[i] == target`. Upper bound: change `<` to `<=`.

3.5 Search a rotated sorted array

```
def search_rotated(a, target):
    lo, hi = 0, len(a) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if a[mid] == target:
            return mid
        if a[lo] <= a[mid]:              # left half sorted
            if a[lo] <= target < a[mid]: hi = mid - 1
            else: lo = mid + 1
        else:                            # right half sorted
            if a[mid] < target <= a[hi]: lo = mid + 1
            else: hi = mid - 1
    return -1

print(search_rotated([4, 5, 6, 7, 0, 1, 2], 0))
```

Output

```
4
```

3.6 Binary search on the answer

```
def min_eating_speed(piles, hours):   # Koko
    def ok(speed):
        return sum(math.ceil(p / speed) for p in piles) <= hours
    lo, hi = 1, max(piles)
    while lo < hi:
        mid = (lo + hi) // 2
        if ok(mid): hi = mid       # works: try smaller
        else: lo = mid + 1
    return lo

print(min_eating_speed([3, 6, 7, 11], 8))
```

Output

```
4
```

Use it when the question is "smallest / largest X such that a check passes" and the check is monotonic: ship packages in D days, split array largest sum, min capacity.

learn.aiwithpriyabrat.com

4.1 Reverse a linked list (the 3-pointer dance)

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val, self.next = val, next
# build_list([1, 2, 3]) / to_list(head): see 11.8
```

```
def reverse(head):
    prev = None
    while head:
        head.next, prev, head = prev, head, head.next
    return prev

print(to_list(reverse(build_list([1, 2, 3, 4]))))
```

Output

```
[4, 3, 2, 1]
```

4.2 Fast & slow pointers

```
def middle(head):
    slow = fast = head
    while fast and fast.next:
        slow, fast = slow.next, fast.next.next
    return slow.val

def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow, fast = slow.next, fast.next.next
        if slow is fast:
            return True
    return False

head = build_list([1, 2, 3, 4, 5])
print(middle(head), has_cycle(head))
head.next.next.next.next.next = head.next   # 5 -> 2
print(has_cycle(head))
```

Output

```
3 False
True
```

Also: k-th node from the end (move fast k steps first), cycle start (reset one pointer to head), palindrome list (middle + reverse half).

4.3 Dummy node: merge two sorted lists

```
def merge(a, b):
    dummy = tail = ListNode()
    while a and b:
        if a.val <= b.val:
            tail.next, a = a, a.next
        else:
            tail.next, b = b, b.next
        tail = tail.next
    tail.next = a or b
    return dummy.next

print(to_list(merge(build_list([1, 4, 6]), build_list([2, 3, 7]))))
```

Output

```
[1, 2, 3, 4, 6, 7]
```

Tip: A dummy head removes the "is the list empty?" special case. Use it whenever the head may change.

4.4 Stack: matching pairs

```
def valid_brackets(s):
    pairs, stack = {")": "(", "]": "[", "}": "{"}, []
    for ch in s:
        if ch in pairs:
            if not stack or stack.pop() != pairs[ch]:
                return False
        else:
            stack.append(ch)
    return not stack

print(valid_brackets("({[]})"), valid_brackets("("))
```

Output

```
True False
```

4.5 Monotonic stack: next greater element

```
def daily_temperatures(t):
    res, stack = [0] * len(t), []    # stack of indices
    for i, x in enumerate(t):
        while stack and t[stack[-1]] < x:
            j = stack.pop()
            res[j] = i - j           # days until warmer
        stack.append(i)
    return res

print(daily_temperatures([73, 74, 75, 71, 69, 72, 76]))
```

Output

```
[1, 1, 4, 2, 1, 1, 0]
```

Each index is pushed and popped once: O(n). Decreasing stack → next greater; increasing → next smaller.

4.6 Largest rectangle in a histogram

```
def largest_rectangle(h):
    stack, best = [], 0              # increasing heights
    for i, x in enumerate(h + [0]):  # 0 flushes the stack
        start = i
        while stack and stack[-1][1] >= x:
            start, height = stack.pop()
            best = max(best, height * (i - start))
        stack.append((start, x))
    return best

print(largest_rectangle([2, 1, 5, 6, 2, 3]))
```

Output

```
10
```

4.7 Practice

Practice: Reverse Linked List II · Remove Nth Node From End · Reorder List · Linked List Cycle II · Copy List with Random Pointer · Min Stack · Evaluate Reverse Polish Notation · Car Fleet · Next Greater Element II · Maximal Rectangle

learn.aiwithpriyabrat.com

5.1 Merge overlapping intervals

```python
def merge(intervals):
    intervals.sort()                  # by start
    out = [intervals[0]]
    for start, end in intervals[1:]:
        if start <= out[-1][1]:       # overlaps the last one
            out[-1][1] = max(out[-1][1], end)
        else:
            out.append([start, end])
    return out

print(merge([[1, 3], [8, 10], [2, 6], [15, 18]]))
```

Output

```
[[1, 6], [8, 10], [15, 18]]
```

Two intervals overlap when `a.start <= b.end and b.start <= a.end`.

5.2 Meeting rooms: a heap of end times

```python
def min_rooms(meetings):
    ends = []                         # min-heap of end times
    for start, end in sorted(meetings):
        if ends and ends[0] <= start:
            heapq.heapreplace(ends, end)   # reuse a room
        else:
            heapq.heappush(ends, end)
    return len(ends)

print(min_rooms([[0, 30], [5, 10], [15, 20]]))
```

Output

```
2
```

5.3 Fewest removals: sort by END (greedy)

```python
def erase_overlaps(intervals):
    removed, last_end = 0, -math.inf
    for start, end in sorted(intervals, key=lambda x: x[1]):
        if start >= last_end:
            last_end = end            # keep it
        else:
            removed += 1
    return removed

print(erase_overlaps([[1, 2], [2, 3], [3, 4], [1, 3]]))
```

Output

```
1
```

Tip: Merging → sort by start. Keeping the most non-overlapping → sort by end.

5.4 Top K: keep a heap of size k

```python
def top_k_frequent(nums, k):
    count = Counter(nums)
    return heapq.nlargest(k, count, key=count.get)

def kth_largest(nums, k):
    heap = []
    for x in nums:
        heapq.heappush(heap, x)
        if len(heap) > k:
            heapq.heappop(heap)       # drop the smallest
    return heap[0]

print(top_k_frequent([1, 1, 1, 2, 2, 3], 2))
print(kth_largest([3, 2, 1, 5, 6, 4], 2))
```

Output

```
[1, 2]
5
```

O(n log k). K closest points: push `(-dist, point)` so the farthest is popped first.

5.5 K-way merge

```python
def merge_k_sorted(lists):
    heap = [(lst[0], i, 0) for i, lst in enumerate(lists) if lst]
    heapq.heapify(heap); out = []
    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

print(merge_k_sorted([[1, 4, 7], [2, 5], [3, 6, 9]]))
```

Output

```
[1, 2, 3, 4, 5, 6, 7, 9]
```

The index `i` breaks ties, so Python never compares two nodes.

5.6 Two heaps: running median

```python
class Median:
    def __init__(self):
        self.low, self.high = [], []   # max-heap, min-heap
    def add(self, x):
        heapq.heappush(self.low, -x)
        heapq.heappush(self.high, -heapq.heappop(self.low))
        if len(self.high) > len(self.low):
            heapq.heappush(self.low, -heapq.heappop(self.high))
    def median(self):
        if len(self.low) > len(self.high):
            return -self.low[0]
        return (-self.low[0] + self.high[0]) / 2

m = Median()
for x in [5, 15, 1, 3]:
    m.add(x); print(m.median(), end=" ")
```

Output

```
5 10.0 5 4.0
```

5.7 Practice

Practice: Insert Interval · Non-overlapping Intervals · Meeting Rooms I/II · Minimum Number of Arrows · Kth Largest Element in a Stream · K Closest Points to Origin · Task Scheduler · Merge K Sorted Lists · Find Median from Data Stream

learn.aiwithpriyabrat.com

6.1 Tree basics

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val, self.left, self.right = val, left, right
# build_tree([3, 9, 20, None, None, 15, 7]): see 11.8
```

Pre-order	node, left, right	copy / serialize a tree
In-order	left, node, right	BST → sorted order
Post-order	left, right, node	heights, delete, sums
Level order	BFS with a queue	levels, right side view

6.2 DFS: return what the parent needs

```
def max_depth(node):
    if not node:
        return 0
    return 1 + max(max_depth(node.left), max_depth(node.right))

def has_path_sum(node, target):
    if not node:
        return False
    if not node.left and not node.right:
        return node.val == target
    rest = target - node.val
    return (has_path_sum(node.left, rest)
            or has_path_sum(node.right, rest))

root = build_tree([5, 4, 8, 11, None, 13, 4, 7, 2])
print(max_depth(root), has_path_sum(root, 22))
```

Output

```
4 True
```

Tip: Ask: "what does each call return, and what does it need from its children?" That's the whole recursion.

6.3 BFS: level by level

```
def level_order(root):
    if not root:
        return []
    res, q = [], deque([root])
    while q:
        level = []
        for _ in range(len(q)):        # one full level
            node = q.popleft()
            level.append(node.val)
            q.extend(c for c in (node.left, node.right) if c)
        res.append(level)
    return res

print(level_order(build_tree([3, 9, 20, None, None, 15, 7])))
```

Output

```
[[3], [9, 20], [15, 7]]
```

Right side view = last value of each level. Minimum depth = first leaf BFS reaches.

6.4 Validate a BST: pass the allowed range down

```
def is_bst(node, lo=-math.inf, hi=math.inf):
    if not node:
        return True
    if not lo < node.val < hi:
        return False
    return (is_bst(node.left, lo, node.val)
            and is_bst(node.right, node.val, hi))

print(is_bst(build_tree([2, 1, 3])))
print(is_bst(build_tree([5, 1, 4, None, None, 3, 6])))
```

Output

```
True
False
```

Watch out: checking only `left < node < right` is the classic bug: a deeper node can break the rule.

6.5 Lowest common ancestor

```
def lca(node, p, q):
    if not node or node.val in (p, q):
        return node
    left, right = lca(node.left, p, q), lca(node.right, p, q)
    return node if left and right else left or right

root = build_tree([3, 5, 1, 6, 2, 0, 8])
print(lca(root, 6, 2).val, lca(root, 6, 8).val)
```

Output

```
5 3
```

In a BST it's simpler: go left if both are smaller, right if both are bigger, else you're there.

6.6 Trie: prefix lookups in O(word length)

```
class Trie:
    def __init__(self):
        self.root = {}
    def insert(self, word):
        node = self.root
        for ch in word:
            node = node.setdefault(ch, {})
        node["$"] = True                # end of a word
    def _walk(self, s):
        node = self.root
        for ch in s:
            if ch not in node:
                return None
            node = node[ch]
        return node
    def search(self, word):
        node = self._walk(word)
        return bool(node and "$" in node)
    def starts_with(self, prefix):
        return self._walk(prefix) is not None

t = Trie()
for w in ["app", "apple", "bat"]:
    t.insert(w)
print(t.search("app"), t.search("ap"), t.starts_with("ap"))
```

Output

```
True False True
```

6.7 Practice

Practice: Invert Binary Tree · Diameter of Binary Tree · Balanced Binary Tree · Subtree of Another Tree · Binary Tree Right Side View · Kth Smallest in a BST · Construct Tree from Preorder & Inorder · Binary Tree Maximum Path Sum · Serialize/Deserialize · Word Search II (trie)

learn.aiwithpriyabrat.com

7.1 Build the graph, then pick the algorithm

```
edges = [(0, 1), (0, 2), (1, 3), (2, 3)]
graph = defaultdict(list)
for a, b in edges:
    graph[a].append(b)
    graph[b].append(a)        # skip for a directed graph
print(dict(graph))
```

Output

```
{0: [1, 2], 1: [0, 3], 2: [0, 3], 3: [1, 2]}
```

reachability, components, cycles	DFS / BFS / Union-Find
fewest edges (unweighted)	BFS
cheapest path, weights ≥ 0	Dijkstra, O(E log V)
negative weights · at most k stops	Bellman-Ford
order with prerequisites	Topological sort
connect everything cheaply	MST: Kruskal / Prim

A grid is a graph too: neighbours are `(r±1, c)` and `(r, c±1)` inside the bounds.

7.2 DFS on a grid: count islands

```
def num_islands(grid):
    rows, cols = len(grid), len(grid[0])
    def sink(r, c):
        if 0 <= r < rows and 0 <= c < cols and grid[r][c] == "1":
            grid[r][c] = "0"            # mark visited
            for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
                sink(r + dr, c + dc)
    count = 0
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == "1":
                count += 1
                sink(r, c)
    return count

grid = [list("11000"), list("11000"), list("00100"), list("00011")]
print(num_islands(grid))
```

Output

```
3
```

Watch out: deep recursion: Python's default limit is 1000. For big grids, use an explicit stack or BFS.

7.3 BFS: fewest steps (unweighted)

```
def shortest_path(grid, start, goal):
    rows, cols = len(grid), len(grid[0])
    q, seen = deque([(start, 0)]), {start}
    while q:
        (r, c), dist = q.popleft()
        if (r, c) == goal:
            return dist
        for nr, nc in ((r+1, c), (r-1, c), (r, c+1), (r, c-1)):
            if (0 <= nr < rows and 0 <= nc < cols
                    and grid[nr][nc] == 0 and (nr, nc) not in seen):
                seen.add((nr, nc))       # mark when queued
                q.append(((nr, nc), dist + 1))
    return -1

maze = [[0, 0, 0], [1, 1, 0], [0, 0, 0]]
print(shortest_path(maze, (0, 0), (2, 0)))
```

Output

```
6
```

Multi-source BFS (rotting oranges, walls and gates): put every source in the queue at distance 0.

7.4 Topological sort: order with dependencies

```
def course_order(n, prereqs):
    graph, indeg = defaultdict(list), [0] * n
    for course, pre in prereqs:
        graph[pre].append(course)
        indeg[course] += 1
    q = deque(i for i in range(n) if indeg[i] == 0)
    order = []
    while q:
        node = q.popleft()
        order.append(node)
        for nxt in graph[node]:
            indeg[nxt] -= 1
            if indeg[nxt] == 0:
                q.append(nxt)
    return order if len(order) == n else []   # [] = cycle

print(course_order(4, [(1, 0), (2, 0), (3, 1), (3, 2)]))
```

Output

```
[0, 1, 2, 3]
```

7.5 Union-Find: are these connected?

```
parent = list(range(6))
def find(x):
    while parent[x] != x:
        parent[x] = parent[parent[x]]   # path halving
        x = parent[x]
    return x
def union(a, b):
    ra, rb = find(a), find(b)
    if ra == rb:
        return False                    # already joined
    parent[ra] = rb
    return True

for a, b in [(0, 1), (1, 2), (3, 4)]:
    union(a, b)
print(find(0) == find(2), find(0) == find(3))
print(len({find(x) for x in range(6)}), "groups")
```

Output

```
True False
3 groups
```

Redundant connection = the first edge where `union` returns False. Kruskal's MST uses it too.

7.6 Dijkstra: weighted shortest paths

```
def dijkstra(graph, src):
    dist, heap = {src: 0}, [(0, src)]
    while heap:
        d, node = heapq.heappop(heap)
        if d > dist[node]:
            continue                    # stale entry
        for nxt, w in graph[node]:
            if d + w < dist.get(nxt, math.inf):
                dist[nxt] = d + w
                heapq.heappush(heap, (d + w, nxt))
    return dist

g = {"A": [("B", 4), ("C", 1)], "B": [("D", 1)],
     "C": [("B", 2), ("D", 5)], "D": []}
print(dijkstra(g, "A"))
```

Output

```
{'A': 0, 'B': 3, 'C': 1, 'D': 4}
```

Practice: Clone Graph · Pacific Atlantic Water Flow · Rotting Oranges · Course Schedule II · Number of Provinces · Network Delay Time · Word Ladder · Cheapest Flights Within K Stops

learn.aiwithpriyabrat.com

8.1 The backtracking template

```
def backtrack(path, choices):
    if path is a complete answer:
        res.append(path[:])      # copy!
        return
    for choice in choices:
        if not allowed(choice): continue   # prune early
        path.append(choice)      # choose
        backtrack(path, next_choices)      # explore
        path.pop()               # un-choose
```

Draw the decision tree first: its depth and branching give the Big-O (subsets 2^n, permutations n!).

8.2 Subsets

```
def subsets(nums):
    res = []
    def go(i, path):
        res.append(path[:])
        for j in range(i, len(nums)):
            path.append(nums[j])
            go(j + 1, path)
            path.pop()
    go(0, [])
    return res

print(subsets([1, 2, 3]))
```

Output

```
[[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
```

With duplicates: sort, then skip `nums[j] == nums[j-1]` when `j > i`.

8.3 Permutations

```
def permutations(nums):
    res, used = [], [False] * len(nums)
    def go(path):
        if len(path) == len(nums):
            res.append(path[:]); return
        for i, x in enumerate(nums):
            if not used[i]:
                used[i] = True; path.append(x)
                go(path)
                used[i] = False; path.pop()
    go([])
    return res

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

Output

```
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

8.4 Combination sum (reuse allowed)

```
def combination_sum(cands, target):
    res = []
    cands.sort()
    def go(start, path, remaining):
        if remaining == 0:
            res.append(path[:]); return
        for i in range(start, len(cands)):
            if cands[i] > remaining:
                break                  # sorted: prune
            path.append(cands[i])
            go(i, path, remaining - cands[i])   # i, not i+1
            path.pop()
    go(0, [], target)
    return res

print(combination_sum([2, 3, 6, 7], 7))
```

Output

```
[[2, 2, 3], [7]]
```

8.5 N-Queens: sets make the checks O(1)

```
def n_queens(n):
    cols, d1, d2 = set(), set(), set()
    def go(r):
        if r == n:
            return 1
        count = 0
        for c in range(n):
            if c in cols or r - c in d1 or r + c in d2:
                continue
            cols.add(c); d1.add(r - c); d2.add(r + c)
            count += go(r + 1)
            cols.remove(c); d1.remove(r - c); d2.remove(r + c)
        return count
    return go(0)

print([n_queens(n) for n in range(1, 9)])
```

Output

```
[1, 0, 0, 2, 10, 4, 40, 92]
```

8.6 Bit manipulation tricks

```
nums = [4, 1, 2, 1, 2]
single = 0
for x in nums:
    single ^= x            # pairs cancel: a ^ a = 0
print(single)
n = 12                     # 0b1100
print(bin(n), n & (n - 1), n & -n)   # drop / keep lowest 1
print(n & (n - 1) == 0, 16 & 15 == 0)  # power of two?
print(bin(13).count("1"), (13 >> 2) & 1)   # count, bit 2
items = "abc"              # every subset as a bitmask
print(["".join(items[j] for j in range(3) if m >> j & 1)
       for m in range(1 << 3)])
```

Output

```
4
0b1100 8 4
False True
3 1
['', 'a', 'b', 'ab', 'c', 'ac', 'bc', 'abc']
```

`x & 1`	odd?
`x ^ y`	differs / swap / cancel pairs
`1 << k`	2^k, a one-bit mask
`mask | (1 << i)`	add item i to a set

8.7 Practice

Practice: Subsets II · Permutations II · Combination Sum II · Letter Combinations of a Phone Number · Palindrome Partitioning · Word Search · Generate Parentheses · Sudoku Solver · Missing Number · Counting Bits · Reverse Bits · Sum of Two Integers

learn.aiwithpriyabrat.com

9.1 Build any DP in 5 steps

1. State	what does `dp[i]` (or `dp[i][j]`) mean, in words?
2. Transition	how is a state built from smaller ones? (the last choice)
3. Base cases	the smallest inputs you know directly
4. Order	fill so every needed state is ready (or memoize top-down)
5. Answer	which state holds it: `dp[n]`, `max(dp)`, `dp[m][n]`

Tip: Start top-down with `@cache` (easy to get right), then convert to a table if asked to optimise space.

9.2 1D: climbing stairs and house robber

```python
@cache
def climb(n):                  # ways to reach step n
    return n if n <= 2 else climb(n - 1) + climb(n - 2)

def rob(houses):
    take = skip = 0            # best with / without house i
    for x in houses:
        take, skip = skip + x, max(take, skip)
    return max(take, skip)

print(climb(5), climb(30))
print(rob([2, 7, 9, 3, 1]))
```

Output

```
8 1346269
12
```

9.3 Coin change (unbounded choices)

```python
def coin_change(coins, amount):
    dp = [0] + [math.inf] * amount  # fewest coins for each sum
    for s in range(1, amount + 1):
        for c in coins:
            if c <= s:
                dp[s] = min(dp[s], dp[s - c] + 1)
    return dp[amount] if dp[amount] != math.inf else -1

def ways(coins, amount):           # combinations, not orders
    dp = [1] + [0] * amount
    for c in coins:                # coins outer loop!
        for s in range(c, amount + 1):
            dp[s] += dp[s - c]
    return dp[amount]

print(coin_change([1, 2, 5], 11), coin_change([2], 3))
print(ways([1, 2, 5], 5))
```

Output

```
3 -1
4
```

9.4 Longest increasing subsequence

```python
def lis(nums):                 # O(n log n)
    tails = []                 # smallest tail per length
    for x in nums:
        i = bisect.bisect_left(tails, x)
        if i == len(tails):
            tails.append(x)
        else:
            tails[i] = x
    return len(tails)

print(lis([10, 9, 2, 5, 3, 7, 101, 18]))
```

Output

```
4
```

O(n^2) version: `dp[i] = 1 + max(dp[j] for j < i if a[j] < a[i])`.

9.5 2D: longest common subsequence

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

print(lcs("abcde", "ace"), lcs("abc", "def"))
```

Output

```
3 0
```

9.6 Edit distance

```python
def edit_distance(a, b):
    prev = list(range(len(b) + 1))     # one row is enough
    for i in range(1, len(a) + 1):
        cur = [i] + [0] * len(b)
        for j in range(1, len(b) + 1):
            if a[i - 1] == b[j - 1]:
                cur[j] = prev[j - 1]
            else:   # replace, delete, insert
                cur[j] = 1 + min(prev[j - 1], prev[j], cur[j - 1])
        prev = cur
    return prev[-1]

print(edit_distance("horse", "ros"))
print(edit_distance("intention", "execution"))
```

Output

```
3
5
```

9.7 0/1 knapsack

```python
def knapsack(items, cap):      # items: (weight, value)
    dp = [0] * (cap + 1)
    for w, v in items:
        for c in range(cap, w - 1, -1):   # backwards: use once
            dp[c] = max(dp[c], dp[c - w] + v)
    return dp[cap]

print(knapsack([(1, 1), (3, 4), (4, 5), (5, 7)], 7))
```

Output

```
9
```

Forwards loop = each item unlimited (coin change). Partition Equal Subset Sum is knapsack with booleans.

9.8 The DP families

Linear	climbing stairs, house robber, decode ways
Unbounded	coin change, word break, perfect squares
0/1 knapsack	partition subset, target sum
Two sequences	LCS, edit distance, interleaving string
Grid	unique paths, min path sum, maximal square
Intervals	burst balloons, palindromic substrings
States	stock buy/sell with cooldown, fee

learn.aiwithpriyabrat.com

10.1 Greedy: when the local best is safe

```
def can_jump(nums):
    reach = 0
    for i, step in enumerate(nums):
        if i > reach:
            return False
        reach = max(reach, i + step)
    return True

def min_jumps(nums):
    jumps = end = far = 0
    for i in range(len(nums) - 1):
        far = max(far, i + nums[i])
        if i == end:            # must jump now
            jumps, end = jumps + 1, far
    return jumps

print(can_jump([2, 3, 1, 1, 4]), can_jump([3, 2, 1, 0, 4]))
print(min_jumps([2, 3, 1, 1, 4]))
```

Output

```
True False
2
```

Tip: Prove greedy with an exchange argument: swapping any optimal choice for the greedy one is never worse. If you can't, it's probably DP.

10.2 Kadane: maximum subarray

```
def max_subarray(nums):
    best = cur = nums[0]
    for x in nums[1:]:
        cur = max(x, cur + x)    # extend or restart
        best = max(best, cur)
    return best

print(max_subarray([-2, 1, -3, 4, -1, 2, 1, -5, 4]))
```

Output

```
6
```

Max product subarray: track both the max and the min (a negative flips them).

10.3 Gas station: one pass

```
def start_station(gas, cost):
    if sum(gas) < sum(cost):
        return -1
    tank = start = 0
    for i in range(len(gas)):
        tank += gas[i] - cost[i]
        if tank < 0:            # can't start at or before i
            start, tank = i + 1, 0
    return start

print(start_station([1, 2, 3, 4, 5], [3, 4, 5, 1, 2]))
```

Output

```
3
```

10.4 Index as hash: missing & duplicate numbers

```
def first_missing_positive(nums):
    n = len(nums)
    for i in range(n):          # cyclic sort: put x at x-1
        while 0 < nums[i] <= n and nums[nums[i] - 1] != nums[i]:
            j = nums[i] - 1
            nums[i], nums[j] = nums[j], nums[i]
    for i in range(n):
        if nums[i] != i + 1:
            return i + 1
    return n + 1

print(first_missing_positive([3, 4, -1, 1]))
print(sum(range(6)) - sum([3, 0, 1, 5, 4]))   # missing number
```

Output

```
2
2
```

10.5 Matrix moves

```
m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print([list(r) for r in zip(*m[::-1])])   # rotate 90° clockwise
print([list(r) for r in zip(*m)])         # transpose

def spiral(m):
    out = []
    while m:
        out += m.pop(0)                   # top row
        m = [list(r) for r in zip(*m)][::-1]   # rotate left
    return out

print(spiral([[1, 2, 3], [4, 5, 6], [7, 8, 9]]))
```

Output

```
[[7, 4, 1], [8, 5, 2], [9, 6, 3]]
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
[1, 2, 3, 6, 9, 8, 7, 4, 5]
```

10.6 Product of array except self (no division)

```
def product_except_self(nums):
    res, left = [1] * len(nums), 1
    for i in range(len(nums)):
        res[i], left = left, left * nums[i]
    right = 1
    for i in reversed(range(len(nums))):
        res[i] *= right
        right *= nums[i]
    return res

print(product_except_self([1, 2, 3, 4]))
```

Output

```
[24, 12, 8, 6]
```

Prefix from the left × suffix from the right: a pattern that also solves Trapping Rain Water.

10.7 Practice

Practice: Jump Game II · Gas Station · Partition Labels · Hand of Straights · Maximum Product Subarray · Set Matrix Zeroes · Rotate Image · Spiral Matrix · Find the Duplicate Number · Majority Element (Boyer–Moore)

learn.aiwithpriyabrat.com

11.1 Complexity of each pattern

Pattern	Time	Space
Two pointers · sliding window	O(n)	O(1) – O(k)
Prefix sum + hash map	O(n)	O(n)
Binary search · on the answer	O(log n) · O(n log M)	O(1)
Monotonic stack	O(n)	O(n)
Sort + intervals	O(n log n)	O(n)
Heap top-K	O(n log k)	O(k)
Tree DFS / BFS	O(n)	O(h) / O(w)
Graph BFS / DFS · Dijkstra	O(V + E) · O(E log V)	O(V)
Union-Find	≈ O(1) per op	O(n)
Backtracking	$O(2^n)$ – O(n!)	O(n)
DP	states × transition	states (often 1 row)

11.2 Edge cases to test before you say "done"

- ☐ empty input, one element, two elements
- ☐ all equal values, already sorted, reverse sorted
- ☐ negatives, zeros, duplicates, very large numbers
- ☐ no valid answer (return -1 / [] / False?)
- ☐ trees: empty, one node, a single long chain (skewed)
- ☐ graphs: disconnected parts, cycles, self-loops
- ☐ strings: spaces, upper/lower case, unicode

11.3 Mistakes that cost offers

Coding before clarifying	repeat the problem and the constraints back first
Silent thinking	narrate: "I'm trying X because..."
Mutable default `def f(a=[])`	use `None`, then create inside
Appending `path` not `path[:]`	backtracking results all end up empty
`[[0]*n]*m`	all rows are the SAME list; use a comprehension
Marking visited when popped (BFS)	mark when you push, or nodes repeat
Off-by-one in binary search	use one template (3.4) every time
No Big-O at the end	always state time and space

11.4 A 4-week plan (about 1.5 hours a day)

Week 1	arrays, hashing, two pointers, sliding window, prefix sums	20 problems
Week 2	binary search, stacks, linked lists, intervals, heaps	20 problems
Week 3	trees, tries, graphs (BFS, DFS, topo, union-find)	20 problems
Week 4	backtracking, DP, greedy + 2 timed mock interviews	15 problems

Tip: Learn the pattern, not the problem: after solving one, write the signal that gave it away in one line. Re-solve from scratch 3 days later.

11.5 What to say: a 45-minute interview

0–5 min	"Let me confirm: input is..., output is..., n can be up to..."
5–10 min	"Brute force is $O(n^2)$: check every pair. We can do better with..."
10–30 min	code while explaining each block; name variables clearly
30–40 min	"Let me trace [2, 7, 11] ... now the edge case of an empty list"
40–45 min	"Time O(n), space O(n). With sorted input I could drop to O(1) space."

11.6 If you get stuck

- Solve a tiny example by hand and watch what you do: that's the algorithm.
- Ask "what would make this easy?" (sorted? a lookup table? the answer known?)
- Go through the signals table (1.2) one row at a time.
- Say the brute force and optimise one bottleneck at a time.
- Ask for a hint politely. It costs far less than silence.

11.7 The must-do list: one problem per pattern

Pattern	Solve this first
Two pointers	3Sum
Sliding window	Longest Substring Without Repeating Characters
Prefix sum	Subarray Sum Equals K
Hash map	Longest Consecutive Sequence
Binary search	Search in Rotated Sorted Array · Koko Eating Bananas
Linked list	Reorder List
Monotonic stack	Daily Temperatures
Intervals	Merge Intervals
Heap	Top K Frequent Elements
Trees	Lowest Common Ancestor · Validate BST
Trie	Implement Trie
Graphs	Number of Islands · Course Schedule
Backtracking	Combination Sum
DP	Coin Change · Longest Common Subsequence
Greedy	Jump Game

11.8 Helpers used in the examples

```python
def build_list(vals):
    head = None
    for v in reversed(vals):
        head = ListNode(v, head)
    return head

def to_list(node):
    out = []
    while node:
        out.append(node.val)
        node = node.next
    return out

def build_tree(vals):           # level order, None = gap
    if not vals:
        return None
    nodes = [None if v is None else TreeNode(v) for v in vals]
    kids = iter(nodes[1:])
    for node in nodes:
        if node:
            node.left = next(kids, None)
            node.right = next(kids, None)
    return nodes[0]
```

Paste these under the imports (1.5) and the two classes (4.1, 6.1): every example in this sheet then prints exactly what's shown.

learn.aiwithpriyabrat.com

12.1 The one idea behind each pattern

Pattern	Why it works
Two pointers	sorted order tells you which pointer to move, so no pair is checked twice
Sliding window	each element enters and leaves the window once: O(n), not O(n·k)
Prefix sum	sum(i..j) = P[j+1] − P[i]: a range becomes two lookups
Hash map	trade memory for time: "have I seen the complement?" in O(1)
Binary search	works on any yes/no condition that flips once, not only arrays
Fast & slow	different speeds meet inside a cycle; at the end, slow is in the middle
Monotonic stack	pop the elements the new one beats: each is pushed and popped once
Intervals	after sorting by start, only the last merged interval can overlap
Heap	keep only the k best; never sort all n
Trees	decide what each call returns to its parent; the rest follows
Graphs	BFS = fewest steps, DFS = explore everything, topo = order, DSU = groups
Backtracking	choose → explore → un-choose; prune early
DP	same sub-question asked again → store the answer
Greedy	valid only if the local best never blocks a better answer

12.2 Monotonic deque: sliding window maximum

```python
def max_window(nums, k):
    q, out = deque(), []          # indexes, values falling
    for i, x in enumerate(nums):
        while q and nums[q[-1]] <= x:
            q.pop()               # can never be the max again
        q.append(i)
        if q[0] <= i - k:
            q.popleft()           # slid out of the window
        if i >= k - 1:
            out.append(nums[q[0]])
    return out

print(max_window([1, 3, -1, -3, 5, 3, 6, 7], 3))
```

Output

```
[3, 3, 5, 5, 6, 7]
```

O(n) instead of O(n·k): a stack that also drops from the front. Practise: Sliding Window Maximum.

12.3 Difference array: many range updates

```python
def apply_ranges(n, updates):     # (start, end, add)
    diff = [0] * (n + 1)
    for s, e, add in updates:
        diff[s] += add
        diff[e + 1] -= add
    out, run = [], 0
    for d in diff[:-1]:
        run += d
        out.append(run)
    return out

print(apply_ranges(5, [(1, 3, 2), (2, 4, 3), (0, 2, -2)]))
```

Output

```
[-2, 0, 3, 5, 3]
```

The reverse of a prefix sum: each update is O(1), one pass at the end. Solves Car Pooling and Corporate Flight Bookings.

12.4 Quickselect: k-th largest in O(n) average

```python
import random
def kth_largest(nums, k):
    pivot = random.choice(nums)
    big = [x for x in nums if x > pivot]
    mid = [x for x in nums if x == pivot]
    small = [x for x in nums if x < pivot]
    if k <= len(big):
        return kth_largest(big, k)
    if k <= len(big) + len(mid):
        return pivot
    return kth_largest(small, k - len(big) - len(mid))

print(kth_largest([3, 2, 1, 5, 6, 4], 2),
      kth_largest([3, 2, 3, 1, 2, 4, 5, 5, 6], 4))
```

Output

```
5 4
```

Heap = O(n log k) and works on streams; quickselect = O(n) average, $O(n^2)$ worst case.

12.5 Fenwick tree: prefix sums that change

```python
class Fenwick:
    def __init__(self, n):
        self.t = [0] * (n + 1)
    def add(self, i, delta):      # i is 1-based
        while i < len(self.t):
            self.t[i] += delta
            i += i & -i
    def total(self, i):           # sum of 1..i
        s = 0
        while i:
            s += self.t[i]
            i -= i & -i
        return s

f = Fenwick(5)
for i, v in enumerate([5, 2, 9, 1, 7], 1):
    f.add(i, v)
print(f.total(3), f.total(5) - f.total(1))
f.add(2, 10); print(f.total(3))
```

Output

```
16 19
26
```

Update and query both O(log n). Use it when values change between range-sum queries (a plain prefix sum would need O(n) rebuilds). Practise: Range Sum Query – Mutable.

12.6 Problems that combine two patterns

Problem	Patterns
Longest substring with k distinct	sliding window + hash map
Top K frequent words	hash map (count) + heap
Koko eating bananas, split array	binary search on the answer + greedy check
Word search II	trie + backtracking on a grid
Shortest path visiting all nodes	BFS + bitmask state
Network delay time	graph + heap (Dijkstra)
Longest path in a matrix	DFS + memoization (DP on a grid)
LRU cache	hash map + doubly linked list

12.7 DP or greedy? Recursion or iteration?

A counterexample breaks the greedy choice	DP
"Count the ways" or "min/max over choices"	DP
Sort, then one pass with a provable rule	greedy
Recursion depth could pass ~1,000	iterate, or a stack (Python's limit)
Few states actually reachable	top-down with `@cache`
Every state needed, memory tight	bottom-up, keep one row

learn.aiwithpriyabrat.com