Ahh lovely choice, love ❤️ — **LeetCode 509. Fibonacci Number**

1. Problem Title & Link

- **509. Fibonacci Number**
- 🔗 https://leetcode.com/problems/fibonacci-number/

2. Problem Statement (Short Summary)
The **Fibonacci sequence** is defined as:

F(0) = 0, F(1) = 1
F(n) = F(n-1) + F(n-2) for n > 1
Given n, return the **n-th Fibonacci number**.

3. Examples (Input → Output)

Input: n = 2
Output: 1
Explanation: F(2) = F(1) + F(0) = 1 + 0 = 1

Input: n = 3
Output: 2

Input: n = 10
Output: 55

4. Constraints

- 0 <= n <= 30
  (Though simple, used to build recursion and DP understanding.)

5. Thought Process (Step by Step)

There are **three major approaches**, each excellent for teaching progressively ❤️

🧩 Approach 1: Recursive (Brute Force)

Use the definition directly:

F(n) = F(n-1) + F(n-2)
But this recomputes subproblems many times → exponential time $O(2^n)$.

Good for teaching recursion structure, *not efficiency*.

🧩 Approach 2: Memoization (Top-Down DP)

Store results in a dictionary/array to avoid recomputation.

✅ Time: O(n)

✅ Space: O(n) (recursion + memo table)

🧩 Approach 3: Tabulation (Bottom-Up DP)

Start from base cases and build up iteratively.

✅ Time: O(n)

✅ Space: O(1)

6. Pseudocode (Iterative – Best for Students)

```
if n <= 1:
    return n

a = 0
b = 1

for i in range(2, n+1):
    c = a + b
    a = b
    b = c

return b
```

7. Code Implementation

✅ Python

```
class Solution:
    def fib(self, n: int) -> int:
        if n <= 1:
            return n
        a, b = 0, 1
        for _ in range(2, n + 1):
```

```
            a, b = b, a + b
        return b
```

✅ **Java**

```
class Solution {
    public int fib(int n) {
        if (n <= 1) return n;
        int a = 0, b = 1;
        for (int i = 2; i <= n; i++) {
            int c = a + b;
            a = b;
            b = c;
        }
        return b;
    }
}
```

8. Time & Space Complexity

Approach	Time	Space	Notes
Recursive	$O(2^n)$	O(n)	Exponential, repeated work
Memoized	O(n)	O(n)	Fast, uses recursion
Iterative	O(n)	O(1)	Best balance (efficient + simple)

9. Dry Run (Step-by-Step Execution)

👉 Input: n = 5

Step	i	a (F(i-2))	b (F(i-1))	c (F(i))
Start	—	0	1	—
2	2	0	1	1
3	3	1	1	2
4	4	1	2	3
5	5	2	3	5

✅ Output: 5

10. Concept Insight Table

Core Concept	Common Use Cases	Common Traps	Builds / Next Steps
Dynamic Programming (Recurrence Optimization) — break problems into overlapping subproblems and reuse results.	- Recursion → Memoization → Tabulation transitions - Series problems (Catalan, Tribonacci) - Optimization questions	- Forgetting base cases (n=0, n=1) - Stack overflow in plain recursion - Off-by-one in iterative loop	🔷 Builds to LC 70 (Climbing Stairs) 🔷 Connects to DP State Transition Thinking 🔷 Foundation for Tabulation & Space Optimization

11. Common Mistakes / Edge Cases

- Missing if n <= 1 base case.
- Recomputing recursively without memoization → TLE.
- Confusing “index” vs “term number.”

12. Variations / Follow-Ups

- **LC 70:** Climbing Stairs — same Fibonacci recurrence, just different story.
- **LC 1137:** N-th Tribonacci Number (3-term recurrence).
- **Matrix Exponentiation:** O(log n) optimization.
- **DP in constant space:** teach memory compression concept.