☎️

DP technique

Dynamic Programming — Exam Notes

1. What is Dynamic Programming?

- A **technique** for solving problems with:
  - **Optimal substructure**: the solution to a problem can be built from solutions to its subproblems.
  - **Overlapping subproblems**: the same subproblem occurs many times in recursion.
- Always corresponds to solving a **DAG of states**:
  - Each state = a subproblem.
  - Edges = dependencies.

2. Modeling DP Problems

When you see a problem:

1. **Identify the state(s):** What parameters uniquely describe a subproblem?
2. **Base cases:** When does the answer become trivial (0, 1, or empty)?
3. **Recurrence:** Express state `(...)` in terms of smaller states.
4. **Evaluation order:** Either top-down (recursion + memoization) or bottom-up (fill table).
5. **Traceback:** If you need the actual solution, not just its value, store choices and retrace.

3. Longest Common Subsequence (LCS)

Problem: Find the longest subsequence common to both strings.

- **State:** `(i, j)` = LCS length of first `i` chars of X and first `j` chars of Y.
- **Base case:** If `i=0` or `j=0` → `0`.
- **Recurence:**
  - If `X[i-1] == Y[j-1]` :$f(i,j)=f(i-1,j-1)+1$

    f(i,j)=f(i−1,j−1)+1f(i,j) = f(i-1,j-1) + 1
  - Else:$f(i,j)=\max(f(i-1,j),f(i,j-1))$

    f(i,j)=max(f(i−1,j),f(i,j−1))f(i,j) = \max(f(i-1,j), f(i,j-1))
- **Table:** 2D grid, rows = X, cols = Y, filled left-to-right.
- **Complexity:** O(| X | · | Y |)O(|X|\cdot|Y|)O(| X | · | Y |) time and memory.

Traceback for LCS

- Store parent pointers or retrace decisions.
- Diagonal move → characters matched (add to LCS).
- Up/left move → skip one character.
- Multiple optimal paths → multiple valid LCS's.

---

4. 0/1 Knapsack

Problem: Maximize total value within capacity `W`, given `n` items with weights/values.

- **State:** `(i, w)` = max value using first `i` items with capacity `w`.
- **Base cases:** `f(0, w) = 0`, `f(i, 0) = 0`.
- **Recurrence:**
  - If `w_i > w`:f(i,w)=f(i−1,w)

    f(i,w)=f(i−1,w)f(i, w) = f(i-1, w)
  - Else:f(i,w)=max(f(i−1,w),f(i−1,w−wi)+vi)

    f(i,w)=max(f(i−1,w),f(i−1,w−wi)+vi)f(i, w) = \max(f(i-1, w), f(i-1, w-w_i) + v_i)
- **Table:** Rows = items, cols = capacities.
- **Complexity:** O(nW)O(nW)O(nW) time, O(nW)O(nW)O(nW) memory.

Traceback for Knapsack

- Start bottom-right `(n, W)`.
- If value came from `f(i-1, w)` → item not taken.
- If value came from `f(i-1, w-w_i)+v_i` → item taken, reduce `w`.
- Repeat until `i=0`.

---

5. Memory Optimization

- Many DP tables only need the **previous row** → reduce memory.
- Example (Knapsack):
  - Instead of `dp[i][w]`, store only `dp[w]` (1D).
  - Iterate capacities **backwards** so you don't overwrite values needed later.
- Complexity:
  - Time: unchanged.
  - Space: reduced from `O(nW)` → `O(W)`.
- Trade-off: traceback becomes difficult because intermediate rows are discarded.

---

6. General DP Notes

- **Top-Down vs Bottom-Up**: both give same result.
  - Top-down = recursion + memoization (lazy, only needed states computed).

- Bottom-up = table (iterative, predictable order).
- **Traceback**: general technique to reconstruct solutions.
- **All DP problems**: can be visualized as solving a DAG of states.
- **Complexity pattern**:
  - Time = (#states) × (time per recurrence).
  - Space = (#states stored).

---

7. Example Complexity Summaries

- **LCS**: O(mn)O(mn)O(mn) time, O(mn)O(mn)O(mn) memory (can be optimized to O(min(m,n))O(\min(m,n))O(min(m,n)) for length only).
- **Knapsack**: O(nW)O(nW)O(nW) time, O(nW)O(nW)O(nW) memory (optimized to O(W)O(W)O(W)).
- **Coin Change**: O(X ∣ C ∣)O(X|C|)O(X ∣ C ∣) where `x` = amount, `c` = coins.

Item	Value viv_ivi	Weight wiw_iwi
1	60	2
2	100	3
3	120	4

Final DP Table

i\w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	60	60	60	60
2	0	0	60	100	100	160
3	0	0	60	100	120	160