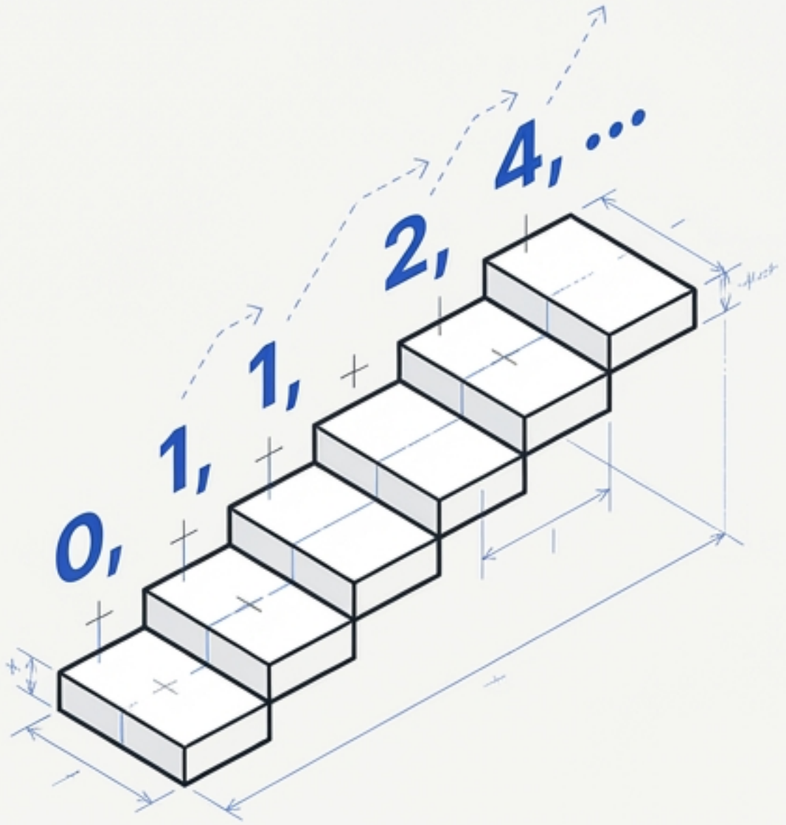
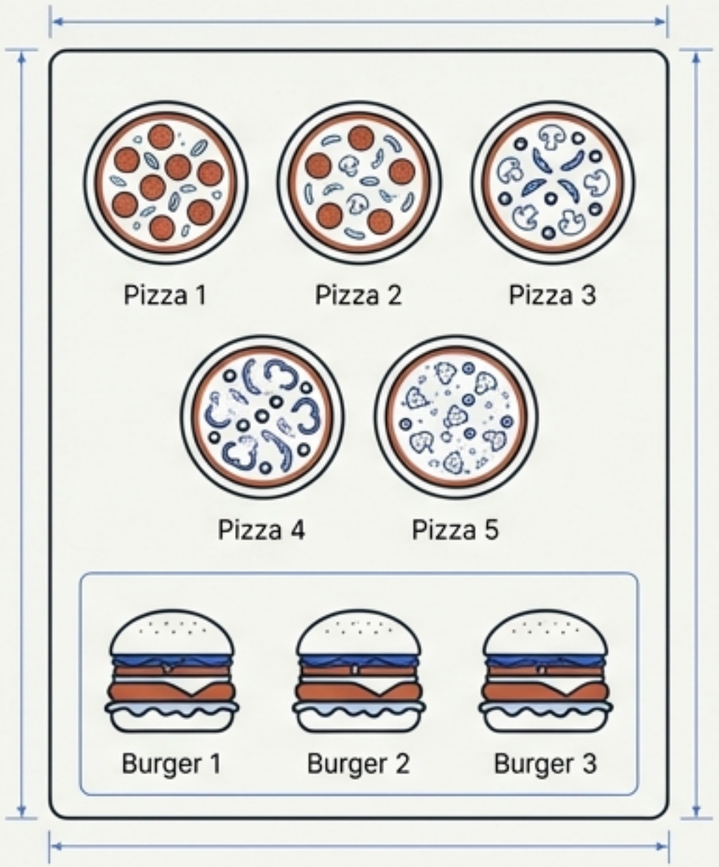


Counting Principles & The Staircase Problem

A Visual Guide to Dynamic Programming

EVOLUTION OF A SOLUTION // INTUITION TO OPTIMIZED CODE

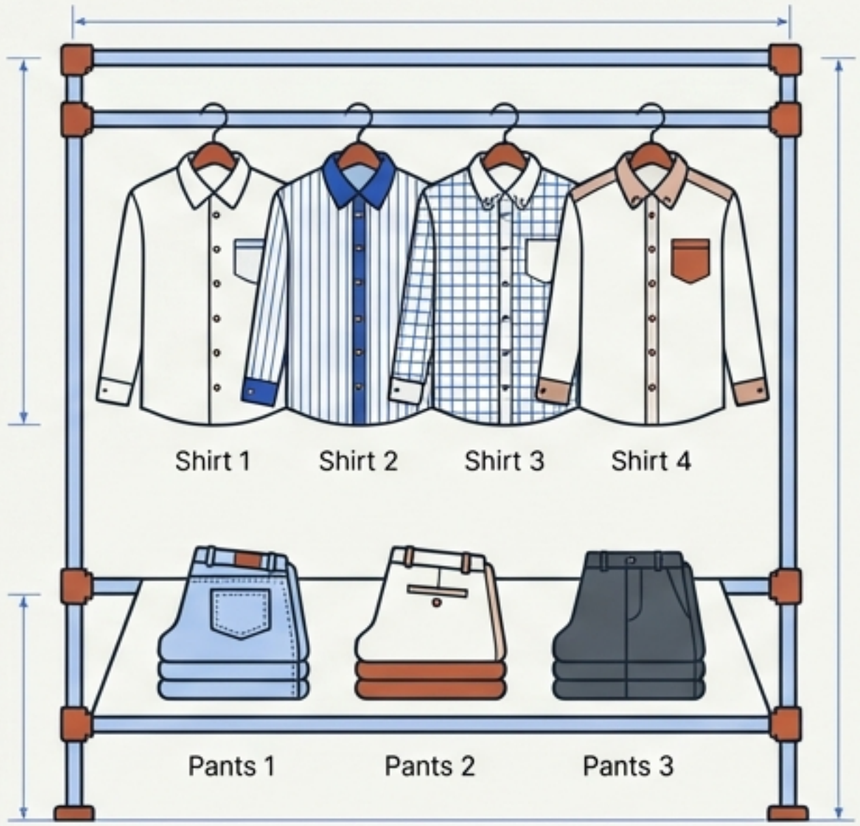
The Sum Rule (OR)



If you can do A in m ways OR B in n ways, the total is m + n.

Total Choices = 5 + 3 = 8

The Product Rule (AND)

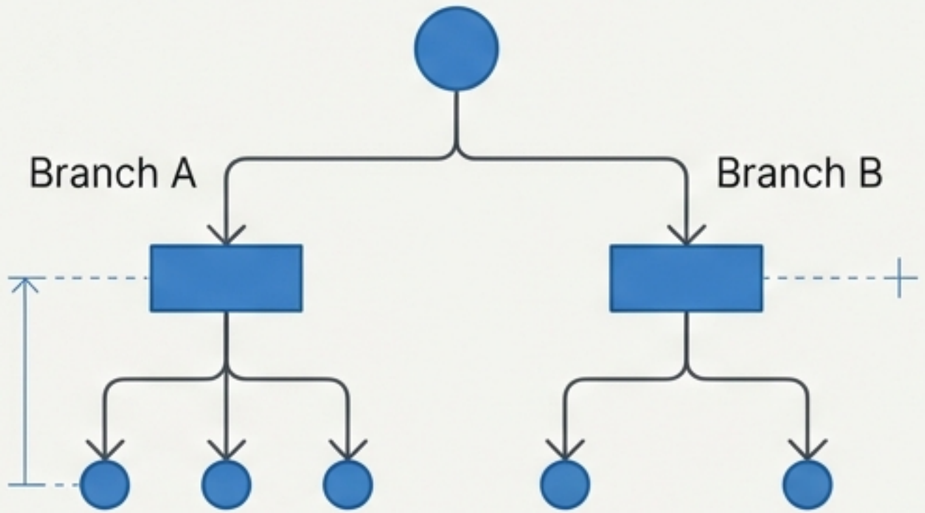


If you can do A in m ways AND B in n ways, the total is m × n.

Total Outfits = 4 × 3 = 12

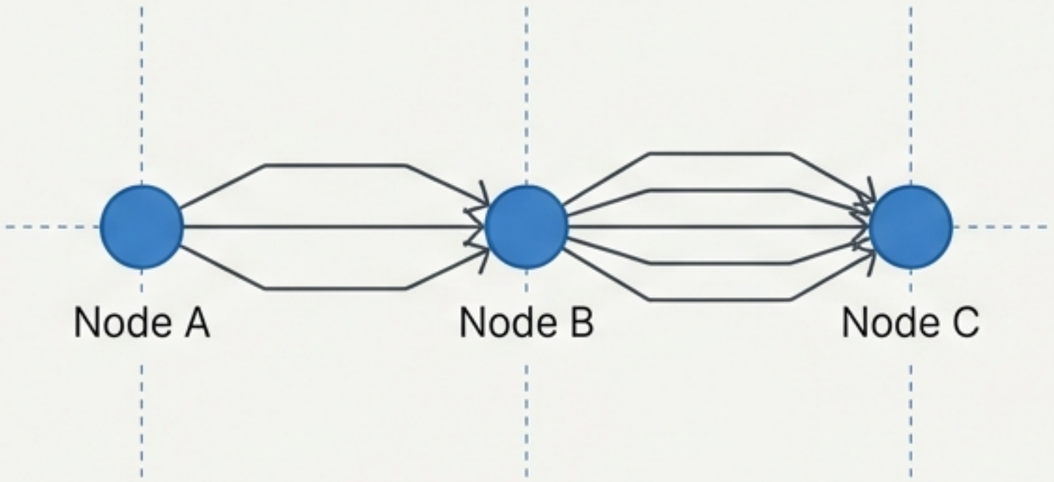
Visualizing the Difference

Sum Rule (Parallel)



Independent Choices. 3 + 2 = 5

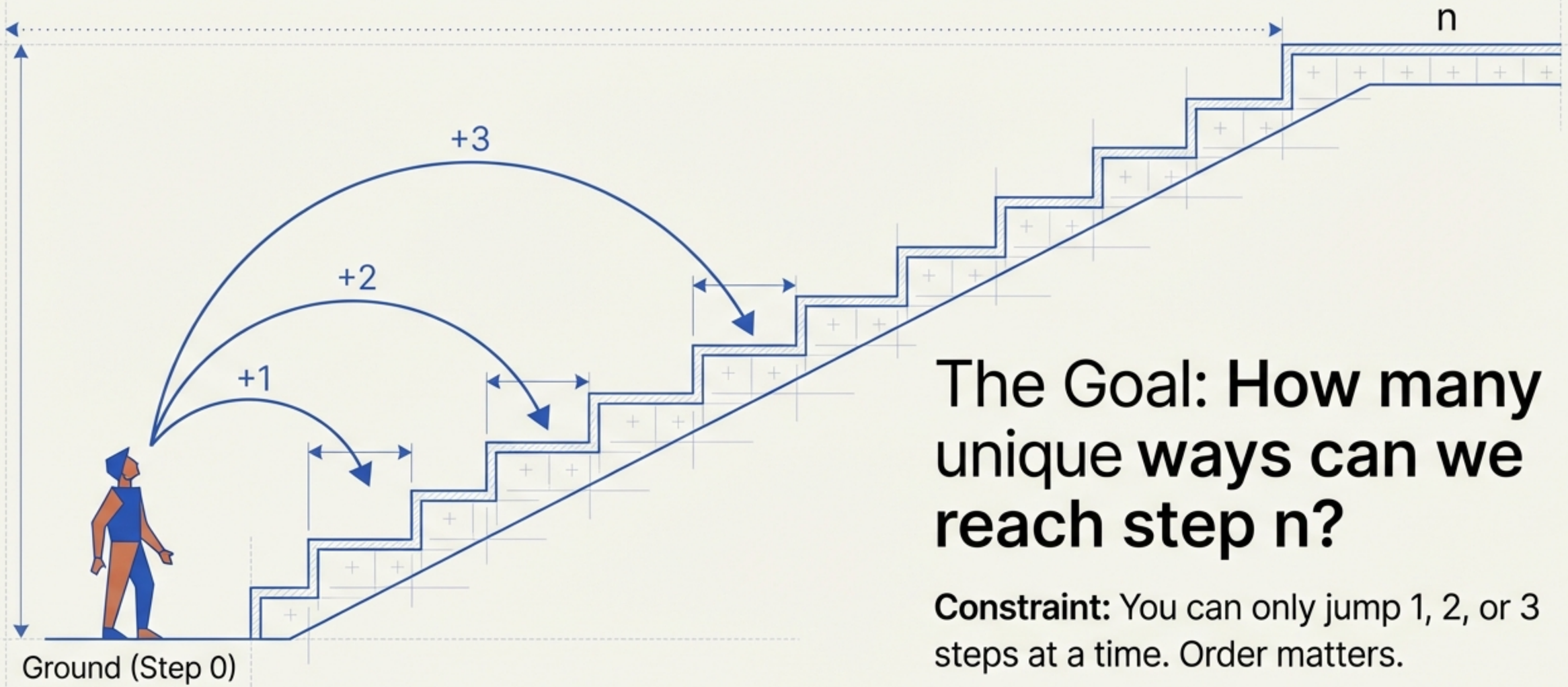
Product Rule (Sequential)



Compound Stages. 3 × 4 = 12

Key Insight: Independent choices add up. Sequential stages multiply.

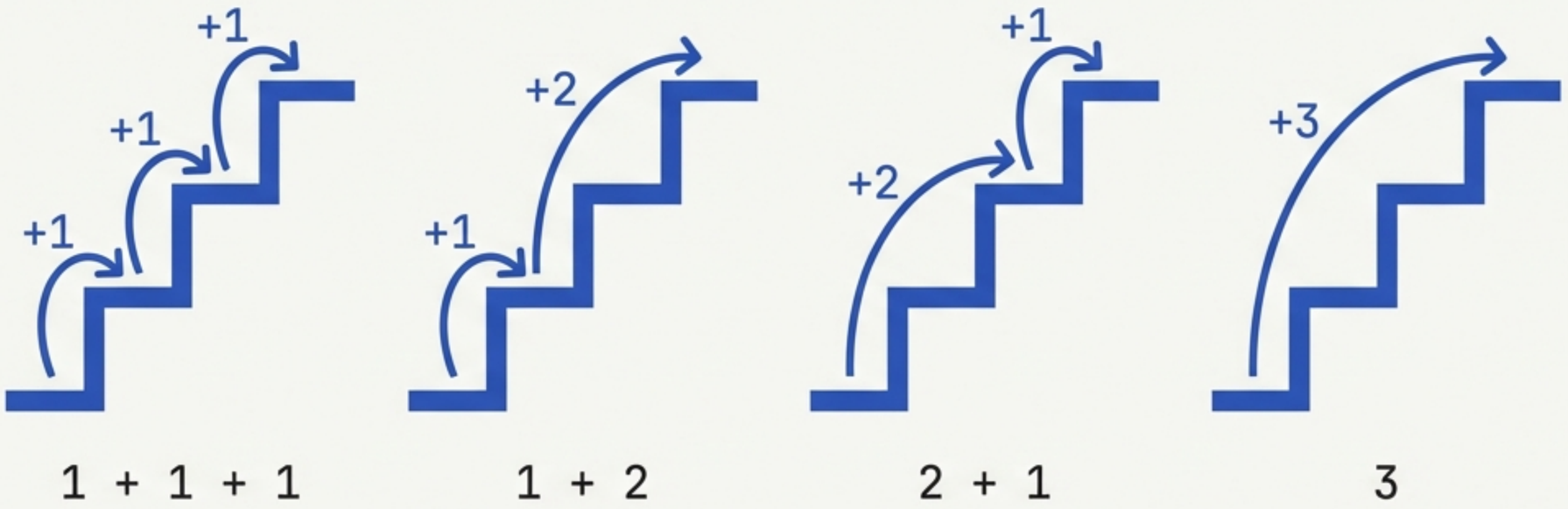
The Challenge: The Staircase Problem



The Goal: **How many** unique **ways can we reach step n?**

Constraint: You can only jump 1, 2, or 3 steps at a time. Order matters.

Intuition: Solving for Small N (n=3)

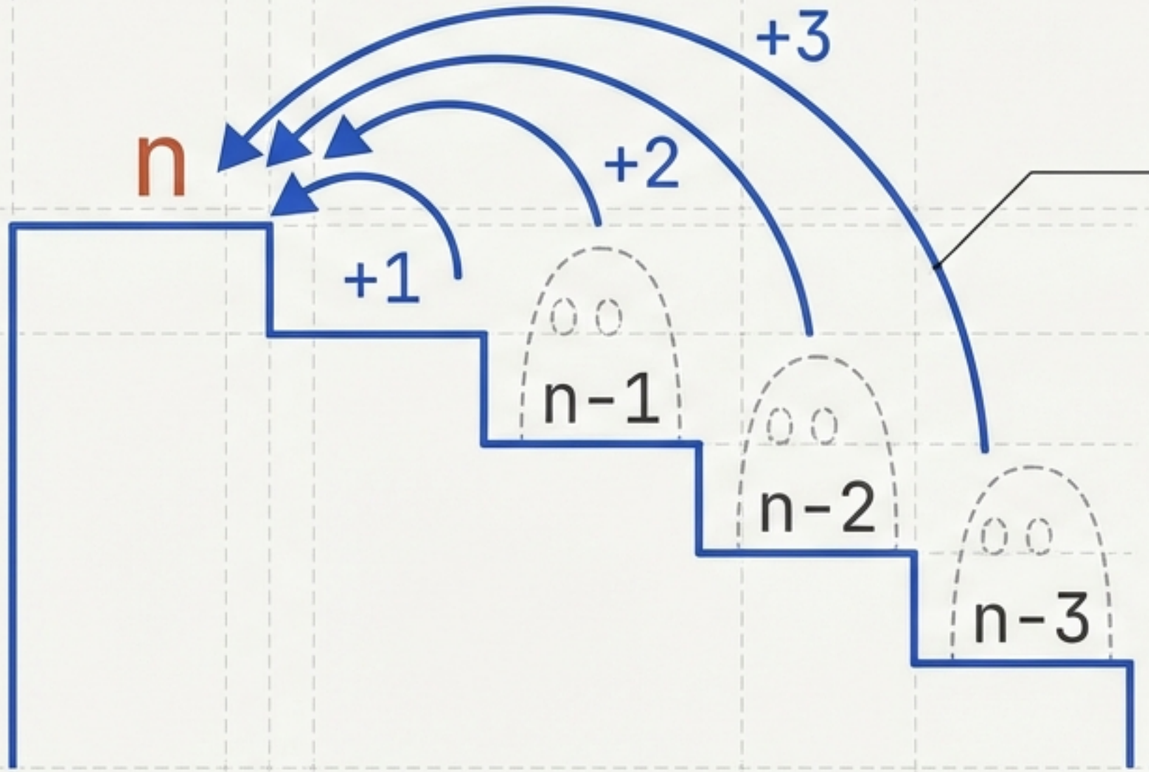


1 + 1 + 1 1 + 2 2 + 1 3

We can list these manually for small numbers. But as n grows, the combinations explode. We need a a repeatable pattern, not a list.

Total Ways = 4

The Core Insight: The “Last Jump” Principle



To land on step n, you MUST have come from n-1, n-2, or n-3. There is no other way.

Therefore, Ways(n) = Sum of ways to reach those previous steps.

The Recurrence Relation

$$f(n) = f(n-1) + f(n-2) + f(n-3)$$

Base Case 0

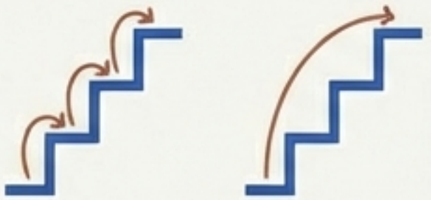
$f(0) = 1$ (Ground)

Base Case 1

$f(1) = 1$ (1 step jump)

Base Case 2

$f(2) = 2$ (1+1 or 2)

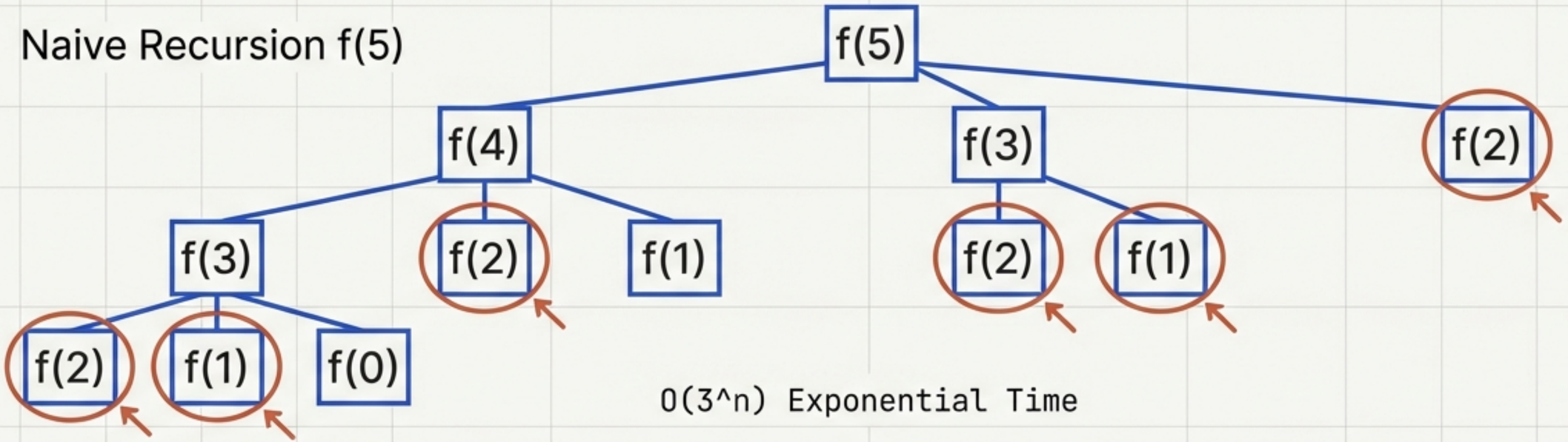


This formula aggregates the history of the path.
It sums the possibilities of the previous valid states.

Architectural Editorial

The Trap of Recursion vs. Bottom-Up Building

Naive Recursion f(5)



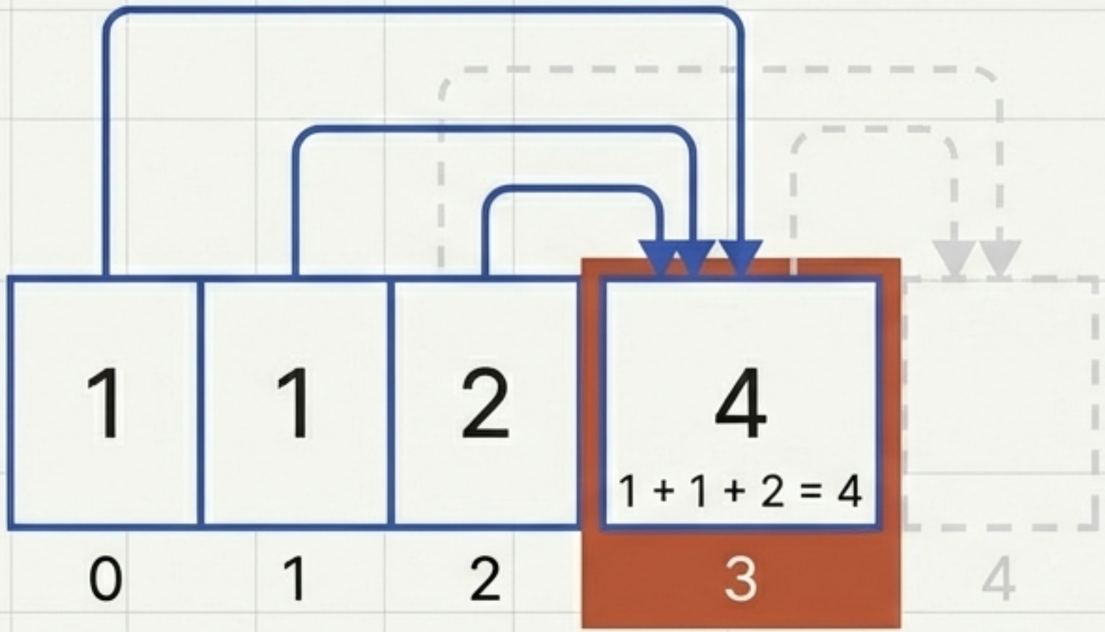
`O(3^n) Exponential Time`

Bottom-Up Dynamic Programming. O(n) Linear Time.

Don’t re-solve the same puzzle. Solve it once, remember it, and build on it.

Architectural Editorial

Solution V1: The Array Method



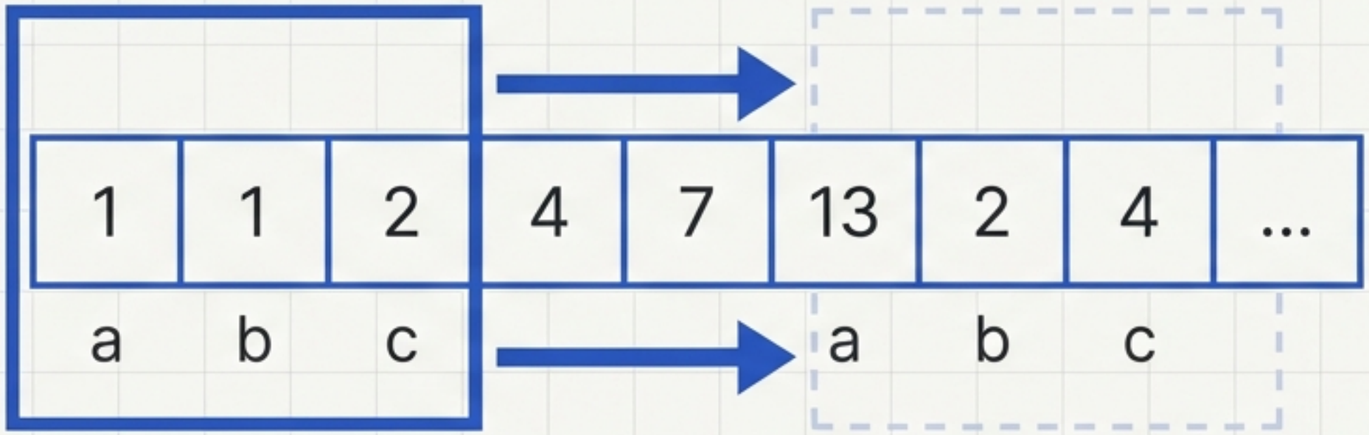
Visualization: Building the `ways` array iteratively, using the sum of the three previous states.

```
ways = [0] * (n + 1)
ways[0], ways[1], ways[2] = 1, 1, 2

for i in range(3, n + 1):
    ways[i] = ways[i-1] + ways[i-2]
            + ways[i-3] + ways[i-3]

return ways[n]
```

Time: O(n) | Space: O(n)

Solution V2: Space Optimization

Sliding Window



We don’t need the whole staircase. We only need the last 3 steps.

```
a, b, c = 1, 1, 2

for i in range(3, n + 1):
    current = a + b + c
    a, b, c = b, c, current

return c
```

Time: O(n) | **Space:** O(1) (Constant)

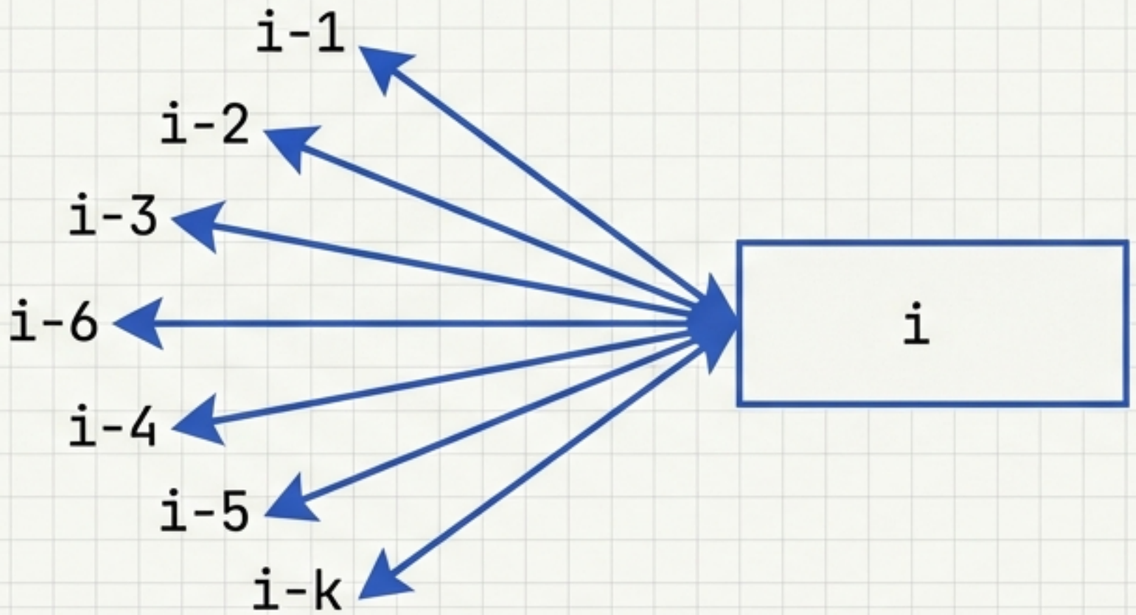
Architectural Editorial

Pattern Recognition: The Tribonacci Sequence

Fibonacci (Sum of last 2)	**Tribonacci / Staircase** (Sum of last 3)
0, 1, 1, 2, 3, 5, 8, 13...	1, 1, 2, 4, 7, 13, 24, 44, 81...
2,	7,
5,	24,
8,	44,
13...	81...

Key Test Cases:
f(5) = 13
f(10) = 274

Generalization: The ‘k’ Steps Variant



What if we can jump up to k steps?

```
# Inside main loop
total = 0
for j in range(1, k + 1):
    if i - j >= 0:
        total += ways[i - j]
ways[i] = total
```

The logic remains identical. The window size simply expands from 3 to k.

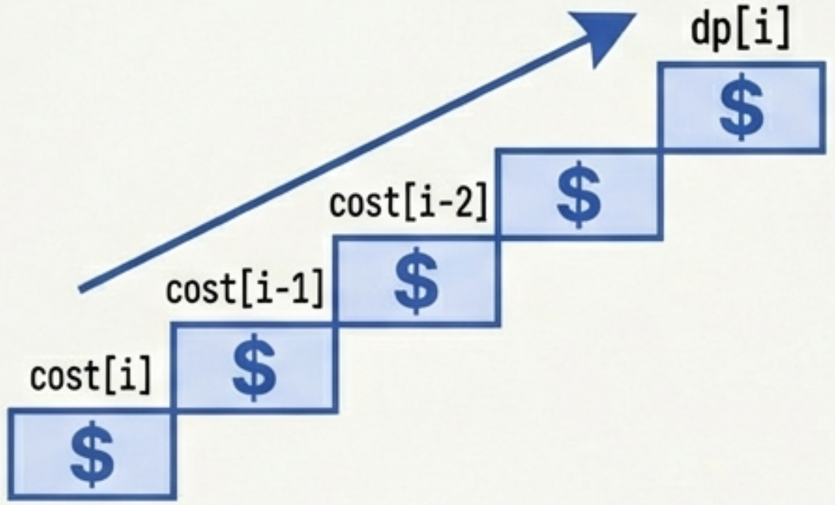
Complexity Analysis & Decision Guide

Method	Time Complexity	Space Complexity	Use Case
Naive Recursion	O(3^n) (Exp)	O(n) Stack	Avoid (Too Slow)
Array DP	O(n)	O(n)	Learning / Debugging
Sliding Window	**O(n)**	**O(1)**	**Production / Interview**

For very large n, O(1) space is the critical engineering constraint.

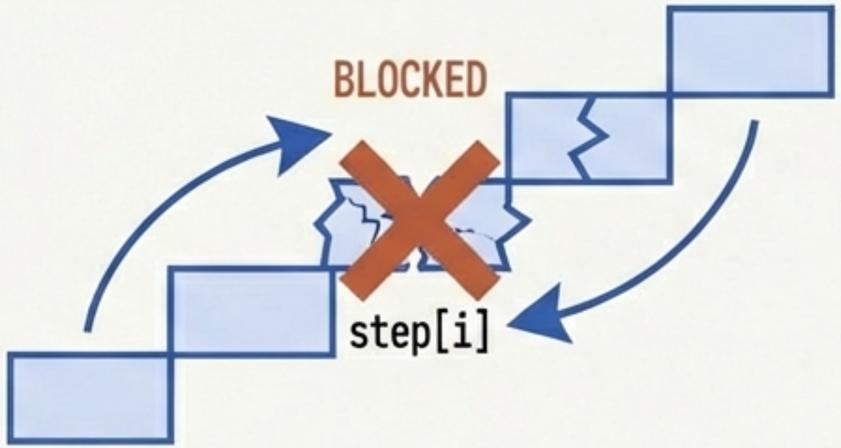
Practica Practical Extensions

Min Cost Climbing




```
dp[i] = cost[i] + min(dp[i-1], dp[i-2])
```

Blocked Steps




```
If step is broken: ways[i] = 0
```

The framework is flexible. We can add costs, obstacles, or varying constraints without changing the core Dynamic Programming structure.

The Staircase Cheat Sheet

The Mental Model

Ways(n) = Sum of ways to reach previous possible steps.

Base Cases

- 0 -> 1
- 1 -> 1
- 2 -> 2

The Formula

$$f(n) = f(n-1) + f(n-2) + f(n-3)$$

Optimized Template Code

```
def countWays(n):
    a, b, c = 1, 1, 2
    for _ in range(3, n + 1):
        a, b, c = b, c, a + b + c
    return c
```

Strategy: Start small. Find the pattern. Build bottom-up.