🪜 746. Min Cost Climbing Stairs

🔍 Problem Statement

You are given an integer array `cost` where `cost[i]` is the cost of the *i-th* step on a staircase.

Once you pay the cost, you can either climb **one or two steps**.

You can start from **index 0 or index 1**.

Return the **minimum cost** to reach the **top of the floor**.

🧩 Example 1

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

Output: `15`

Explanation:

- Start from index `1`.
- Pay `15` and climb two steps to reach the top.

  ✅ Total cost = `15`

🧩 Example 2

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

Output: `6`

Explanation:

Optimal path → 0 → 2 → 4 → 6 → 7 → 9 → top

✅ Total cost = `1 + 1 + 1 + 1 + 1 + 1 = 6`

⚙️ Approach: Dynamic Programming (Top-Down with Memoization)

🔶 Intuition

At each step `i`, we can either come from:

- step `i-1` (pay `cost[i-1]`)
- or step `i-2` (pay `cost[i-2]`)

So,

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

We finally take the **minimum of the last two steps** to reach the top.

🧠 Recurrence Relation

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

Base cases:

```
helper(0) = cost[0]
helper(1) = cost[1]
```

Final answer:

```
min(helper(n-1), helper(n-2))
```

💻 Code (C++)

```
class Solution {
public:
    int helper(vector<int>& cost, int i, vector<int>& dp) {
        if (i == 0 || i == 1) return cost[i];
        if (dp[i] != -1) return dp[i];
        return dp[i] = cost[i] + min(helper(cost, i - 1, dp), helper(cost, i - 2, dp));
    }

    int minCostClimbingStairs(vector<int>& cost) {
        int n = cost.size();
        vector<int> dp(n, -1);
```

```
        return min(helper(cost, n - 1, dp), helper(cost, n - 2, dp));
    }
};
```

⏱️ Time Complexity

- **O(n)** → Each state (step) is computed once.

💾 Space Complexity

- **O(n)** → For the recursion + memoization array.

🧭 Alternate (Bottom-Up DP)

```
int minCostClimbingStairs(vector<int>& cost) {
    int n = cost.size();
    for (int i = 2; i < n; i++) {
        cost[i] += min(cost[i-1], cost[i-2]);
    }
    return min(cost[n-1], cost[n-2]);
}
```

✅ **Optimized**: No extra space used.

🏁 Key Takeaways

- Can start from step 0 or 1.
- Either climb one or two steps each time.
- Final cost = `min(dp[n-1], dp[n-2])`.
- Classic **Dynamic Programming** pattern (like Fibonacci).
