

# ■ Factorial of N

The cleanest recursion example — build your mental model here

● Recursion

■ EASY

## ■ What You Will Learn

- How recursion reduces a problem by one step each time
- The call stack in action — pushing and popping frames
- Base case: when to stop recursing
- Tail recursion and iterative equivalence

## ■ Intuition First

$5! = 5 \times 4 \times 3 \times 2 \times 1$ . But recursively:  $5! = 5 \times 4!$ . And  $4! = 4 \times 3!$ . And so on until  $1! = 1$  (we know this directly). So factorial is just: 'multiply  $n$  by the factorial of everything smaller.' The function keeps shrinking the problem by 1 each time.

Real-world twin: How many ways can 5 people line up? 5 choices for position 1, then 4 choices for position 2... =  $5! = 120$ . Factorial counts arrangements.

### ■ The Pattern

$n! = n \times (n-1)!$  This single equation IS the recursive solution. Base case:  $0! = 1$  and  $1! = 1$ . Most recursion problems follow this 'current  $\times$  result of smaller problem' pattern.

## ■ Visual — Call Stack for factorial(4)

CALL PHASE (pushing onto stack):

factorial(4) called

factorial(3) called

factorial(2) called

factorial(1) called

returns 1

RETURN PHASE (popping):

factorial(4) =  $4 \times 6 = 24$  ←

factorial(3) =  $3 \times 2 = 6$  ←

factorial(2) =  $2 \times 1 = 2$  ←

factorial(1) = 1 (base) ←

Stack (top = most recent call):

| factorial(1) | ← base case returns 1

| factorial(2) | ← gets 1, returns  $2 \times 1 = 2$

| factorial(3) | ← gets 2, returns  $3 \times 2 = 6$

| factorial(4) | ← gets 6, returns  $4 \times 6 = 24$

## ■■ How It Works — Step by Step

1. factorial(4) pauses and calls factorial(3)
2. factorial(3) pauses and calls factorial(2)
3. factorial(2) pauses and calls factorial(1)

4. factorial(1) hits base case → returns 1
5. Now unwind: factorial(2) gets 1 → returns  $2 \times 1 = 2$
6. factorial(3) gets 2 → returns  $3 \times 2 = 6$
7. factorial(4) gets 6 → returns  $4 \times 6 = 24$  ✓

## ■ Java Code

```
1  class Solution {
2      public long factorial(int n) {
3          // BASE CASE: 0! = 1 and 1! = 1
4          if (n <= 1) return 1;
5
6          // RECURSIVE CASE: n! = n * (n-1)!
7          return n * factorial(n - 1);
8      }
9  }
10
11 // Iterative equivalent (O(1) space):
12 public long factorialIterative(int n) {
13     long result = 1;
14     for (int i = 2; i <= n; i++) result *= i;
15     return result;
16 }
```

## ■■ Time & Space Complexity

| Type  | Value       | What It Means                             |
|-------|-------------|-------------------------------------------|
| Time  | <b>O(n)</b> | n multiplications, one per recursive call |
| Space | <b>O(n)</b> | n stack frames — one for each call        |

## ■■ Common Mistakes

### ■■ Mistake

Writing `return n * factorial(n)` instead of `factorial(n-1)`. This creates an infinite loop — you must make the problem **SMALLER** each step.

### ■■ Integer Overflow

`factorial(13) > Integer.MAX_VALUE`. Use `long` for `n` up to 20. Beyond that, use `BigInteger`.

## ■ Pro Tips

### ■ Recognize the Pattern

Any problem of the form ' $f(n) = n \text{ OP } f(n-1)$ ' screams recursion. factorial, sum of n, power(x,n), etc. The body is literally one line: `return n * factorial(n-1)`.

## ■ Practice Problems

| # | Problem                            | Diff   |
|---|------------------------------------|--------|
| 1 | Factorial of N (custom/GFG)        | Easy   |
| 2 | Power(x, n) — LeetCode 50          | Medium |
| 3 | Sum of digits recursively          | Easy   |
| 4 | Count down from N to 1 recursively | Easy   |

## ■ Quick Revision

|   |                                                                            |
|---|----------------------------------------------------------------------------|
| ✓ | factorial(n) = $n * \text{factorial}(n-1)$ ; base: $\text{factorial}(0)=1$ |
| ✓ | Call stack grows to depth n — each frame stores local state                |
| ✓ | Unwinds LIFO (Last In First Out) — innermost returns first                 |
| ✓ | Use long for large n to avoid overflow                                     |
| ✓ | Iterative version uses $O(1)$ space — preferred in production              |

## ■ Visual Thinking

Picture the call stack as a stack of plates in a cafeteria. Each recursive call adds a plate. When you hit the base case, the server (CPU) starts pulling plates off from the top. Each plate does its tiny calculation and hands the result to the plate below. The bottom plate (your original call) gets the final answer last.