Day 22 DP

ITSRUNTYM

Dynamic Programming (DP) is a powerful algorithmic technique used to **solve problems by breaking them down into smaller subproblems, solving each subproblem only once, and storing their results (memoization)** to avoid redundant work.

WHY USE DYNAMIC PROGRAMMING?

- **When to use DP**:
  Use DP when a problem has:
  1. **Overlapping Subproblems**
     → Subproblems are solved multiple times (e.g., Fibonacci).
  2. **Optimal Substructure**
     → Optimal solution of the problem depends on the optimal solutions of its subproblems.
- **Why it helps**:
  Instead of recalculating the result for the same input multiple times (like in plain recursion), **DP saves the result in memory and reuses it**, leading to a significant performance boost.

TYPES OF DP APPROACHES

Type	Description	Example Problem
Top-Down (Memoization)	Use recursion + store results in a table	Fibonacci using recursion and dp[]
Bottom-Up (Tabulation)	Solve subproblems iteratively, build solution from base	Fibonacci using a loop and dp[]
Space Optimization	Optimize space by storing only required results	Fibonacci using two variables

1. Top-Down (Memoization)
2. Bottom-Up (Tabulation)
3. Space Optimization

We will explain:
1. Why it's named like that
2. The **difference between the three**
3. When to use each
4. With diagrams and code for each

1. Top-Down Approach (Memoization)
Why this name?

- **"Top-Down"**: We start from the **main problem** and break it down into **subproblems recursively**.
- **"Memoization"**: We **store answers to subproblems** in a table (dp[]) to avoid recomputation.

Characteristics:

- Uses **recursion**
- Uses a **cache (dp[])** to save results of subproblems
- Useful when you're comfortable thinking recursively

General Template (Recursive + Cache)

```
int solve(int n, int[] dp) {
   if (n == 0 || n == 1) return n;
if (dp[n] != -1) return dp[n]; // already solved
return dp[n] = solve(n - 1, dp) + solve(n - 2, dp);
}
```

Diagram (Top-Down Flow)

```
Start from fib(5)
     |
 fib(5) = fib(4) + fib(3)
     |
  recurse on fib(4), fib(3), ...
Store in dp[] during the way down
```

2. Bottom-Up Approach (Tabulation)
Why this name?

- **"Bottom-Up"**: We solve **smaller subproblems first**, and use them to solve **bigger problems**.
- **"Tabulation"**: We use a **table (array)** to store the result in a **bottom-up fashion**.

Characteristics:

- Uses **loops (iteration)**, no recursion
- Builds the solution from **base cases upward**

- More memory efficient than top-down (no call stack overhead)

General Template

```
int fib(int n) {
    int[] dp = new int[n + 1];
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[n];
}
```

Diagram (Bottom-Up Flow)

```
Build from dp[0] → dp[1] → dp[2] → dp[3] ... up to dp[n]
dp[0] = 0
dp[1] = 1
dp[2] = dp[1] + dp[0] = 1
...
dp[n] = result
```

3. Space Optimization
Why this name?

- We **optimize space** usage by noticing that we don't need the whole dp[] table – only a few previous values.

Characteristics:

- Applies only when **current state depends on few previous states**
- Reduces space from O(n) → O(1) in some cases

General Template

```
int fib(int n) {
    if (n <= 1) return n;
    int prev2 = 0, prev1 = 1;
    for (int i = 2; i <= n; i++) {
        int curr = prev1 + prev2;
        prev2 = prev1;
        prev1 = curr;
    }
    return prev1;
}
```

Approach	Time Complexity	Space Complexity	Notes
Top-Down (Memoization)	O(n)	O(n)	Uses recursion + dp[]
Bottom-Up (Tabulation)	O(n)	O(n)	Iterative, no recursion
Space Optimization	O(n)	O(1)	Best for linear recurrence problems

Problem	DP Type	LeetCode Link
Fibonacci Number	1D DP	🔗 Link
Climbing Stairs	1D DP	🔗 Link
Min Cost Climbing Stairs	1D DP	🔗 Link
House Robber	1D DP	🔗 Link
Unique Paths	2D Grid DP	🔗 Link
Minimum Path Sum	2D Grid DP	🔗 Link
Longest Common Subsequence	2D DP on Strings	🔗 Link
Longest Palindromic Subsequence	2D DP on Strings	🔗 Link
Partition Equal Subset Sum	0/1 Knapsack (1D DP)	🔗 Link
Combination Sum IV	Subset Sum / Count Ways	🔗 Link
Target Sum	0/1 Knapsack / Subset Sum	🔗 Link
Edit Distance	2D DP on Strings	🔗 Link
Dungeon Game	2D DP (Reverse Build)	🔗 Link
Jump Game II	1D DP + Greedy	🔗 Link
Cherry Pickup II	3D DP (Advanced Grid)	🔗 Link