📓 Documentation: Fibonacci Number (Leetcode 509)

1. **Problem Statement**

   Given an integer n, return the nth number in the Fibonacci sequence.

   The Fibonacci sequence is defined as:

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

   Constraints:

   - 0 <= n <= 30

2. **Intuition**

   The Fibonacci sequence builds on the idea of recurrence—each number is the sum of the two before it. Since the value at F(n) depends only on F(n-1) and F(n-2), we can compute values iteratively using just two variables, making the solution both time and space efficient.

3. **Key Observations**

   - The sequence starts from 0 and 1.
   - Every next number is determined solely by the previous two numbers.
   - We don't need the entire sequence stored—just the last two values.
   - Brute-force recursive solutions are highly inefficient due to overlapping subproblems.

4. Approach

We use an iterative (bottom-up) dynamic programming approach with two variables:

- If n is 0, return 0.
- If n is 1, return 1.
- Start from the third number, compute each Fibonacci value up to n using two variables that store the last two computed values.
- Return the final computed value.

5. Edge Cases

- n = 0 → Output: 0
- n = 1 → Output: 1
- The function handles all values from 0 to 30 as per constraints.

6. Complexity Analysis

🕒 Time Complexity

- O(n): We loop through from 2 to n once.

🖫 Space Complexity

- O(1): Only two variables are used, regardless of n.

7. Alternative Approaches

a) Recursive Solution (Brute-force)

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

- ❌ Time Complexity: O(2^n) — exponential
- ❌ Inefficient for large n due to repeated calculations

b) Memoization (Top-down DP)

```
def fib(n, memo = {}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib(n-1, memo) + fib(n-2, memo)
    return memo[n]
```

- ✅ Efficient with O(n) time
- 🎁 Uses O(n) space due to the dictionary

c) Bottom-Up with Array

```
def fib(n):
    if n <= 1:
        return n
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n+1):
        dp[i] = dp[i-1] + dp[i-2]
```

return dp[n]

- ✅ Time: O(n)
- ❌ Space: O(n)

8. Test Cases

Input	Expected Output	Explanation
0	0	F(0) = 0
1	1	F(1) = 1
2	1	F(2) = F(1) + F(0) = 1 + 0
3	2	F(3) = F(2) + F(1) = 1 + 1
4	3	F(4) = F(3) + F(2) = 2 + 1
10	55	Computed iteratively
30	832040	Edge case at upper constraint

9. Final Thoughts

- The iterative method is best for small n with minimal memory use.
- Recursive solutions without memoization should be avoided due to performance issues.
- For larger Fibonacci problems (e.g., $n > 10^5$), matrix exponentiation or Binet's Formula may be considered.
- This problem is a classic example of optimizing recursive relationships using dynamic programming principles.