🥰

Recursion

🔄 Mastering Recursion for FAANG Interviews

> The Complete Deep-Dive Guide: From Intuition to Optimal Solutions

---

PART 1: CONCEPTUAL FOUNDATION

1.1 Core Intuition

What is Recursion Really Asking?

Recursion is a problem-solving paradigm where a function **calls itself** to solve smaller instances of the same problem. At its heart, recursion answers this question:

> "Can I break this problem into smaller versions of itself until I reach a trivially solvable case?"

Every recursive solution has two essential components:

Component	Purpose	Example (Factorial)
Base Case	The stopping condition; the simplest instance that can be solved directly	`if (n <= 1) return 1`
Recursive Case	Breaks the problem into smaller subproblems and combines their solutions	`return n * factorial(n-1)`

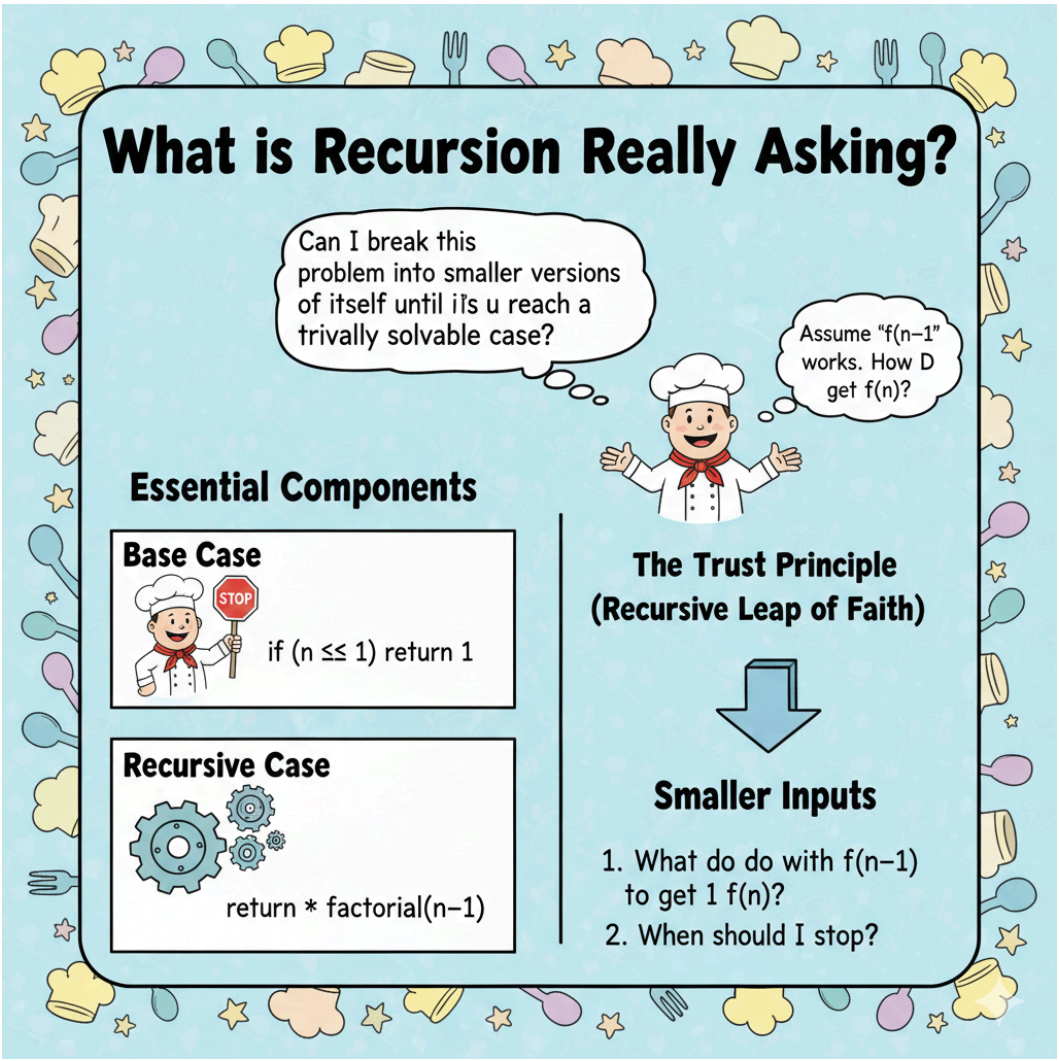
The Trust Principle (Recursive Leap of Faith)

The most important mental model for recursion:

> "Assume your recursive call works correctly for smaller inputs, then use that result to solve the current problem."

You don't need to trace through every recursive call mentally. Trust that `f(n-1)` returns the correct answer, and focus only on:

1. What do I do with the result of `f(n-1)` to get `f(n)`?
2. When should I stop recursing?



Real-World Analogies

🪆 Russian Nesting Dolls (Matryoshka)

```
To find the smallest doll:
  1. Open the current doll
  2. If there's another doll inside → repeat with inner doll
  3. If no doll inside → you found the smallest (BASE CASE)
```

🪞 Two Mirrors Facing Each Other

```
Each reflection contains another reflection,
creating infinite depth — like infinite recursion without a base case!
```

📂 Filesystem Navigation

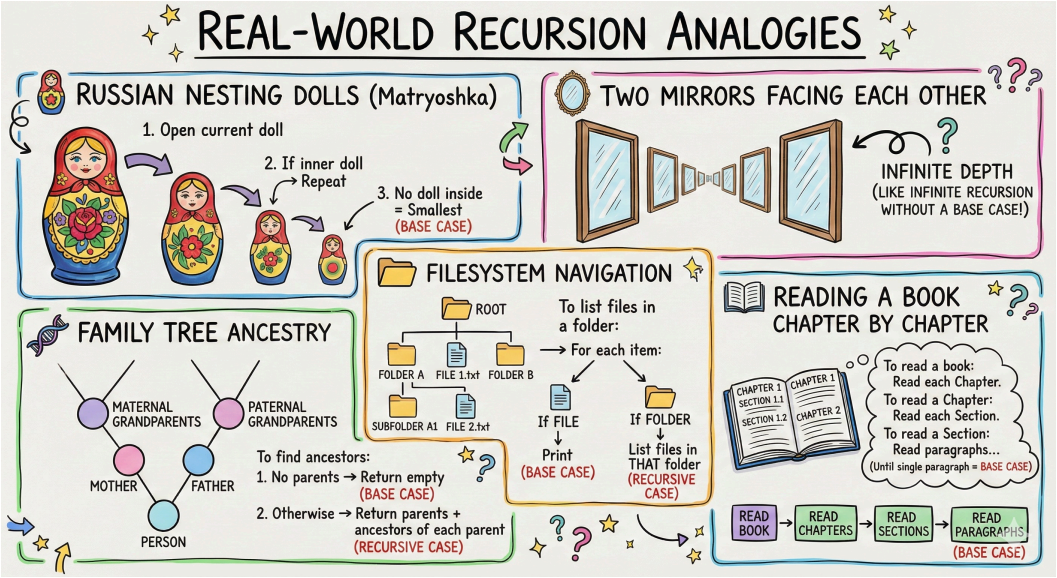
```
To list all files in a folder:
  1. For each item in current folder:
     - If it's a file → print it (BASE CASE)
     - If it's a folder → list all files in THAT folder (RECURSIVE CASE)
```

🧬 Family Tree Ancestry

```
To find all ancestors of a person:
  1. If person has no parents → return empty (BASE CASE)
  2. Otherwise → return parents + ancestors of each parent (RECURSIVE CASE)
```

📖 Reading a Book Chapter by Chapter

```
To read a book:
  1. If no chapters left → done (BASE CASE)
  2. Otherwise → read chapter 1, then read remaining chapters (RECURSIVE CASE)
```



Why Does This Pattern Exist?

Recursion exists because many problems have **self-similar structure**. The same logic that solves the whole problem also solves its parts.

Classes of Problems Recursion Solves:

Problem Class	Why Recursion Fits	Examples
Divide & Conquer	Problem splits into independent subproblems	Merge Sort, Quick Sort, Binary Search
Tree/Graph Traversal	Each node's children form subtrees	DFS, Tree Height, Path Finding
Backtracking	Explore choices, undo if wrong	N-Queens, Sudoku, Permutations

Problem Class	Why Recursion Fits	Examples
Dynamic Programming	Overlapping subproblems with optimal substructure	Fibonacci, Knapsack, Edit Distance
Mathematical Sequences	Defined in terms of previous terms	Factorial, Power, GCD
Combinatorics	Generate all possibilities systematically	Subsets, Combinations, Parentheses

1.2 Pattern Recognition

How Do I Identify When to Use Recursion?

The Recursion Checklist:

```
✅ Can the problem be broken into SMALLER INSTANCES of itself?
✅ Is there a CLEAR BASE CASE (trivially solvable smallest instance)?
✅ Does combining subproblem solutions give the overall solution?
✅ Does the problem involve HIERARCHICAL or NESTED structures?
```

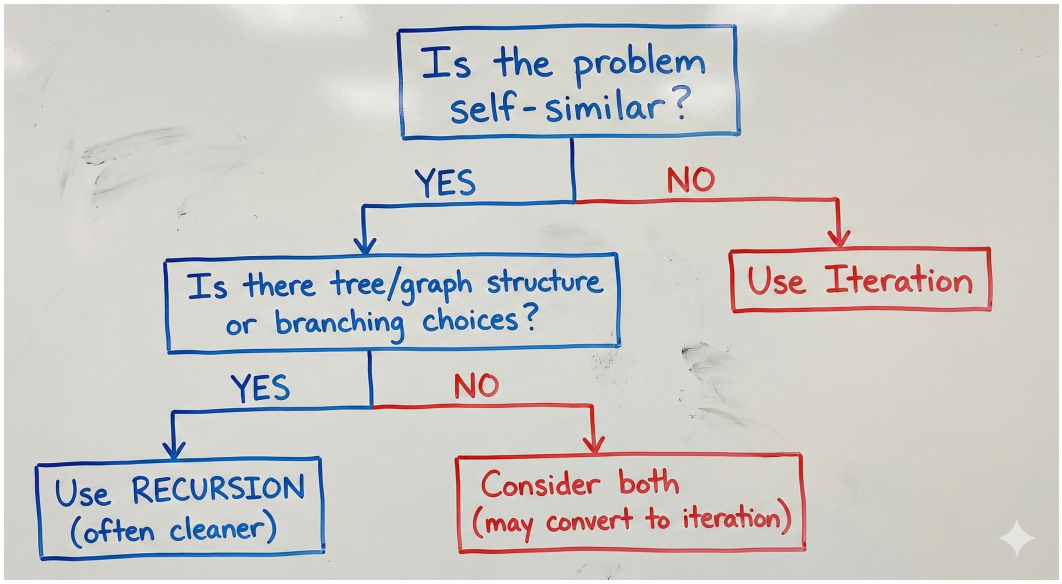
Key Problem Characteristics

Characteristic	Why It Suggests Recursion
Hierarchical data (trees, graphs)	Natural parent-child relationships
"All possibilities" or "all combinations"	Systematic exploration via branching
Mathematical recurrence relation	`f(n)` defined in terms of `f(n-1)`, `f(n-2)`, etc.
Divide and conquer feasibility	Problem can be split and merged
Self-similar structure	Subproblems look like the original

Trigger Words in Problem Statements

```
🚩 "Find ALL..."          → Backtracking/Exhaustive search
🚩 "Generate ALL..."      → Recursive enumeration
🚩 "Count the number of ways..." → Recursive counting with DP potential
🚩 "Minimum/Maximum..."   → DP with recursion + memoization
🚩 "Tree" or "Graph"      → DFS/Recursive traversal
🚩 "Nested" or "Balanced" → Stack-based or recursive structure
🚩 "Subsets/Subsequences" → Include/exclude pattern
🚩 "Parentheses"          → Recursive generation
🚩 "Path from root to..." → Tree recursion
```

Decision Framework: Recursion vs Iteration



PART 2: SOLUTION PROGRESSION

We'll use **Fibonacci Numbers** as our canonical example because it perfectly demonstrates the evolution from naive recursion to optimal solutions.

Problem: Compute the nth Fibonacci number where:

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

Approach 1: Pure Recursion (Brute Force)

A. Intuition & Strategy

The most direct translation of the mathematical definition into code. We literally implement the recurrence relation as-is:

- Base case: Return `0` if `n=0`, return `1` if `n=1`
- Recursive case: Return `fib(n-1) + fib(n-2)`

Why it works: The mathematical definition IS a recursive definition. We're just encoding it directly.

B. Pseudocode

```
FUNCTION fibonacci(n):
    // Base Cases
    IF n == 0:
        RETURN 0
    IF n == 1:
        RETURN 1
```

```
    // Recursive Case
    RETURN fibonacci(n - 1) + fibonacci(n - 2)
```

C. Working Code (JavaScript)

```javascript
/**
 * Pure Recursive Fibonacci
 * Time: O(2^n) - Exponential
 * Space: O(n) - Recursion depth
 */
function fibonacciRecursive(n) {
    // Base Cases: F(0) = 0, F(1) = 1
    if (n === 0) return 0;
    if (n === 1) return 1;

    // Recursive Case: F(n) = F(n-1) + F(n-2)
    return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);
}

// Test
console.log(fibonacciRecursive(10)); // Output: 55
```

D. Complexity Analysis

Time Complexity: O(2ⁿ) — Exponential

Mathematical Derivation:

Let `T(n)` = number of function calls to compute `fib(n)`

```
T(0) = 1                    (just returns 0)
T(1) = 1                    (just returns 1)
T(n) = T(n-1) + T(n-2) + 1  (calls fib(n-1), fib(n-2), plus itself)
```

This recurrence is similar to Fibonacci itself! The solution is:

```
T(n) ≈ 2 × F(n+1) - 1

Since F(n) ≈ φⁿ / √5  where φ = (1+√5)/2 ≈ 1.618 (golden ratio)

Therefore: T(n) = O(φⁿ) ≈ O(1.618ⁿ) ⊂ O(2ⁿ)
```

More intuitively: Each call branches into 2 calls, depth is ~n → roughly 2^n calls.

Space Complexity: O(n) — Linear

```
Maximum recursion depth = n
Each stack frame stores: return address, parameter n, local variables
Space = O(n) for the call stack
```

E. Dry Run

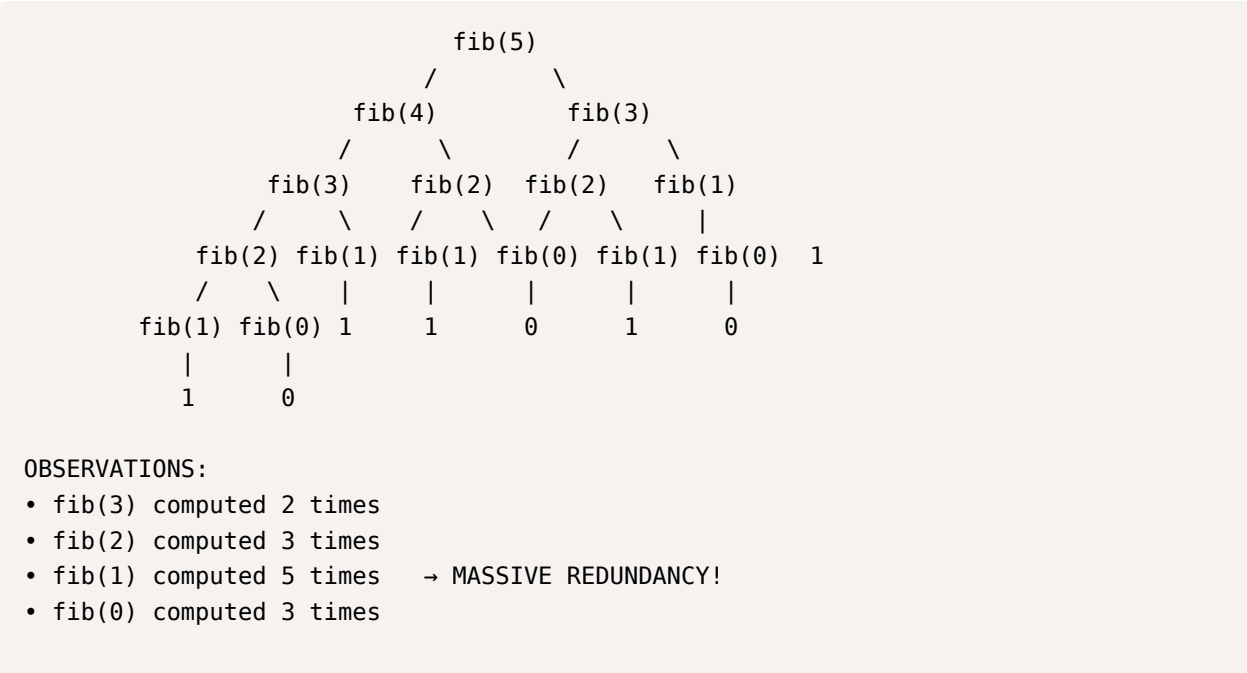
Input: `n = 5`

Call Stack Trace (Depth-First):

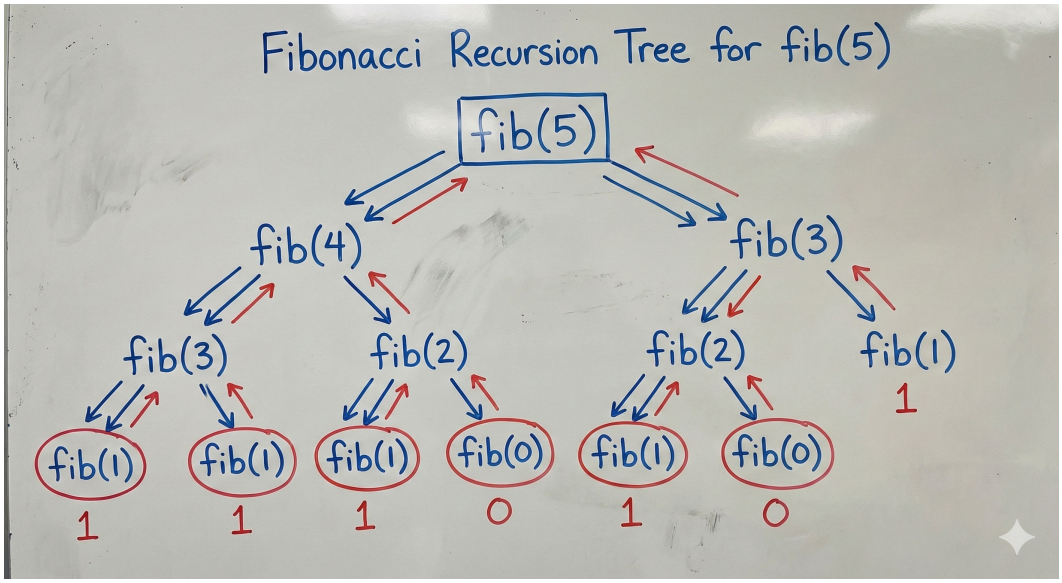
Step	Call	n	Action	Return Value
1	fib(5)	5	Calls fib(4), fib(3)	Waiting...
2	fib(4)	4	Calls fib(3), fib(2)	Waiting...
3	fib(3)	3	Calls fib(2), fib(1)	Waiting...
4	fib(2)	2	Calls fib(1), fib(0)	Waiting...
5	fib(1)	1	**BASE CASE**	**Returns 1**
6	fib(0)	0	**BASE CASE**	**Returns 0**
7	fib(2)	2	Has both results: 1+0	**Returns 1**
8	fib(1)	1	**BASE CASE**	**Returns 1**
9	fib(3)	3	Has both results: 1+1	**Returns 2**
10	fib(2)	2	Calls fib(1), fib(0)	Waiting...
11	fib(1)	1	**BASE CASE**	**Returns 1**
12	fib(0)	0	**BASE CASE**	**Returns 0**
13	fib(2)	2	Has both results: 1+0	**Returns 1**
14	fib(4)	4	Has both results: 2+1	**Returns 3**
15	fib(3)	3	Calls fib(2), fib(1)	Waiting...
...	...	...	(Repeats similar pattern)	...
Final	fib(5)	5	Has both results: 3+2	**Returns 5**

Total calls for fib(5): 15 calls

F. Visual Diagram — Recursion Tree



```
This is why pure recursion is O(2ⁿ) — exponential waste!
```



Approach 2: Recursion with Memoization (Top-Down DP)

A. Intuition & Strategy

The pure recursive approach recalculates the same values many times (see tree above). **Memoization** stores results of expensive function calls and returns the cached result when the same inputs occur again.

Core idea:

- Keep a "memo" (cache/dictionary) of already-computed values
- Before computing `fib(n)`, check if it's in the memo
- After computing `fib(n)`, store it in the memo

Why it works: Each unique subproblem is solved only ONCE. Subsequent calls just look up the answer.

B. Pseudocode

```
MEMO = {}  // Global cache

FUNCTION fibMemo(n):
    // Check cache first
    IF n IN MEMO:
        RETURN MEMO[n]

    // Base Cases
    IF n == 0:
        RETURN 0
    IF n == 1:
```

```
        RETURN 1

    // Compute and cache
    RESULT = fibMemo(n - 1) + fibMemo(n - 2)
    MEMO[n] = RESULT

    RETURN RESULT
```

C. Working Code (JavaScript)

```javascript
/**
 * Memoized Recursive Fibonacci (Top-Down DP)
 * Time: O(n) - Each subproblem solved once
 * Space: O(n) - Memo table + recursion stack
 */
function fibonacciMemo(n, memo = {}) {
    // Check cache first
    if (n in memo) return memo[n];

    // Base Cases
    if (n === 0) return 0;
    if (n === 1) return 1;

    // Compute, cache, and return
    memo[n] = fibonacciMemo(n - 1, memo) + fibonacciMemo(n - 2, memo);
    return memo[n];
}

// Alternative: Using closure for cleaner API
function fibonacciMemoClean(n) {
    const memo = { 0: 0, 1: 1 };

    function fib(n) {
        if (n in memo) return memo[n];
        memo[n] = fib(n - 1) + fib(n - 2);
        return memo[n];
    }

    return fib(n);
}

// Test
console.log(fibonacciMemo(50)); // Output: 12586269025 (instant!)
```

D. Complexity Analysis

Time Complexity: O(n) — Linear

Mathematical Derivation:

```
• There are exactly (n + 1) unique subproblems: fib(0), fib(1), ..., fib(n)
• Each subproblem is computed exactly ONCE (subsequent calls are O(1) lookups)
• Each computation does O(1) work (addition + hash operations)

Total time = (n + 1) × O(1) = O(n)
```

Space Complexity: O(n) — Linear

```
• Memo table stores (n + 1) entries → O(n)
• Maximum recursion depth = n → O(n)
• Total space = O(n) + O(n) = O(n)
```

E. Dry Run

Input: `n = 5`, `memo = {}`

Step	Call	n	memo (before)	Action	memo (after)	Return
1	fib(5)	5	{}	Not in memo, compute	—	Waiting
2	fib(4)	4	{}	Not in memo, compute	—	Waiting
3	fib(3)	3	{}	Not in memo, compute	—	Waiting
4	fib(2)	2	{}	Not in memo, compute	—	Waiting
5	fib(1)	1	{}	**BASE CASE**	{}	**1**
6	fib(0)	0	{}	**BASE CASE**	{}	**0**
7	fib(2)	2	{}	1 + 0 = 1, cache it	{2: 1}	**1**
8	fib(1)	1	{2:1}	**BASE CASE**	{2:1}	**1**
9	fib(3)	3	{2:1}	1 + 1 = 2, cache it	{2:1, 3:2}	**2**
10	fib(2)	2	{2:1, 3:2}	**CACHE HIT!**	{2:1, 3:2}	**1**
11	fib(4)	4	{2:1, 3:2}	2 + 1 = 3, cache	{2:1, 3:2, 4:3}	**3**
12	fib(3)	3	{2:1, 3:2, 4:3}	**CACHE HIT!**	same	**2**
13	fib(5)	5	{2:1, 3:2, 4:3}	3 + 2 = 5, cache	{2:1, 3:2, 4:3, 5:5}	**5**

Total unique computations: 6 (vs 15 for pure recursion)

F. Visual Diagram — Pruned Recursion Tree

```
                    fib(5)
                   /      \
              fib(4)       fib(3) ← CACHE HIT (returns 2)
```

```
                    /      \
                fib(3)    fib(2) ← CACHE HIT (returns 1)
               /     \
           fib(2)   fib(1)
           /    \      |
       fib(1) fib(0)   1
          |      |
          1      0

MEMOIZATION EFFECT:
• Gray branches are NEVER computed (cache hits)
• Only the leftmost "spine" is fully computed
• Total calls: O(n) instead of O(2ⁿ)

           Pure Recursion          With Memoization
fib(5):         15 calls      →           9 calls
fib(10):       177 calls      →          19 calls
fib(20):    21,891 calls      →          39 calls
fib(50):   40+ BILLION        →          99 calls
```

Approach 3: Iterative (Bottom-Up DP) — Optimal

A. Intuition & Strategy

Instead of starting from `fib(n)` and working down (top-down), we start from `fib(0)` and work UP (bottom-up). We build the solution iteratively, using only the previous two values.

Core idea:

- We only need `fib(n-1)` and `fib(n-2)` to compute `fib(n)`
- Store just these two values in variables
- Update them as we iterate from 0 to n

Why it works: Eliminates recursion overhead entirely. Uses the recurrence relation directly in a loop.

B. Pseudocode

```
FUNCTION fibIterative(n):
    // Handle base cases
    IF n == 0: RETURN 0
    IF n == 1: RETURN 1

    // Initialize first two values
    prev2 = 0  // fib(i-2)
    prev1 = 1  // fib(i-1)

    // Build up from fib(2) to fib(n)
    FOR i FROM 2 TO n:
        current = prev1 + prev2
```

```
        prev2 = prev1
        prev1 = current

    RETURN prev1
```

C. Working Code (JavaScript)

```javascript
/**
 * Iterative Fibonacci (Bottom-Up DP, Space Optimized)
 * Time: O(n) - Single pass
 * Space: O(1) - Only two variables
 */
function fibonacciIterative(n) {
    // Base cases
    if (n === 0) return 0;
    if (n === 1) return 1;

    // Only need previous two values
    let prev2 = 0;  // fib(i-2), starts as fib(0)
    let prev1 = 1;  // fib(i-1), starts as fib(1)

    // Build up from fib(2) to fib(n)
    for (let i = 2; i <= n; i++) {
        const current = prev1 + prev2;
        prev2 = prev1;
        prev1 = current;
    }

    return prev1;
}

// Alternative: Using array (clearer but O(n) space)
function fibonacciIterativeArray(n) {
    if (n <= 1) return n;

    const dp = [0, 1];
    for (let i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}

// Test
console.log(fibonacciIterative(50)); // Output: 12586269025
```

D. Complexity Analysis

Time Complexity: O(n) — Linear

```
• Single loop from 2 to n → (n - 1) iterations
• Each iteration does O(1) work (addition + assignment)

Total time = O(n)
```

Space Complexity: O(1) — Constant

```
• Only storing 3 variables (prev2, prev1, current)
• No recursion stack
• No memo table

Total space = O(1)
```

Comparison Table:

Approach	Time	Space	Recursion Stack?
Pure Recursion	$O(2^n)$	O(n)	Yes, depth n
Memoization	O(n)	O(n)	Yes, depth n
Iterative	**O(n)**	**O(1)**	**No**

E. Dry Run

Input: `n = 5`

Iteration (i)	prev2 (fib[i-2])	prev1 (fib[i-1])	current (fib[i])	After Update
Initial	0	1	—	—
i = 2	0	1	0 + 1 = **1**	prev2=1, prev1=1
i = 3	1	1	1 + 1 = **2**	prev2=1, prev1=2
i = 4	1	2	2 + 1 = **3**	prev2=2, prev1=3
i = 5	2	3	3 + 2 = **5**	prev2=3, prev1=5

Return: `prev1 = 5` ✓

F. Visual Diagram — State Transition

```
Index:     0     1     2     3     4     5
Values:    0     1     1     2     3     5
           ↑     ↑
         prev2  prev1  (initial state)

Step 1 (i=2):  0  +  1  =  1
                  ↑     ↑
                prev2  prev1  →  current

Step 2 (i=3):        1  +  1  =  2
                        ↑     ↑
                      prev2  prev1  →  current
```

```
Step 3 (i=4):              1  +  2  =  3
                              ↑     ↑
                            prev2  prev1  →  current

Step 4 (i=5):                 2  +  3  =  5
                                 ↑     ↑
                               prev2  prev1  →  current

SLIDING WINDOW of size 2 moves through the sequence!
```

Approach 4: Matrix Exponentiation — O(log n) [Advanced]

A. Intuition & Strategy

This is the most optimal approach for very large `n`. It exploits the fact that Fibonacci numbers can be computed using matrix multiplication:

```
| F(n+1)  F(n)   |   | 1  1 |ⁿ
| F(n)    F(n-1) | = | 1  0 |
```

Using **fast exponentiation** (repeatedly squaring), we can compute the matrix power in O(log n) time.

B. Why It Works

Starting from the identity:

```
| F(2) |   | 1  1 | | F(1) |   | 1  1 | | 1 |
| F(1) | = | 1  0 | | F(0) | = | 1  0 | | 0 |
```

Generalizing:

```
| F(n+1) |   | 1  1 |ⁿ | 1 |
| F(n)   | = | 1  0 |  | 0 |
```

So `F(n)` is the element at position [1][0] (or [0][1]) of the matrix `[[1,1],[1,0]]^n`.

C. Working Code (JavaScript)

```javascript
/**
 * Matrix Exponentiation Fibonacci
 * Time: O(log n) - Fast matrix exponentiation
 * Space: O(log n) - Recursion stack for power function
 */
function fibonacciMatrix(n) {
    if (n === 0) return 0;
    if (n === 1) return 1;

    // Multiply two 2x2 matrices
    function multiply(A, B) {
```

```
        return [
            [
                A[0][0] * B[0][0] + A[0][1] * B[1][0],
                A[0][0] * B[0][1] + A[0][1] * B[1][1]
            ],
            [
                A[1][0] * B[0][0] + A[1][1] * B[1][0],
                A[1][0] * B[0][1] + A[1][1] * B[1][1]
            ]
        ];
    }

    // Fast matrix exponentiation using repeated squaring
    function matrixPower(M, n) {
        // Base case: M^1 = M
        if (n === 1) return M;

        // If n is even: M^n = (M^(n/2))^2
        if (n % 2 === 0) {
            const half = matrixPower(M, n / 2);
            return multiply(half, half);
        }

        // If n is odd: M^n = M * M^(n-1)
        return multiply(M, matrixPower(M, n - 1));
    }

    const baseMatrix = [[1, 1], [1, 0]];
    const result = matrixPower(baseMatrix, n);

    return result[0][1]; // F(n) is at position [0][1] or [1][0]
}

// Test
console.log(fibonacciMatrix(50)); // Output: 12586269025
```

D. Complexity Analysis

Time Complexity: O(log n)

```
• Matrix exponentiation uses repeated squaring
• Number of multiplications = O(log n)
• Each multiplication is O(1) for 2x2 matrices (8 multiplications, 4 additions)

Total time = O(log n)
```

Space Complexity: O(log n)

```
• Recursion depth for matrixPower = O(log n)
• Each stack frame stores a 2x2 matrix = O(1)

Total space = O(log n) for recursion stack
```

E. Dry Run

Input: `n = 10`

```
matrixPower(M, 10)
  → 10 is even: compute matrixPower(M, 5), then square

  matrixPower(M, 5)
    → 5 is odd: compute M × matrixPower(M, 4)

    matrixPower(M, 4)
      → 4 is even: compute matrixPower(M, 2), then square

      matrixPower(M, 2)
        → 2 is even: compute matrixPower(M, 1), then square

        matrixPower(M, 1)
          → BASE CASE: return [[1,1],[1,0]]

        Return: [[1,1],[1,0]]² = [[2,1],[1,1]]  (M²)

      Return: [[2,1],[1,1]]² = [[5,3],[3,2]]  (M⁴)

    Return: M × [[5,3],[3,2]] = [[8,5],[5,3]]  (M⁵)

  Return: [[8,5],[5,3]]² = [[89,55],[55,34]]  (M¹⁰)

F(10) = result[0][1] = 55 ✓
```

Only 4 matrix multiplications instead of 10 iterations!

PART 3: EDGE CASES & GOTCHAS

Critical Edge Cases to Handle

Edge Case	Input	Expected	Why It Matters
Zero	`n = 0`	`0`	Base case, often forgotten
One	`n = 1`	`1`	Second base case
Negative	`n = -5`	Error/0	Invalid input, handle gracefully
Large n	`n = 10000`	Huge number	Stack overflow risk, BigInt needed
Empty input	`n = undefined`	Error	Validate inputs

Common Mistakes in Interviews

```
❌ MISTAKE 1: Forgetting base cases
   function fib(n) {
     return fib(n-1) + fib(n-2);  // Infinite recursion!
   }

❌ MISTAKE 2: Wrong base case values
   if (n <= 1) return 1;  // Wrong! fib(0) should be 0

❌ MISTAKE 3: Off-by-one errors in loops
   for (let i = 2; i < n; i++)  // Wrong! Should be i <= n

❌ MISTAKE 4: Not handling negative inputs
   function fib(n) {
     if (n <= 1) return n;  // Returns negative for negative n!
   }

❌ MISTAKE 5: Integer overflow (JS is safe, other languages not)
   // In Java/C++: int overflow for large n
   // Solution: Use long/BigInt

❌ MISTAKE 6: Stack overflow for deep recursion
   fib(100000);  // Stack overflow without iterative solution
```

Boundary Conditions

```javascript
// Robust Fibonacci with all edge cases handled
function fibonacciRobust(n) {
    // Input validation
    if (typeof n !== 'number' || !Number.isInteger(n)) {
        throw new Error('Input must be an integer');
    }
    if (n < 0) {
        throw new Error('Input must be non-negative');
    }

    // Base cases
    if (n === 0) return 0n;  // Using BigInt for large numbers
    if (n === 1) return 1n;

    // Iterative solution (handles large n)
    let prev2 = 0n, prev1 = 1n;
    for (let i = 2; i <= n; i++) {
        const current = prev1 + prev2;
        prev2 = prev1;
        prev1 = current;
    }
```

```
    return prev1;
}
```

PART 4: INTERVIEW TOOLKIT

4.1 The 30-Second Verbal Explanation

For Pure Recursion:

> "Fibonacci is defined recursively: F(n) equals F(n-1) plus F(n-2), with base cases F(0)=0 and F(1)=1. The pure recursive approach directly implements this definition. However, it has exponential time complexity because we recompute the same subproblems multiple times."

For Memoization:

> "To optimize, I use memoization — I cache results of subproblems as I compute them. Before calculating F(n), I check if it's in my memo. This brings time complexity down to O(n) since each subproblem is solved exactly once, though we still use O(n) space for the cache and call stack."

For Iterative:

> "The most optimal approach is iterative bottom-up. Since F(n) only depends on the previous two values, I maintain just two variables and update them as I loop from 0 to n. This gives O(n) time and O(1) space — no recursion overhead."

4.2 Complexity One-Liners

Approach	One-Liner
Pure Recursion	"This runs in O(2^n) time and O(n) space because each call branches into two more calls, creating an exponential tree of depth n."
Memoization	"This runs in O(n) time and O(n) space because we solve each of the n subproblems exactly once and store them in a hash table."
Iterative	"This runs in O(n) time and O(1) space because we make a single pass using only two variables to track the previous values."
Matrix Exp.	"This runs in O(log n) time because we use fast matrix exponentiation with repeated squaring."

4.3 Follow-Up Questions & Variations

Follow-Up	How Solution Changes
"What if n can be very large (10^18)?"	Use matrix exponentiation O(log n)
"What if we need modulo arithmetic?"	Add `% MOD` to all additions to prevent overflow
"Can you do it without recursion?"	Iterative approach or explicit stack
"What's the space-time tradeoff?"	Memo uses O(n) space for O(n) time; iterative uses O(1) space
"Generalize to Tribonacci?"	Same pattern, track 3 previous values instead of 2
"Find nth term of any linear recurrence?"	Matrix exponentiation works for any linear recurrence
"Count paths in a grid?"	Same DP structure: `paths[i][j] = paths[i-1][j] + paths[i][j-1]`

4.4 Trade-off Discussion

```
┌──────────────────────────────────────────────────────────────────┐
│                    TRADE-OFF MATRIX                              │
├──────────────────┬──────────┬──────────┬─────────────────────────┤
│ Approach         │ Time     │ Space    │ Best When...            │
├──────────────────┼──────────┼──────────┼─────────────────────────┤
│ Pure Recursion   │ O(2ⁿ)    │ O(n)     │ Never (educational only)│
│ Memoization      │ O(n)     │ O(n)     │ Top-down thinking helps │
│ Iterative        │ O(n)     │ O(1)     │ Space is constrained    │
│ Matrix Exp.      │ O(log n) │ O(log n) │ n is extremely large    │
└──────────────────┴──────────┴──────────┴─────────────────────────┘

DECISION GUIDE:
• Default choice: Iterative (best space efficiency)
• If problem is naturally top-down: Memoization (easier to reason about)
• If n > 10^9: Matrix exponentiation (necessary for speed)
• If interviewer wants recursion practice: Memoized recursion
```

PART 5: RELATED PROBLEMS

Core Recursion Patterns on LeetCode

#	Problem	Pattern	Key Insight
509	Fibonacci Number	Basic Recursion → DP	Foundation problem
70	Climbing Stairs	Fibonacci variant	`ways(n) = ways(n-1) + ways(n-2)`
746	Min Cost Climbing Stairs	Fibonacci + costs	Add cost dimension
198	House Robber	Linear DP	`rob(i) = max(rob(i-1), rob(i-2) + nums[i])`
213	House Robber II	Circular array	Run twice: exclude first or last
322	Coin Change	Unbounded knapsack	`dp[amount] = min(dp[amount-coin] + 1)`
62	Unique Paths	2D grid DP	`paths[i][j] = paths[i-1][j] + paths[i][j-1]`

Tree Recursion

#	Problem	Pattern	Key Insight
104	Maximum Depth of Binary Tree	Basic tree recursion	`1 + max(depth(left), depth(right))`
226	Invert Binary Tree	Tree transformation	Swap children, then recurse
100	Same Tree	Dual tree recursion	Compare node, then both subtrees
543	Diameter of Binary Tree	Tree recursion + global state	Track max diameter during height calculation
124	Binary Tree Maximum Path Sum	Complex tree recursion	Consider paths through each node

Backtracking (Recursion + State)

#	Problem	Pattern	Key Insight
46	Permutations	Backtracking	Swap and recurse, swap back
78	Subsets	Include/Exclude	Each element: include or exclude
39	Combination Sum	Unbounded backtracking	Can reuse elements
22	Generate Parentheses	Constrained backtracking	Track open/close counts
51	N-Queens	Constraint propagation	Check row, column, diagonals

Divide & Conquer

#	Problem	Pattern	Key Insight
912	Sort an Array (Merge Sort)	Divide, solve, merge	Split, sort halves, merge
215	Kth Largest Element	Quick Select	Partition around pivot
23	Merge K Sorted Lists	Divide & Conquer	Pair-wise merging
50	Pow(x, n)	Fast exponentiation	x^n = (x^(n/2))^2

APPENDIX: Recursion Templates

Template 1: Basic Recursion

```
function recursiveFunction(input) {
    // 1. BASE CASE(S) - when to stop
    if (baseCondition) {
        return baseValue;
    }

    // 2. RECURSIVE CASE - break into smaller problem
    const subproblemResult = recursiveFunction(smallerInput);

    // 3. COMBINE - use subproblem result to solve current
    return combine(subproblemResult);
}
```

Template 2: Memoization

```
function memoizedFunction(input, memo = new Map()) {
    // 1. CHECK CACHE
    if (memo.has(input)) {
        return memo.get(input);
    }

    // 2. BASE CASE(S)
    if (baseCondition) {
        return baseValue;
    }
```

```
    // 3. COMPUTE RECURSIVELY
    const result = /* recursive computation */;

    // 4. CACHE AND RETURN
    memo.set(input, result);
    return result;
}
```

Template 3: Backtracking

```
function backtrack(state, choices, result) {
    // 1. BASE CASE - found valid solution
    if (isComplete(state)) {
        result.push([...state]);  // Store copy!
        return;
    }

    // 2. TRY EACH CHOICE
    for (const choice of choices) {
        // 3. CHECK CONSTRAINTS (pruning)
        if (!isValid(choice, state)) continue;

        // 4. MAKE CHOICE
        state.push(choice);

        // 5. RECURSE
        backtrack(state, choices, result);

        // 6. UNDO CHOICE (backtrack)
        state.pop();
    }
}
```

Template 4: Divide & Conquer

```
function divideAndConquer(input) {
    // 1. BASE CASE - smallest subproblem
    if (input.length <= 1) {
        return input;
    }

    // 2. DIVIDE - split into subproblems
    const mid = Math.floor(input.length / 2);
    const left = input.slice(0, mid);
    const right = input.slice(mid);

    // 3. CONQUER - solve subproblems recursively
```

```
    const leftResult = divideAndConquer(left);
    const rightResult = divideAndConquer(right);

    // 4. COMBINE - merge subproblem solutions
    return merge(leftResult, rightResult);
}
```

Quick Reference Card

```
╔═══════════════════════════════════════════════════════════════╗
║                 RECURSION INTERVIEW CHEAT SHEET                ║
╠═══════════════════════════════════════════════════════════════╣
║  1. ALWAYS DEFINE BASE CASE FIRST                              ║
║  2. TRUST THE RECURSIVE CALL (Leap of Faith)                   ║
║  3. IDENTIFY OVERLAPPING SUBPROBLEMS → Use Memoization         ║
║  4. SPACE OPTIMIZATION: Can you use O(1) space iteratively?    ║
║  5. WATCH FOR: Stack overflow, off-by-one, wrong base case     ║
╠═══════════════════════════════════════════════════════════════╣
║  COMPLEXITY QUICK REFERENCE:                                   ║
║  • Pure recursion with branching: O(branches^depth)            ║
║  • Memoized recursion: O(unique subproblems × work per)        ║
║  • Tail recursion: Can be optimized to O(1) space              ║
╠═══════════════════════════════════════════════════════════════╣
║  WHEN TO USE:                                                  ║
║  • Trees/Graphs → Almost always recursive                      ║
║  • "Generate all" → Backtracking recursion                     ║
║  • Optimal substructure → DP (recursive or iterative)          ║
║  • Linear iteration → Usually iterative is cleaner             ║
╚═══════════════════════════════════════════════════════════════╝
```

"To understand recursion, you must first understand recursion." 🔄

💾 Basic Recursive Functions

Maximum Subarray

🤼 Divide & Conquer