BRP DEVELOPERS / FREE LEARNING LIBRARY

DSA Advanced Problem-Solving Workbook

Advanced edition ? BRP developers free companion guide

DSA & problem solving | Advanced

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

This workbook develops graph, heap, dynamic-programming, greedy, and backtracking reasoning. Each method is described by its state and invariant. Examples are original teaching material; always confirm the problem's constraints before applying a pattern.

Graphs: traversal, cycles, and ordering

Represent a graph with adjacency lists when the number of edges is much smaller than V2. Breadth-first search finds minimum edge counts in an unweighted graph. Depth-first search explores reachability and structure. For directed cycle detection, a DFS can use three states: unseen, active on the current path, and complete. An edge to an active vertex is a back edge and reveals a cycle.

Topological sorting applies only to a directed acyclic graph. Kahn's algorithm counts incoming edges, queues all zero-indegree nodes, and removes their outgoing edges. If fewer than V nodes are emitted, the remaining graph contains a cycle. The runtime is O(V + E).

```
from collections import deque

def topological_order(graph):
    nodes = set(graph)
    for neighbors in graph.values():
        nodes.update(neighbors)
    adjacency = {node: list(graph.get(node, ())) for node in nodes}
    indegree = {node: 0 for node in nodes}
    for neighbors in adjacency.values():
        for neighbor in neighbors:
            indegree[neighbor] += 1
    ready = deque(node for node, degree in indegree.items() if degree == 0)
    order = []
    while ready:
        node = ready.popleft()
        order.append(node)
        for neighbor in adjacency[node]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                ready.append(neighbor)
    if len(order) != len(indegree):
        raise ValueError("graph contains a cycle")
    return order
```

The implementation builds a separate adjacency mapping, so it leaves the caller's graph unchanged. Copying has a cost proportional to V + E; when ownership is clear and mutation is acceptable, an in-place normalization may be a reasonable alternative. State that trade-off in a review.

Heaps: keep only the candidates you need

A heap exposes the smallest item efficiently. Use it for repeated minimum selection, merging sorted streams, or retaining the best k values. Python's `heapq` is a min-heap. To retain the largest k values, keep a min-heap of size k; once full, discard candidates no larger than the smallest retained value.

For merging k sorted lists containing N total values, a heap with one current item per list gives O(N log k) time and O(k) auxiliary space. Include a tie-breaker in heap tuples when payload objects are not orderable:

```
heap_item = (priority, sequence_number, payload)
```

A monotonically increasing sequence number resolves equal priorities without comparing arbitrary payloads.

Dynamic programming: define the state before the recurrence

Dynamic programming is useful when subproblems overlap and a solution can be composed from smaller states. State should contain enough information to decide future transitions. Write down: what `dp[i]` means, the base case, the transition, the iteration order, and the final answer.

For the number of ways to reach step n by taking one or two steps, `dp[i]` is the number of routes to i. The recurrence is `dp[i] = dp[i - 1] + dp[i - 2]`, with `dp[0] = 1` and `dp[1] = 1`. The one-dimensional array can be reduced to two rolling values because each state uses only the previous two. The time is O(n); rolling storage is O(1).

For a grid with obstacles, `dp[r][c]` can mean paths to cell `(r, c)`. An obstacle has zero paths; otherwise add paths from valid top and left neighbors. Be explicit about whether the start is blocked and whether movement is only right/down. Most DP errors are mismatched state definitions or base cases, not syntax.

Greedy algorithms require an exchange argument

A locally attractive choice is not automatically globally optimal. For interval scheduling with the goal of selecting the maximum number of nonoverlapping intervals, sort by end time and repeatedly select the compatible interval that finishes earliest.

Why does it work? Consider an optimal solution. Its first interval can be replaced by the compatible interval with the earliest finish; the replacement leaves at least as much room for all later intervals. Repeating this argument establishes an optimal solution that follows the greedy choice. Sorting costs O(n log n); the scan is O(n).

State the interval convention: if intervals are half-open, `[start, end)`, an interval starting exactly when the prior ends is compatible. Different endpoint semantics change the comparison.

Backtracking: make the state reversible

Backtracking explores partial candidates, applies a constraint, and undoes a choice. A common template is: choose, recurse, unchoose. The invariant is that the current path represents exactly the decisions on the recursion stack.

For combinations, only choose later indices so a set is generated once. Prune when the remaining number of candidates cannot fill the requested size. For constraint-heavy problems, prune as soon as a partial assignment violates a necessary condition. Explain why a pruning rule is safe; an overly aggressive rule can discard valid solutions.

If recursion depth can grow with input size, account for Python's recursion limit and consider an explicit stack. If all solutions are requested, output size itself can be exponential; no algorithm can print exponentially many results in polynomial total time.

Disjoint sets: connectivity under merging

A disjoint-set union structure supports connectivity queries while components are merged. Store a parent for each element and a component size or rank. Path compression shortens future find operations; union by size attaches the smaller tree below the larger. Their amortized cost is effectively constant for practical input sizes, more precisely O(?(n)) per operation.

A useful invariant is that every node reaches exactly one component root. `union(a, b)` returns whether it joined two previously separate components. That return value supports cycle detection in an undirected edge

stream.

Advanced review questions

1. Why is BFS not the right shortest-path method when edge weights differ? Because the first discovered route need not have minimum total weight; use Dijkstra for nonnegative weights or another method matching the weight model.
2. When can greedy interval selection fail? When the objective or compatibility rule differs. For example, maximizing covered duration is not the same objective as maximizing interval count; the earliest-finish proof does not establish it.
3. What makes a DP state sufficient? Given the state and a candidate transition, no discarded history changes the legal future choices or their value.
4. Why add a sequence number to heap entries? Equal priorities otherwise cause tuple comparison to reach payload objects, which may not support ordering.

For every advanced solution, name the state, invariant, proof idea, complexity, and the constraint that would require a different technique.

More free guides: brpdevs.online/learn/