



LeetCode 509 — Fibonacci Number

1. Problem Title & Link

- **Title:** LeetCode 509 — Fibonacci Number
- **Link:** <https://leetcode.com/problems/fibonacci-number/>

2. Problem Statement (Short Summary)

Return the nth Fibonacci number where:

$$F(0) = 0$$

$$F(1) = 1$$

$$F(n) = F(n-1) + F(n-2) \text{ for } n > 1$$

Given n, return F(n).

3. Examples (Input → Output)

Input: n = 2

Output: 1

Input: n = 3

Output: 2

Input: n = 10

Output: 55

4. Constraints

- $0 \leq n \leq 30$ (on LeetCode this is typical)
- Small n — fits in 32-bit int for given constraints.
- Focus of problem: recursion vs memoization vs iterative DP.

5. Core Concept (Pattern / Topic)

Recurrence / Dynamic Programming (DP)

This is the canonical example of:

- Recurrence relations,
- Overlapping subproblems (use memoization),
- Space-optimized DP (iterative two-variable state).

6. Thought Process (Step-by-Step Explanation)

Approaches (teach in order):



1. **Recursive (Brute force)** — direct, simple, but exponential time $O(2^n)$. Good to show the recurrence tree and motivate optimization.
2. **Top-down Memoization** — store computed values in cache to avoid recomputation. Time $O(n)$, Space $O(n)$ (recursion + memo).
3. **Bottom-up Iterative (Tabulation / Space optimized)** — compute from $0..n$ using two variables. Time $O(n)$, Space $O(1)$. This is the recommended solution for interviews here.

We'll give code for the **iterative (space-optimized)** approach and mention memoized variant as an alternate.

7. Visual / Intuition Diagram

Recurrence tree for $F(4)$ (recursive) — shows repeated subcalls:

x

Memoization collapses repeated nodes; iterative builds values bottom-up:

$F(0)=0, F(1)=1, F(2)=1, F(3)=2, F(4)=3, \dots$

8. Pseudocode (Iterative, Space-Optimized)

```
if n <= 1: return n
a, b = 0, 1 # a = F(0), b = F(1)
for i from 2 to n:
    c = a + b
    a = b
    b = c
return b
```

9. Code Implementation

✓ Python (iterative, optimal)

```
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 (iterative, optimal)

```
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;
}
}

```

10. Time & Space Complexity

Metric	Complexity
Time	$O(n)$ — single loop up to n
Space	$O(1)$ — two variables only (ignoring call stack)

Memoized top-down: Time $O(n)$, Space $O(n)$.

11. Common Mistakes / Edge Cases

- Using naive recursion for larger $n \rightarrow$ exponential time (TLE).
- Off-by-one in loop bounds (ensure loop runs exactly $n-1$ times from initial state).
- Not handling $n=0$ correctly (should return 0).
- Integer overflow only if constraints larger — but for $n \leq 30$ fine. If n were large, consider using long/ BigInteger or matrix exponentiation.

12. Detailed Dry Run (Step-by-Step Table)

Let's do a **detailed dry run** for $n = 10$ (classic example).

We compute iteratively with $a = F(i-2)$, $b = F(i-1)$, and new $c = F(i)$.

Initial:

$i =$ (we start loop at 2)

$a = 0 \ # F(0)$

$b = 1 \ # F(1)$

i	a (F(i-2))	b (F(i-1))	c = a + b (F(i))	after update: a, b
2	0	1	1	a=1, b=1
3	1	1	2	a=1, b=2
4	1	2	3	a=2, b=3
5	2	3	5	a=3, b=5



6	3	5	8	a=5, b=8
7	5	8	13	a=8, b=13
8	8	13	21	a=13, b=21
9	13	21	34	a=21, b=34
10	21	34	55	a=34, b=55

After finishing loop, $b = 55 \rightarrow$ return 55. ✓

Also small dry runs:

- $n=0 \rightarrow$ return 0
- $n=1 \rightarrow$ return 1
- $n=2 \rightarrow$ loop runs once \rightarrow return 1

13. Common Use Cases (Real-Life / Interview)

- Teaching recurrence relations and DP fundamentals.
- Modeling growth patterns (rabbit problem, naive models).
- Basis for dynamic programming pattern recognition (overlapping subproblems).
- Leads into matrix exponentiation / fast doubling for very large n .

14. Common Traps

- Thinking recursion is acceptable for larger n — leads to exponential blowup.
- Not noticing that Fibonacci can be computed in $O(\log n)$ via matrix exponentiation or fast doubling (advanced).
- Using floating point or Binet formula for integer Fibonacci — precision issues.

15. Builds To (Related Problems)

- **LC 70** — Climbing Stairs (same recurrence)
- **LC 746** — Min Cost Climbing Stairs (DP)
- **LC 509 variants**: Tribonacci, DP generalizations
- Matrix exponentiation / fast doubling (for large n)

16. Alternate Approaches + Comparison

Approach	Time	Space	Notes
Naive Recursion	$O(2^n)$	$O(n)$	Educational only
Memoization (top-down)	$O(n)$	$O(n)$	Simple and fast
Iterative DP (bottom-up)	$O(n)$	$O(1)$	Best for this problem
Fast doubling / Matrix expo	$O(\log n)$	$O(1)$	For huge n (not needed for constraints)



17. Why This Solution Works (Short Intuition)

The Fibonacci recurrence uses only the two previous states. By iteratively propagating these two states forward, we compute $F(n)$ in linear time with constant memory.

18. Variations / Follow-Up Questions

- Compute Fibonacci modulo m (use fast doubling for very large n).
- Count ways problems (e.g., climbing stairs) map directly to Fibonacci.
- Use matrix exponentiation to compute $F(n)$ in $O(\log n)$ time for very large n .
- Implement bottom-up table and return full sequence (useful for other problems).