

■ Fibonacci Number

Where recursion is born — the classic self-referential sequence

● Recursion

■ EASY

■ What You Will Learn

- What recursion really means — a function that calls itself
- The two pillars of every recursive solution: base case & recursive case
- How the call stack works during recursion
- Why naive recursion is slow and how memoization fixes it
- The difference between top-down (recursion) and bottom-up (DP)

■ Intuition First

Fibonacci: each number is the sum of the two before it. 0,1,1,2,3,5,8,13,21... To find fib(5), you need fib(4) and fib(3). To find fib(4), you need fib(3) and fib(2). It's like asking 'what comes before?' over and over until you hit something you already know (fib(0)=0, fib(1)=1).

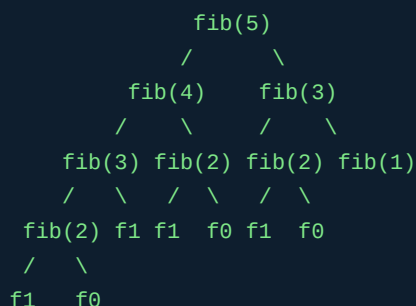
Real-world twin: Rabbits! A pair of rabbits takes a month to mature. After that, they produce one pair per month. The count of rabbit pairs each month follows the Fibonacci sequence exactly.

■ Key Insight

Every recursive solution needs TWO things: (1) A base case — the simplest answer you know without calling yourself. (2) A recursive case — how to break the big problem into smaller ones of the same type.

■ Visual — The Recursion Tree for fib(5)

Each node calls two children. The tree keeps splitting until it hits fib(0) or fib(1).



Base cases (leaves): fib(0)=0, fib(1)=1
Notice fib(3) is computed TWICE, fib(2) THREE times!
This is the redundancy memoization eliminates.

■ ■ How It Works — Step by Step

1. Call `fib(n)`. If `n` is 0 or 1 → return `n` immediately (base case)
2. Otherwise, recursively call `fib(n-1)` and `fib(n-2)`
3. Java pushes each call onto the call stack (like stacking plates)
4. Once a base case is hit, it returns its value up the chain
5. Each level adds the two returned values and passes the sum upward

■ Dry Run — `fib(4)`

```
fib(4)
→ fib(3) + fib(2)
  → fib(2) + fib(1)      → fib(1) + fib(0)
    → fib(1) + fib(0)      → 1          → 0
      → 1          → 0
        returns 1
      fib(2) = 1+0 = 1
    fib(3) = 1+1 = 2      fib(2) = 1+0 = 1
  fib(4) = 2+1 = 3 ✓
```

■ Java Code — Naive Recursion

```
1  class Solution {
2  public int fib(int n) {
3    // BASE CASE: we know these answers directly
4    if (n == 0) return 0;
5    if (n == 1) return 1;
6
7    // RECURSIVE CASE: break into smaller subproblems
8    return fib(n - 1) + fib(n - 2);
9  }
10 }
```

■ Java Code — Memoized (Fast!)

```
1  class Solution {
2  int[] memo = new int[101]; // Cache results
3
4  public int fib(int n) {
5    if (n == 0) return 0;
6    if (n == 1) return 1;
7    if (memo[n] != 0) return memo[n]; // Already computed!
8
9    memo[n] = fib(n-1) + fib(n-2); // Compute & cache
10   return memo[n];
11 }
12 }
```

Time & Space Complexity

Type	Value	What It Means
Time	$O(2^n)$ naive / $O(n)$ memo	Naive: tree doubles every level. Memo: each n computed once.
Space	$O(n)$ call stack	Call stack depth = n . Memo array size = n .

Common Mistakes

Missing Base Case

Forgetting `if(n==0)` or `if(n==1)` causes infinite recursion and `StackOverflowError`. ALWAYS write base cases FIRST before the recursive call.

Off-by-one

Some definitions have `fib(1)=1`, `fib(2)=1`. Others have `fib(0)=0`, `fib(1)=1`. Read the problem carefully to know which starting point they use.

Pro Tips

Bottom-up DP

For interviews, the iterative bottom-up version is most impressive: `int a=0,b=1; for(int i=2;i<=n;i++){int c=a+b;a=b;b=c;} return b;` — $O(n)$ time, $O(1)$ space!

Think Like This

Every recursive call is a question: 'What's the answer for a smaller version?' Base cases are the answers you know without asking. The magic is trust — trust that smaller calls return correctly.

Practice Problems

#	Problem	Diff
1	Fibonacci Number (LeetCode 509)	Easy
2	Climbing Stairs (LeetCode 70)	Easy
3	Tribonacci Number (LeetCode 1137)	Easy
4	House Robber (LeetCode 198)	Medium

Quick Revision



`fib(n) = fib(n-1) + fib(n-2); base: fib(0)=0, fib(1)=1`

✓	Naive: $O(2^n)$ time — exponential, very slow
✓	Memoization: cache results → $O(n)$ time
✓	Bottom-up DP: $O(n)$ time, $O(1)$ space — best!
✓	Always write BASE CASE first in any recursive function

■ Visual Thinking

Imagine the recursion tree as an upside-down family tree. $\text{fib}(5)$ is the grandparent. It needs two parents — $\text{fib}(4)$ and $\text{fib}(3)$. Each of those needs their own parents. The tree keeps growing until every branch hits $\text{fib}(0)$ or $\text{fib}(1)$ — the great-great-great-grandparents who just KNOW their answer. Memoization is like writing the answer on a sticky note so you never ask the same grandparent twice.