👘

Dynamic Programming

📘 Dynamic Programming Exam Notes

What is Dynamic Programming (DP)?

- **DP is a problem-solving technique**, not a single algorithm.
- It applies when two properties hold:
  1. **Optimal Substructure –** the solution to a problem can be built from solutions to its subproblems.
  2. **Overlapping Subproblems –** the same subproblem appears multiple times during recursion.

Key Concepts

Subproblem

- A smaller instance of the original problem.
- Solving all needed subproblems gives the full solution.

Optimal Substructure

- The optimal solution can be constructed from optimal solutions to subproblems.
- Example: Shortest path = shortest subpath + one edge.

Overlapping Subproblems

- When the same subproblem is required in multiple branches of recursion.
- Without memoization, recursion may recompute it many times.

Why Counting Leaf-Paths in a Tree is **Not DP**

- Problem: “For each vertex, how many paths to a leaf?”
- Base case: leaf → 1 path.
- Recurrence: vertex = sum of children’s paths.
- **Has optimal substructure** ✅
- **No overlapping subproblems** ❌ (each subtree is unique).
- ∴ Not DP, just recursionlecture09_dp_intro(1).

Tree vs DAG in DP

- **Tree recursion**: Each subproblem is unique, no reuse.
- **DAG recursion**: Different paths in recursion graph hit the same subproblem. Memoization reuses results.
- Any DP can be seen as solving problems on a **DAG of states**.

Why Counting Leaf-Paths in a DAG **Is DP**

- Same recurrence as tree case.
- But now multiple vertices can share children (subproblems).
- ∴ Overlapping subproblems exist.
- Need memoization or DP table to avoid recomputationlecture09_dp_intro(1).

Pseudocode (conceptual, bottom-up)

```
for node in reverse_toposort(DAG):
    if node is leaf:
        dp[node] = 1
    else:
        dp[node] = sum(dp[child] for child in children(node))
```

Top-Down vs Bottom-Up

- **Top-Down (recursion + memoization)**
  - Start at root/target state.
  - Only explore needed subproblems.
  - Naturally finds valid evaluation order (DFS).
- **Bottom-Up (tabulation)**
  - Start from base cases.
  - Requires known order (e.g., via topological sort).
- Both give the same result; choice is implementation preference.

Longest Path in a DAG

- Base case: leaf → length 0.
- Recurrence:

  ```
  longest(u) = 0              if u is leaf
  longest(u) = max(longest(v) + 1 for v in children(u)) otherwise
  ```

- Works for weighted DAGs by adding edge weights instead of `+1`.

Pseudocode (bottom-up)

```
for u in reverse_toposort(DAG):
    if u has no children:
        dp[u] = 0
    else:
        dp[u] = max(dp[v] + 1 for v in children(u))
```

All DP Problems are DAGs

- DP relies on **optimal substructure + overlapping subproblems**.
- The recursion graph can always be collapsed into a **DAG of states**.
- Vertices = states; edges = dependencies.

Coin Change Example

- Problem: Given coins `C`, find min coins to make amount `X`.
- State: how much is left to pay.
- Base case: `f(0) = 0`.
- Recurrence:

  ```
  f(x) = min( f(x - c) + 1 for c in C if c <= x )
  ```

- DAG property: state `x` strictly decreases → no cycles.

Complexity

- `O(X * |C|)` time (each state checks all coins).
- `O(X)` memory.
- Without DP, naive recursion could be exponential.

✅ Quick Rules for Identifying DP

1. **Is the problem recursive?** (Can be broken into smaller subproblems?)
2. **Does it have optimal substructure?** (Best solution = combination of smaller bests?)
3. **Are subproblems overlapping?** (Do we solve the same ones multiple times?)
   - If yes → use memoization/tabulation.