

Dynamic Programming Patterns & State Transition Equations

0/1 & Unbounded Knapsack, LCS/Edit Distance, Interval DP, Bitmask DP, and space-reduction templates.

EXECUTIVE SUMMARY

Systematic taxonomy of Dynamic Programming (DP) archetypes with recurrence relations, iteration-order invariants, and rolling-array space optimizations from $O(N*W)$ down to $O(W)$.

KEY EQUATIONS & INVARIANTS

* 0/1 Knapsack Recurrence (1D Space-Optimized):

LaTeX:
$$dp[w] = \max(dp[w], dp[w - \text{weight}_i] + \text{val}_i)$$

$$\text{for } w = W \text{ down to } \text{weight}_i$$

Note: Iterating capacity w backwards prevents using the same item multiple times in the same step.

* Levenshtein Edit Distance:

LaTeX:
$$dp[i][j] = \begin{cases} dp[i-1][j-1] & \text{if } s_1[i] = s_2[j] \\ \min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) & \text{otherwise} \end{cases}$$

Note: Models minimum insertions, deletions, and substitutions to transform s_1 into s_2 .

1. Five Core DP Archetypes & Iteration Orders

Every DP formulation requires three components: (1) minimal state variables that satisfy the Markov property, (2) base cases, and (3) a topological evaluation order over the state DAG.

Archetype | State Definition | Loop Direction | Classic Problems

0/1 Knapsack | $dp[w]$ = max value with capacity w | Outer: items; Inner: w from W down to $wt[i]$ | Subset Sum, Partition Equal Subset
 Unbounded Knapsack | $dp[w]$ = min/max ways for capacity w | Outer: items; Inner: w from $wt[i]$ up to W | Coin Change, Rod Cutting
 Two-Sequence Alignment | $dp[i][j]$ on prefixes $A[:i]$, $B[:j]$ | i from $1..m$, j from $1..n$ (rolling 2 rows) | LCS, Edit Distance, Regex Match
 Interval DP | $dp[i][j]$ on subarray $s[i..j]$ | Length len from $2..n$; left i from $0..n-len$ | Matrix Chain Multiplication, Burst Balloons
 Bitmask DP | $dp[mask][u]$ over visited subset $mask$ | $mask$ from $0..(1<n)-1$ | Traveling Salesperson (TSP) in $O(n^2 2^n)$

[PYTHON] 1D Space-Optimized 0/1 Knapsack vs Unbounded Coin Change

```
def knapsack_01(weights: list[int], values: list[int], capacity: int) -> int:
    dp = [0] * (capacity + 1)
    for wt, val in zip(weights, values):
        for w in range(capacity, wt - 1, -1): # Reverse prevents reuse
            dp[w] = max(dp[w], dp[w - wt] + val)
    return dp[capacity]

def min_coins_unbounded(coins: list[int], amount: int) -> int:
    INF = amount + 1
    dp = [0] + [INF] * amount
    for c in coins:
        for a in range(c, amount + 1): # Forward allows unlimited reuse
            dp[a] = min(dp[a], dp[a - c] + 1)
    return dp[amount] if dp[amount] < INF else -1
```

COMMON PRODUCTION & INTERVIEW PITFALLS

- Iterating capacity forward in 0/1 Knapsack silently converts it into Unbounded Knapsack.
- Using `@functools.lru_cache(None)` on deep recursive DP trees (>1000 frames) hits Python's `RecursionError` unless converted to bottom-up tabulation.

FREQUENTLY ASKED QUESTIONS

Q: When is top-down memoization faster than bottom-up tabulation?

A: When the state space is sparse and only a small fraction of all possible states are reachable from the initial query, top-down memoization skips unreachable states entirely.