

Climbing Stairs

Your first DP — Fibonacci in disguise ■

● EASY

■ Hook

You're at the bottom of a staircase with n steps. Each time you can climb 1 or 2 steps. How many distinct ways can you reach the top? This is the 'Hello World' of Dynamic Programming — simple enough to understand, deep enough to reveal the core DP pattern.

■ Real World

Think of it like counting paths in a city grid, or counting binary sequences, or counting ways to tile a floor. The same recurrence shows up everywhere.

■ Core Intuition — Overlapping Subproblems

To reach step n , you either came from step $n-1$ (took 1 step) or step $n-2$ (took 2 steps). So: **ways(n) = ways($n-1$) + ways($n-2$)**. That's it. That's Fibonacci. The insight is: *don't recompute the same subproblem twice* — store results.

■ The Two DP Pillars

1. OPTIMAL SUBSTRUCTURE: answer to n depends on answers to $n-1$ and $n-2$. 2. OVERLAPPING SUBPROBLEMS: ways(3) is computed inside ways(5) AND ways(4). Memoize it — compute once, reuse many times.

■ Recursion Tree + DP Table

NAIVE RECURSION TREE for $n=5$ (shows repeated work):

```

      f(5)
     /  \
    f(4)  f(3) ← f(3) computed TWICE!
   /  \  /  \
  f(3) f(2) f(2) f(1) ← f(2) computed 3 TIMES!
 / \ / \ / \
f(2) f(1) f(1) f(0) f(1) f(0)

```

DP TABLE (bottom-up, $n=5$):

i:	0	1	2	3	4	5
dp[]:	1	1	2	3	5	8

← $dp[i] = dp[i-1] + dp[i-2]$
 ↑ answer = 8 ways

■ ■ How It Works

1

Base cases

$dp[0]=1$ (one way to stay at bottom), $dp[1]=1$ (only one way to reach step 1).

2

Fill table

For i from 2 to n : $dp[i] = dp[i-1] + dp[i-2]$.

3

Return $dp[n]$

The answer is stored at index n .

4

Space optimize

You only need the last two values — use two variables instead of array.

■ Code

```
// Approach 1: Bottom-Up DP O(n) time, O(n) space
public int climbStairs(int n) {
    if (n <= 1) return 1;
    int[] dp = new int[n+1];
    dp[0] = 1; dp[1] = 1;
    for (int i = 2; i <= n; i++)
        dp[i] = dp[i-1] + dp[i-2];
    return dp[n];
}

// Approach 2: Space-Optimized O(n) time, O(1) space
public int climbStairsOpt(int n) {
    if (n <= 1) return 1;
    int prev2 = 1, prev1 = 1;
    for (int i = 2; i <= n; i++) {
        int curr = prev1 + prev2;
        prev2 = prev1;
        prev1 = curr;
    }
    return prev1;
}
```

■ ■ Complexity

■ TIME

$O(n)$

■ SPACE

$O(1)$ optimized

■ ■ Common Mistakes

■ Starting base case at 0 or 1 incorrectly

$dp[0]=1$ means 'one way to stand at ground (do nothing)'. $dp[1]=1$. If you start $dp[1]=0$ or $dp[0]=0$, everything cascades wrong.

■ Practice

#	Problem	Diff	Pattern
1	Climbing Stairs (LC 70)	Easy	DP / Fibonacci
2	Min Cost Climbing Stairs (LC 746)	Easy	DP
3	N-th Tribonacci Number (LC 1137)	Easy	DP
4	Frog Jump (count ways)	Medium	DP
5	Decode Ways (LC 91)	Medium	DP

- ✓ $ways(n) = ways(n-1) + ways(n-2) \leftarrow$ the recurrence
- ✓ Base: $ways(0)=1, ways(1)=1$
- ✓ Bottom-up DP table or two-variable rolling approach
- ✓ Time: $O(n)$ | Space: $O(1)$ with optimization
- ✓ This IS Fibonacci — just reframed as a path-counting problem

■ MENTAL MODEL — Remember This Forever

To reach step N , you must have come from $N-1$ or $N-2$. The number of ways to reach N = sum of ways to reach those two predecessors. DP says: compute each subproblem once, store it, never repeat. This is the heartbeat of every DP solution you'll ever write.