BRP DEVELOPERS / FREE LEARNING LIBRARY

DSA Patterns Starter

Beginner-to-intermediate edition ? BRP developers free companion guide

DSA & problem solving | Beginner to intermediate

Original educational companion guide. No affiliation with any interview platform or certification provider.

A problem-solving guide for turning a prompt into a tested algorithm. The examples are original and use standard Python; they are not copied from any challenge platform. Work through the invariant before memorizing a pattern.

Start with constraints and a contract

Write down what the input represents, what must be returned, whether order matters, and whether inputs may contain duplicates or negative values. Estimate the input size. A nested scan is often O(n2); sorting is O(n log n); a single pass with a hash map is expected O(n). The right choice depends on the constraints and the required output.

Keep time and space separate. A set of all distinct values uses O(k) extra space for k distinct items. Sorting in place may use different auxiliary memory than building a new output. Explain any average-case assumption, especially for hashing.

Hash maps: remember what you have seen

A dictionary is useful when a value can be summarized by a key. To find whether an earlier number complements the current number to reach a target, store the values already visited:

```
def pair_indices(values, target):
    earlier = {}
    for index, value in enumerate(values):
        needed = target - value
        if needed in earlier:
            return earlier[needed], index
        earlier[value] = index
    return None
```

The invariant is: before processing index `i`, `earlier` contains exactly the values at earlier indices. Checking before inserting prevents one item from pairing with itself. Repeated values work because the second occurrence can find the first. Expected time is O(n), extra space O(n).

For frequency questions, use `collections.Counter`. For grouping, choose a normalized key such as a sorted character tuple only when the alphabet and input size make that cost acceptable. The key should represent the property that defines a group.

Two pointers and sliding windows

Two pointers are useful when sorted order or a contiguous interval supplies structure. For a sorted list, move the left pointer up when the sum is too small and the right pointer down when it is too large. Each pointer moves at most n times, so the scan is O(n).

A sliding window maintains a contiguous range. For nonnegative values, the sum can be updated as the window expands and contracts:

```
def shortest_at_least(target, values):
    left = 0
    total = 0
    best = len(values) + 1
    for right, value in enumerate(values):
        total += value
        while total >= target:
            best = min(best, right - left + 1)
            total -= values[left]
            left += 1
    return 0 if best > len(values) else best
```

This algorithm assumes values are nonnegative. With negative numbers, removing the left value does not move the sum monotonically, so the `while` condition no longer justifies discarding the prefix. This is a common interview trap: state the assumption or select a different technique.

Stacks: defer decisions until you know enough

A stack is useful for nested structure and ?next greater/smaller? questions. In a monotonic stack, keep indices whose values remain candidates. When a new value resolves older candidates, pop them and record the answer. Each index is pushed and popped at most once, giving O(n) time.

For matching brackets, push opening symbols and require each closing symbol to match the top. Reject a closing symbol when the stack is empty and reject leftover openings at the end. This directly encodes the nesting invariant.

Binary search: define the boundary

Binary search is clearest when the goal is a boundary, not merely ?find something.? The following returns the first index whose value is at least `target`:

```
def lower_bound(values, target):
    left, right = 0, len(values)
    while left < right:
        middle = left + (right - left) // 2
        if values[middle] < target:
            left = middle + 1
        else:
            right = middle
    return left
```

The search interval is half-open: `[left, right)`. At every iteration, values before `left` are known to be too small; values at or after `right` are known to be large enough. The interval shrinks until both boundaries meet. The result can equal `len(values)` when no element qualifies.

Breadth-first search: layers mean distance

For an unweighted graph, breadth-first search visits vertices in nondecreasing number of edges from the start. Mark a vertex when enqueuing it, not when dequeuing it, to avoid duplicate queue entries:

```
from collections import deque

def distances(graph, start):
    distance = {start: 0}
    queue = deque([start])
    while queue:
        node = queue.popleft()
        for neighbor in graph[node]:
            if neighbor not in distance:
                distance[neighbor] = distance[node] + 1
                queue.append(neighbor)
    return distance
```

With adjacency lists, traversal is O(V + E). Unreachable vertices are absent from the returned mapping. For weighted edges, ordinary BFS is not sufficient unless all edge weights are equal; use an appropriate shortest-path algorithm.

Practice and review

Try these without looking at the answers first:

1. Return the first repeated value in a sequence. Define whether ?first? means earliest second occurrence or smallest index of the original occurrence.
2. Given sorted values, return the insertion position for a target. Use the lower-bound invariant and test empty input.
3. Explain why the shortest-subarray sliding window above fails when negative values are allowed.
4. Find an unweighted shortest path and reconstruct the route, not only its distance. Store each discovered vertex's parent.

A strong solution includes a clear contract, invariant, complexity, and tests for empty input, duplicates, and boundaries. Patterns are tools; assumptions decide when they are valid.

More free guides: brpdevs.online/learn/