

Dynamic Programming State Patterns

Dynamic programming (DP) problems are solved by defining **states** that capture subproblems. Each state is typically one or more indices (or bitmasks) that summarize the portion of the problem solved so far. We use an array or table `dp[...]` where each entry corresponds to a state. Common state patterns include one-dimensional (`dp[i]`), two-dimensional (`dp[i][j]`), interval/range (`dp[l][r]`), range with turn (`dp[l][r][player]`), and bitmask states (`dp[mask]` or `dp[mask][i]`). Below we explain each pattern in detail, what the notation means, and where it's used. We also define key terms like *minimum/maximum cost*, *number of ways*, *subarray*, and *bitmask*. Example problems illustrate each pattern.

1D DP (`dp[i]`) – Prefix or linear states

A **1D DP** state `dp[i]` usually refers to solving the problem for a prefix of length `i` (or “up to index `i`”). For example, `dp[i]` might be the optimal (min/max) value or the number of ways to achieve something using the first `i` elements or up to position `i`. Common uses:

- **Fibonacci/LIS/Max-sum problems**: e.g. `dp[i]` = best value (length/cost/sum) ending at `i`.
- **Coin change or “ways to sum”**: e.g. `dp[i]` = number of ways to form sum `i`. Here `dp[0]=1` as base.
- **Path counting on a line**: ways to reach step `i` from the start.
- **Knapsack with single dimension** (when one dimension is fixed or implicit).

The value stored can represent different things: **minimum/maximum cost** to reach state `i`, or **number of ways** to reach it. For instance, in the “ways to form sum” problem, `dp[i]` counts the ways (a natural number). In a *minimization* problem, `dp[i]` might store the minimum cost to reach `i` (we'd initialize with ∞ for unreachable states). A simple recurrence often looks like `dp[i] = f(dp[i-1], dp[i-2], ...)`, depending on subproblem transitions.

Example: Coin Change (min coins) – Let `dp[x]` = minimum coins to make amount `x`. Transition: `dp[x] = min(dp[x - coin[j]] + 1)` over coin types. Or **LIS length** – `dp[i]` = length of longest increasing subsequence ending at `i`, computed by checking all `j < i` with `a[j] < a[i]`. See GeeksforGeeks: “define `dp[i]` = number of ways to form sum `i`”.

2D DP (`dp[i][j]`) – Two parameters

A **2D DP** state `dp[i][j]` has two indices. Often these mean “the answer considering the first `i` items/positions and with parameter `j`.” Typical uses include:

- **Knapsack or subset problems**: `dp[i][j]` can mean whether we can achieve sum `j` using the first `i` elements (subset-sum) or the max value with capacity `j` using first `i` items. For example, in the subset-sum problem, “we create a 2D array `dp` of size $(n+1)(sum+1)$ where `dp[i][j]` is true if some subset of elements `0..i` has sum `j`”. The recurrence is `dp[i][j] = dp[i-1][j] OR dp[i-1][j - arr[i-1]]` (if `arr[i-1] ≤ j`).
- **LCS/Edit Distance/Matrix path**: `dp[i][j]` = length or cost for prefixes of two sequences of lengths `i` and `j`. For example, `dp[i][j]` might be the minimum edit distance between the first `i` chars of string `A` and first `j`

of string B, or the number of palindromic subsequences in A[i..j].

- Grid DP (path in 2D grid): $dp[i][j]$ = best cost or ways to reach cell (i, j) from the origin, using neighbors (up, left, etc.). For a grid, $dp[i][j] = \min(dp[i-1][j], dp[i][j-1]) + \text{cost}(i, j)$.

- "Min cost path" problems: $dp[i][j]$ = minimum cost to reach cell (i, j) , given movement constraints ².

Again, the stored value can be a cost or count. For example, in **Subset Sum** ², $dp[i][j]$ is a boolean or count of ways. In **Knapsack (max value)**, $dp[i][j]$ is a numeric max value for weight j . In **Matrix Chain Multiplication** or **LPS** (longest palindromic subsequence), $dp[i][j]$ is the optimal value for subarray/substring $[i..j]$. For instance, in longest palindromic subsequence, "element" $dp[i][j]$ stores the length of LPS of substring $s[i]$ to $s[j]$ ³. That is a classic 2D DP on a range.

Example: Subset-Sum - Let $dp[i][j] = \text{true/false}$ whether sum j is achievable from first i elements ². Or LCS - $dp[i][j]$ = length of LCS of $A[1..i]$ and $B[1..j]$. Each transition depends on one or two smaller states.

Interval (Range) DP ($dp[l][r]$) - Subarrays/subsequences

An **interval DP** state $dp[l][r]$ deals with a contiguous subarray or substring from index l to r . Here the DP table is 2D, but conceptually it represents the subproblem on the interval $[l..r]$. Common applications:

- **Palindromes and brackets:** $dp[l][r]$ = best answer for substring $s[l..r]$. For example, in **palindromic subsequence** we saw above, $dp[l][r]$ stores LPS length ³. In **palindromic partitioning**, $dp[l][r]$ might store the minimum cuts to partition $s[l..r]$ into palindromes.

- **Matrix chain multiplication:** $dp[l][r]$ = minimum cost to multiply matrices from l to r . We compute by splitting at some k with $dp[l][r] = \min(dp[l][k] + dp[k+1][r] + \text{cost})$ (combine two subarrays).

- **"Merging stones" or subarray merging:** Similar to matrix chain.

- **Game intervals:** If players take turns removing elements from ends of an array segment, $dp[l][r]$ can record the outcome/value for that interval (see next section).

- **General "range DP" problems:** Many problems with optimal substructure on ranges use $dp[l][r]$.

Key term: *subarray* means a contiguous slice of an array. For example, $A[l..r] = [A[l], A[l+1], \dots, A[r]]$.

The DP transitions usually consider all ways to split or handle that interval. For example, the LPS tabulation approach says "we create a 2D array $dp[i][j]$ where $dp[i][j]$ stores the length of LPS of substring $s[i..j]$ " ³. One computes $dp[i][j]$ using smaller intervals: e.g. if $s[i] == s[j]$, then $dp[i][j] = dp[i+1][j-1] + 2$, else $dp[i][j] = \max(dp[i+1][j], dp[i][j-1])$. Similarly, matrix chain DP splits at a middle k : $dp[l][r] = \min(dp[l][k] + dp[k+1][r] + \text{cost})$.

Example: Minimum insertions to palindrome: $dp[l][r]$ = min ops to make $s[l..r]$ a palindrome. We choose: if $s[l] == s[r]$, then $dp[l][r] = dp[l+1][r-1]$; else $dp[l][r] = 1 + \min(dp[l+1][r], dp[l][r-1])$. Another example from game theory (next): in some coin-picking games $dp[l][r]$ holds the best achievable score difference on interval $[l..r]$.

Interval DP with Player/Turn ($dp[l][r][player]$) – Game DP

In two-player *game* problems (players alternate moves on an interval), we often add a third dimension for the **current player** or turn. A typical state is $dp[l][r][p]$, where p indicates whose turn it is (e.g. 0 or 1). Then $dp[l][r][p]$ might represent the best score or win/lose status for player p on subarray $[l..r]$. The extra dimension disambiguates which player is about to move, since the optimal choice depends on that.

For example, suppose players alternately remove stones from either end of an array. We might define $dp[l][r][0]$ = maximum score difference (Player 0 minus Player 1) if it's player 0's turn on $[l..r]$, and $dp[l][r][1]$ similarly for player 1's turn. The recurrence then chooses the best move for the current player (minimax). A simpler binary outcome version defines $DP[L][R] = 1$ if the subarray $[L..R]$ is a winning state (first player to move on that interval wins) and 0 if losing. GeeksforGeeks explains a similar idea: "Let $DP[L][R]$ denote whether the subarray from L to R is a Losing state or a Winning state. $DP[L][R] = 0$ means $[L,R]$ is a losing subarray, $DP[L][R] = 1$ means it is winning." ⁴. One then uses smaller intervals $[L+1, R]$ or $[L, R-1]$ to decide the current state. In practice, implementing $dp[l][r][player]$ makes the logic explicit: you try both end-moves for the current player and switch $player$.

Example: Removal Game (CSES) – Two players pick from ends of a list of coins trying to maximize their sum. You can implement it with $dp[l][r]$ = best score difference for the current player on $[l..r]$, or as $dp[l][r][0/1]$. GeeksforGeeks' game example (players building increasing sequence from ends) also uses $DP[L][R] = 0/1$ for losing/winning ⁴.

Bitmask DP ($dp[mask]$, $dp[mask][i]$) – Subset states

A **bitmask DP** uses an integer bitmask to represent a subset of items. Here states like $dp[mask]$ or $dp[mask][i]$ appear. The mask has one bit per element (up to ~20 items), and bit=1 means that element is included/visited. Bitmask DP is common when we want to remember exactly which subset we have used. Typical pattern:

- **TSP/Hamiltonian path:** Let $dp[mask][i]$ = best (min-cost or max-value) path that visits all items in $mask$ and ends at item i . For example, in the Travelling Salesman Problem, $dp[mask][u]$ = minimum distance to start from a source, visit all nodes in $mask$, and finish at node u . One recurrence is: $dp[mask][i] = \min(dp[mask \text{ without } i][j] + \text{cost}(j, i))$ over j in $mask$.
- **Counting/ways problems:** e.g. assigning caps/hats to people uses a DP over cap masks to count assignments.
- **Other subsets:** Partitioning nodes, matching, etc.

The state size is usually $O(2^N)$ (or $2^N \times N$ if storing an end index). The bitmask ($mask$) itself encodes which items are chosen. For example, USACO's bitmask DP tutorial says: "Let $dp[S][i]$ be the number of routes that visit all the cities in the subset S and end at city i " ⁵. Here S is a bitmask of visited cities. (In code, S can be an integer from 0 to 2^N-1 .) This pattern generalizes to $dp[mask][i]$ or $dp[i][mask]$. Typically, $dp[mask][i]$ means "using exactly the items in $mask$, ending at i ."

We also use bitmask DP to count combinations. For instance, in the "unique cap assignment" problem, one may use $dp[mask]$ to count ways to assign caps corresponding to bits in $mask$. In these problems the DP stores **number of ways** (or min cost).

Example: *Travelling Salesman Problem* - there are N cities, define $dp[mask][i]$ = minimum distance to start at city 0, visit exactly the cities in $mask$, and end at city i . Base case $dp[1 \ll 0][0] = 0$. Transitions: add a city to the tour. This uses bitmasks heavily (each mask subsets). USACO Guide gives exactly this: *"Let $dp[S][i]$ be the number of routes that visit all the cities in subset S and end at city i "*⁵ (one can change "number of routes" to "min cost" as needed). This is a standard *"min-cost Hamiltonian path"* DP. Other examples: using a bitmask of size $n \leq 20$ for DP on subsets (often denoted $mask$).

Key terms summary: - *Minimum/Maximum cost:* DP can store an optimal (min or max) cost or value for each state.
 - *Number of ways:* DP can instead count the number of valid solutions/subsets, storing an integer count (modulo something if large).
 - *Subarray:* a contiguous slice of an array, denoted $A[l..r]$. Interval DP states often refer to such subarrays.
 - *Bitmask:* a binary mask (integer) whose bits encode inclusion of items. A mask of length N represents any subset of $\{1..N\}$.

Each DP pattern above has its characteristic **state meaning** and **transitions**. By carefully defining the state to capture exactly the needed information (index, subarray ends, visited-set, etc.), you can write recurrences that build up from base cases. The cited examples and tutorials illustrate these common state definitions

¹ ² ⁴ ³ ⁵ .

Sources: DP introductions and examples on GeeksforGeeks and USACO Guide provide these state definitions and usages ¹ ² ⁴ ³ ⁵ . These explain how to interpret $dp[...]$ arrays in classic problems.

¹ Steps to solve a Dynamic Programming Problem - GeeksforGeeks

<https://www.geeksforgeeks.org/dsa/solve-dynamic-programming-problem/>

² Subset Sum Problem - GeeksforGeeks

<https://www.geeksforgeeks.org/dsa/subset-sum-problem-dp-25/>

³ Longest Palindromic Subsequence (LPS) - GeeksforGeeks

<https://www.geeksforgeeks.org/dsa/longest-palindromic-subsequence-dp-12/>

⁴ Dynamic Programming in Game Theory for Competitive Programming - GeeksforGeeks

<https://www.geeksforgeeks.org/dynamic-programming-in-game-theory-for-competitive-programming/>

⁵ Bitmask DP · USACO Guide

<https://usaco.guide/gold/dp-bitmasks>