Topic Introduction

Today we're diving into **Dynamic Programming (DP)**, a powerhouse technique for solving problems that can be broken down into overlapping subproblems. If you've ever heard the phrase "solve smaller versions, build up the answer," you're already halfway there!

Dynamic Programming is all about solving complex problems by breaking them down into simpler subproblems, solving each subproblem just once, and storing their solutions—typically in an array or table (memoization or tabulation). This prevents redundant work and transforms exponential brute-force solutions into efficient polynomial-time algorithms.

How does DP work?

- If a problem can be defined in terms of smaller subproblems (with overlapping subproblems and optimal substructure), DP is likely a fit.
- You'll often see DP used for: counting ways to do something, finding optimal (min/max) solutions with constraints, or breaking sequences into parts.
- It's especially common in interview questions involving steps, sequences, or decisions with dependencies.

Example (not using today's problems):
Imagine you want to find the number of ways to reach the end of a board of size `n`, moving 1 or 2 spaces at a time. You can define `ways(n) = ways(n-1) + ways(n-2)`. This is a classic DP recurrence!

Let's look at three classic DP problems that all share a similar mathematical backbone:

- Climbing Stairs
- Min Cost Climbing Stairs
- Fibonacci Number

Why are these grouped together?
All three are built around the same recurrence: each state depends on the last one or two states. Climbing Stairs asks, "How many ways?" Min Cost Climbing adds a twist: it's not just about counting, but optimizing cost. And Fibonacci is the mathematical underpinning of both!

Problem 1: Climbing Stairs

Problem Link: Climbing Stairs - LeetCode

Restated Problem:
You are climbing a staircase with `n` steps. Each time, you can go up either 1 or 2 steps. How many distinct ways are there to reach the top?

Example Input/Output:
Input: `n = 4`
Output: `5`
Explanation:

- 1+1+1+1
- 1+2+1
- 2+1+1
- 1+1+2
- 2+2

Thought Process:

Try to break this down on paper for small values of `n`. You'll notice a pattern:

- To reach step `n`, you must have come from either step `n-1` (taking 1 step) or step `n-2` (taking 2 steps).
- This means: `ways(n) = ways(n-1) + ways(n-2)`

Try this test case yourself:

Input: `n = 5`

How many ways are there?

Brute-force approach:

Try all possible step combinations recursively.

- Time complexity: Exponential, O(2^n), because each call branches into two further calls.

Optimal approach (Dynamic Programming):

- Recognize that this is essentially the Fibonacci sequence!
- Use an array `dp` where `dp[i]` represents the number of ways to reach step `i`.
- Base cases:
  - `dp[0] = 1` (one way to stand at the bottom)
  - `dp[1] = 1` (one way to reach the first step)
- For each step from 2 to `n`, compute:
  - `dp[i] = dp[i-1] + dp[i-2]`

Python Solution:

```
def climbStairs(n):
    # Handle base cases
    if n == 0 or n == 1:
        return 1

    # Initialize the dp array
    dp = [0] * (n + 1)
    dp[0] = 1  # 1 way to stay at ground
    dp[1] = 1  # 1 way to reach first step

    # Build up the solution
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]
```

Time Complexity: O(n)

Space Complexity: O(n) (can be optimized to O(1) with two variables!)

What's happening?

- We initialize the ways to reach step 0 and 1.
- For each step from 2 to n, we calculate the number of ways to get there.
- The answer is stored in `dp[n]`.

Trace Example (n = 4):

- dp[0] = 1
- dp[1] = 1
- dp[2] = dp[1] + dp[0] = 2
- dp[3] = dp[2] + dp[1] = 3
- dp[4] = dp[3] + dp[2] = 5

Try this test case:

Input: `n = 6`

How many ways are there?

Take a moment to solve this on your own before jumping into the solution.

Reflective prompt:

Did you know this can also be solved with just two variables (like Fibonacci)? Try rewriting the code with no array!

Problem 2: Min Cost Climbing Stairs

Problem Link: Min Cost Climbing Stairs - LeetCode

Restated Problem:

You're given an array `cost`, where `cost[i]` is the cost of step `i`. You can start at step 0 or step 1, and each time you can climb 1 or 2 steps. What is the minimum total cost to reach the top (past the last step)?

How is this similar/different?

Like Climbing Stairs, but instead of counting ways, you're minimizing cost. The DP pattern is nearly identical, but now you add the current step's cost.

Brute-force approach:

Try all possible paths, summing costs recursively.

- Very slow: O(2^n) time.

Optimal approach (Dynamic Programming):

- Let `dp[i]` be the minimum cost to reach step `i`.
- For each step, you can come from `i-1` or `i-2`, so:

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

- You can start at step 0 or 1, so the answer is `min(dp[n-1], dp[n-2])`.

Step-by-step explanation:

- Initialize `dp[0] = cost[0]`, `dp[1] = cost[1]`.

- For each step from 2 to n-1:
    - Calculate the minimal cost to reach that step.
- The answer is the minimum of the last two `dp` values, since you can finish by taking either a 1-step or 2-step jump to the end.

Example Input/Output:

Input: `cost = [10, 15, 20]`

Output: `15`

Explanation: Start at step 1 (cost 15), then take a 2-step to the top (no added cost).

Try this test case yourself:

Input: `cost = [1, 100, 1, 1, 1, 100, 1, 1, 100, 1]`

What's the minimum cost?

Pseudocode:

```
function minCostClimbingStairs(cost):
    n = length of cost
    dp[0] = cost[0]
    dp[1] = cost[1]
    for i from 2 to n-1:
        dp[i] = min(dp[i-1], dp[i-2]) + cost[i]
    return min(dp[n-1], dp[n-2])
```

Trace Example (cost = [10, 15, 20]):

- dp[0] = 10
- dp[1] = 15
- dp[2] = min(15, 10) + 20 = 10 + 20 = 30
- min(dp[1], dp[2]) = min(15, 30) = 15

Another test case to dry run:

Input: `cost = [5, 7, 2, 4, 3, 8]`

What's the minimum cost to reach the top?

Time Complexity: O(n)

Space Complexity: O(n) (can be optimized to O(1))

Problem 3: Fibonacci Number

Problem Link: Fibonacci Number - LeetCode

Restated Problem:

Given `n`, return the nth Fibonacci number.

- F(0) = 0
- F(1) = 1
- F(n) = F(n-1) + F(n-2) for n > 1

What's different?

This is the mathematical sequence at the heart of the last two problems. No cost, no steps—just compute the nth number in the sequence.

Brute-force approach:

Simple recursion: `fib(n) = fib(n-1) + fib(n-2)`

- Time complexity: O(2^n), very slow for large n.

Optimal approach (Dynamic Programming):

Same DP structure as before!

- Let `dp[0] = 0`, `dp[1] = 1`
- For `i` from 2 to n:

`dp[i] = dp[i-1] + dp[i-2]`

- Return `dp[n]`

Pseudocode:

```
function fib(n):
    if n == 0 or n == 1:
        return n
    dp[0] = 0
    dp[1] = 1
    for i from 2 to n:
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

Example:

Input: `n = 4`

Output: `3`

Sequence: 0, 1, 1, 2, 3

Another test case:

Input: `n = 7`

Can you compute the answer?

Time Complexity: O(n)

Space Complexity: O(n) (try to optimize to O(1) with two variables!)

Nudge:

Notice how all three problems can be solved with almost the same state transition? Can you spot more such problems?

Summary and Next Steps

You've just explored three classic DP problems that all use the *same* core pattern:

- The solution at position `i` depends on positions `i-1` and `i-2`.
- Each builds on the last, moving from pure counting (Fibonacci), to counting ways (Climbing Stairs), to optimizing (Min Cost Climbing).

Key Insights:

- Recognize the recurrence relation pattern. Whenever you see choices like "1 or 2 steps," think of DP!
- Build up solutions from base cases.
- For optimization problems, adjust your state to track min/max as needed.
- You can often reduce space from O(n) to O(1) by only keeping the last two states.

Common mistakes:

- Forgetting the proper base cases.
- Off-by-one errors in DP table size or indices.
- Not considering all ways to start or finish (like starting at step 0 or 1 in Min Cost Climbing Stairs).

Action List

- Solve all three problems on your own, even the one we coded together.
- Try writing the solutions with only two variables (no arrays).
- Explore related problems with different step sizes or constraints.
- Compare your solutions with others—see how they handle edge cases.
- If you get stuck, review the recurrence and walk through small examples by hand.
- Most importantly: keep practicing! Mastery comes from repetition and reflection.

Happy climbing (and coding)!