🎒 Dynamic Programming: Knapsack Problem

Problem Setup:

Items: 5 items with profits P and weights W

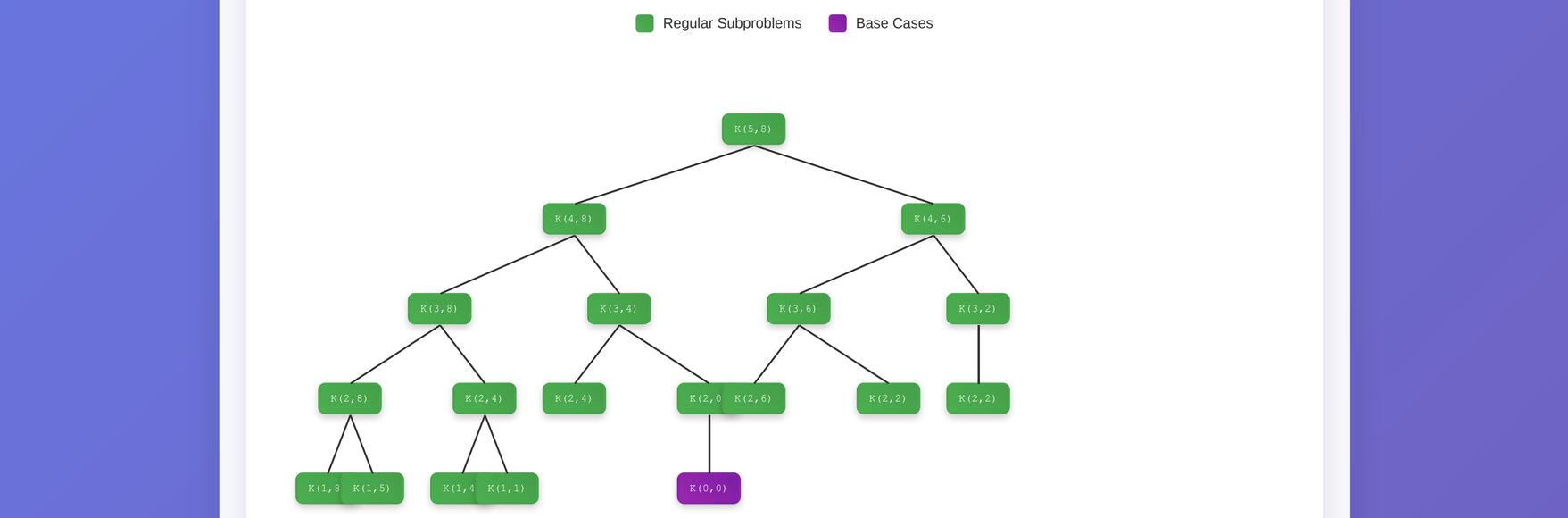
Knapsack Capacity: 8

Goal: Maximize profit while staying within weight limit

Item	1	2	3	4	5
Profit (P)	3	4	5	6	5
Weight (W)	2	3	4	5	2

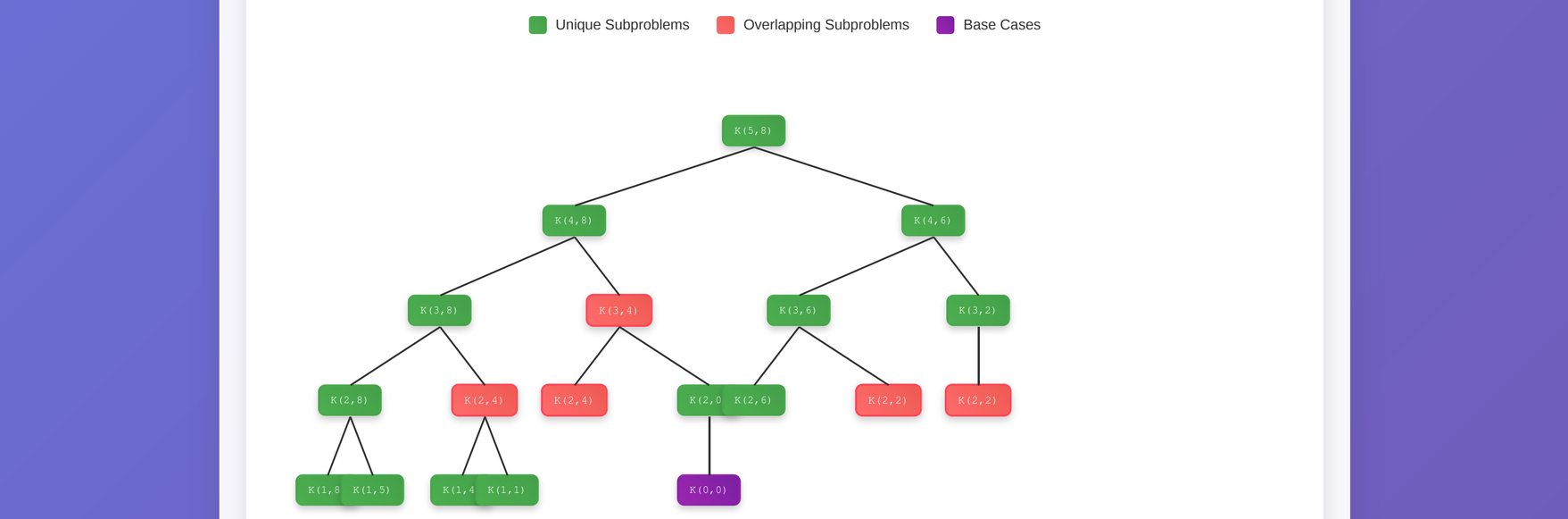
1️⃣ Recursive Tree Structure

This shows the complete recursive call tree for solving K(5,8). Each node represents a subproblem K(i,w).



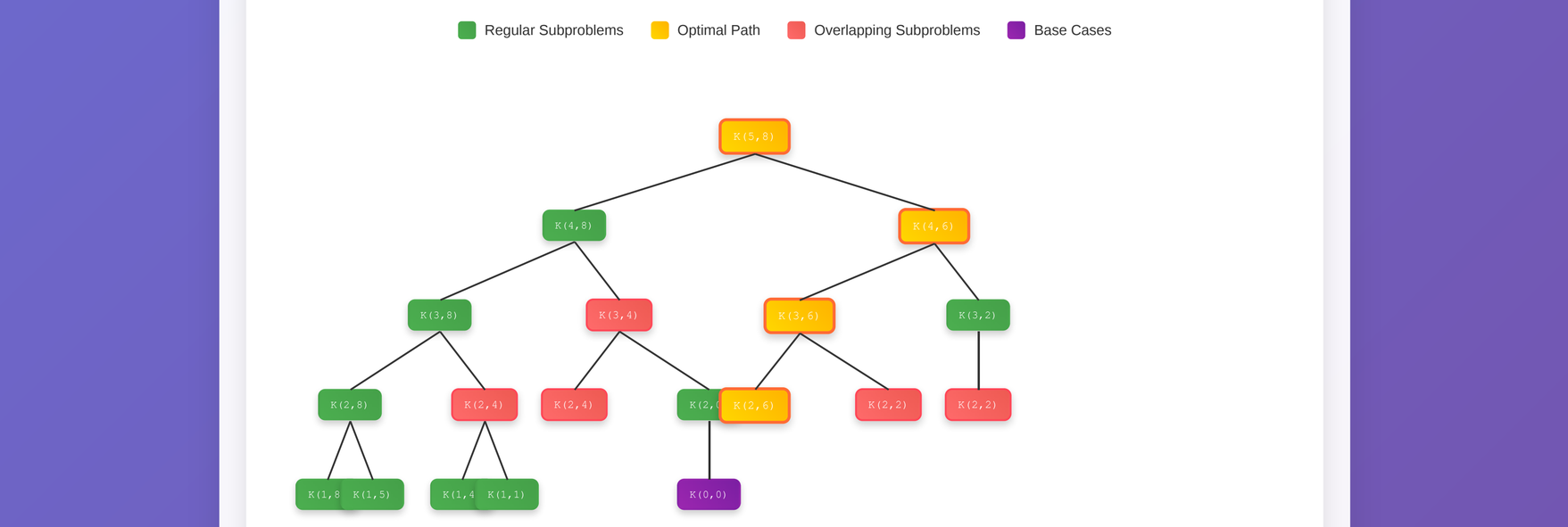
2️⃣ Overlapping Subproblems Highlighted

Red nodes show subproblems that appear multiple times in the recursion tree. These repeated calculations are what make DP beneficial!



3️⃣ Optimal Substructure Path

Gold nodes show one optimal path through the recursion tree. The optimal solution contains optimal solutions to subproblems.



🔄 Overlapping Subproblems

The same subproblem K(i,w) appears multiple times in the recursive tree. For example, K(3,4) and K(2,4) are computed several times with different paths leading to them.

Why DP helps: We can store the result of each subproblem the first time and reuse it instead of recalculating.

🏗️ Optimal Substructure

The optimal solution contains optimal solutions to subproblems. If K(5,8) is optimal, then the subproblems it depends on must also be optimal.

Principle of Optimality: Every sub-decision in the optimal solution is itself optimal.

📝 Understanding the Recursion: K(i, w)

K(i, w) = Maximum profit using items 1 to i with weight capacity w

Recurrence:

- If weight[i] > w: K(i,w) = K(i-1,w) *(can't include item i)*
- Otherwise: K(i,w) = max(K(i-1,w), profit[i] + K(i-1,w-weight[i]))

Base case: K(0,w) = 0 (no items = no profit)

🎯 Why Dynamic Programming Works Here:

1. Overlapping Subproblems: Subproblems like K(2,4), K(3,4), K(2,2) appear multiple times in different branches of the recursion tree. Instead of solving them repeatedly, we can memoize (store) their results.

2. Optimal Substructure: The optimal solution to K(5,8) depends on optimal solutions to smaller subproblems. If we choose to include item 5, then K(4,6) must be optimal for the remaining capacity.

Contrast with Divide & Conquer: In problems like merge sort, subproblems don't overlap - each subarray is unique. Here, the same (item, weight) combinations appear multiple times, making memoization beneficial.