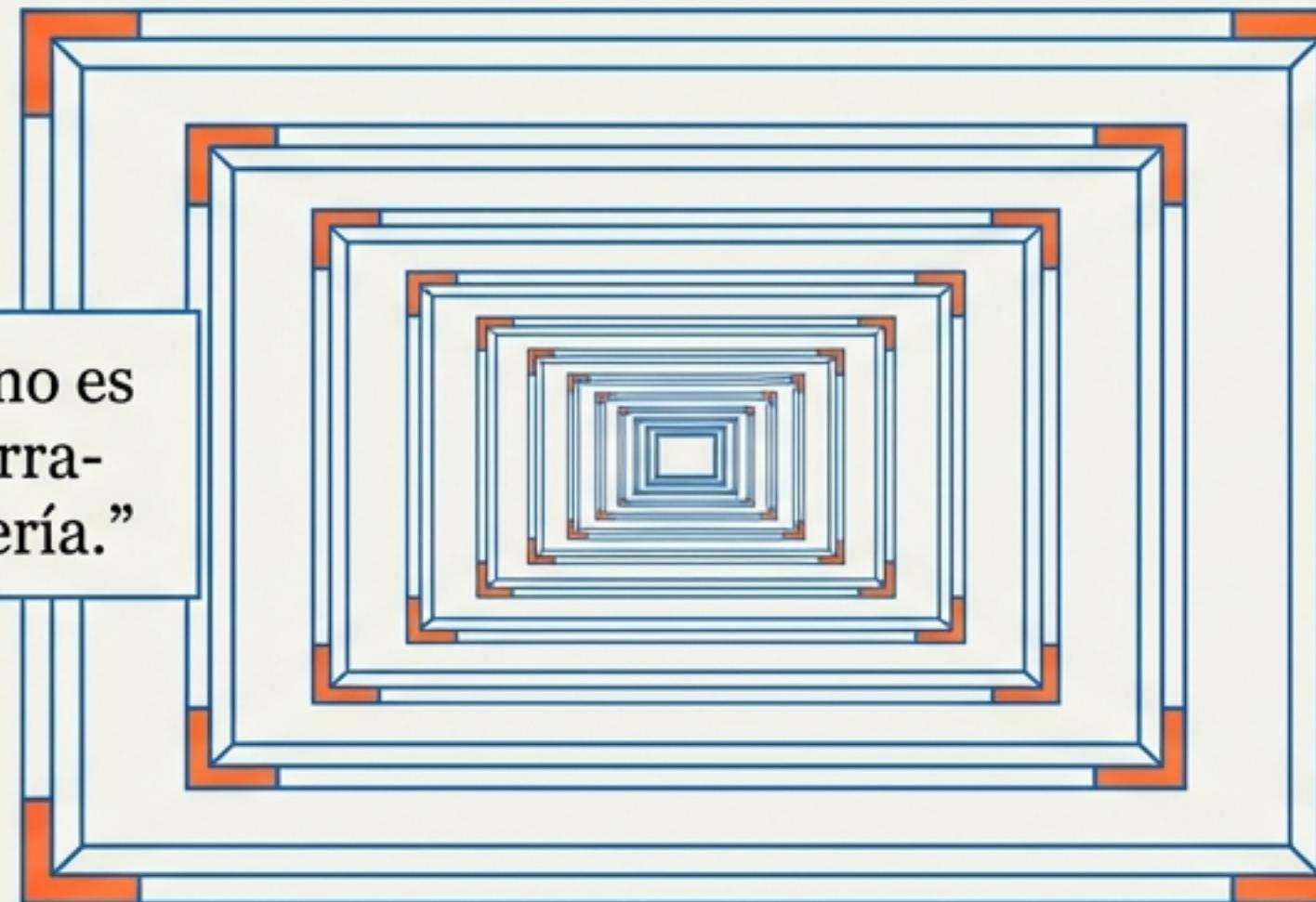


# RECURSIVIDAD EN PROGRAMACIÓN

De la teoría matemática a la implementación eficiente

“La recursividad no es magia; es una herramienta de ingeniería.”



## MECÁNICA

Anatomía de la función y gestión de memoria (Stack).

## APLICACIÓN

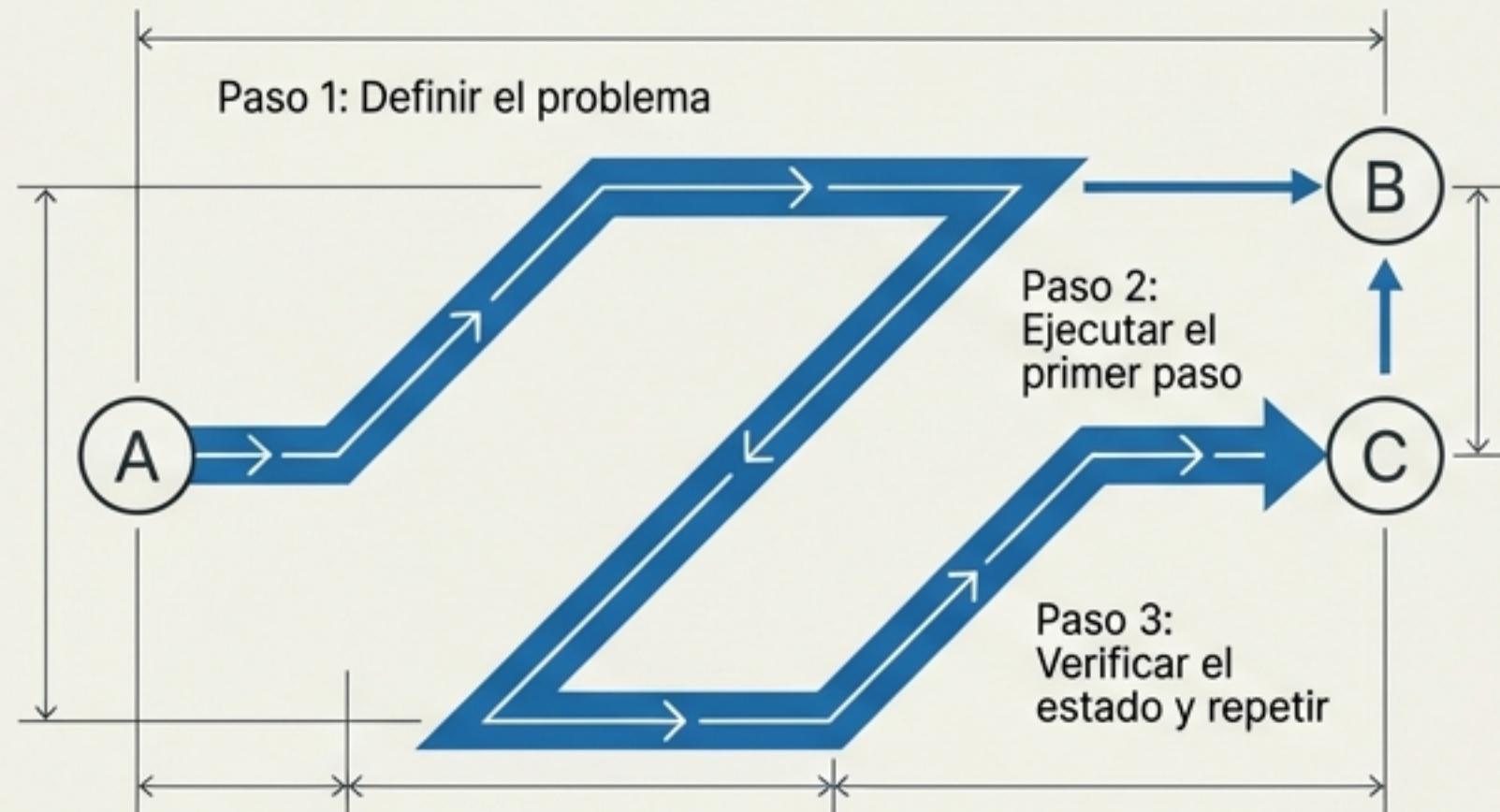
Ejemplos: Factorial, Fibonacci y Torres de Hanoi.

## OPTIMIZACIÓN

Riesgos de Stack Overflow y uso de Drivers.

# LA ESTRATEGIA: ¿ITERACIÓN O RECURSIVIDAD?

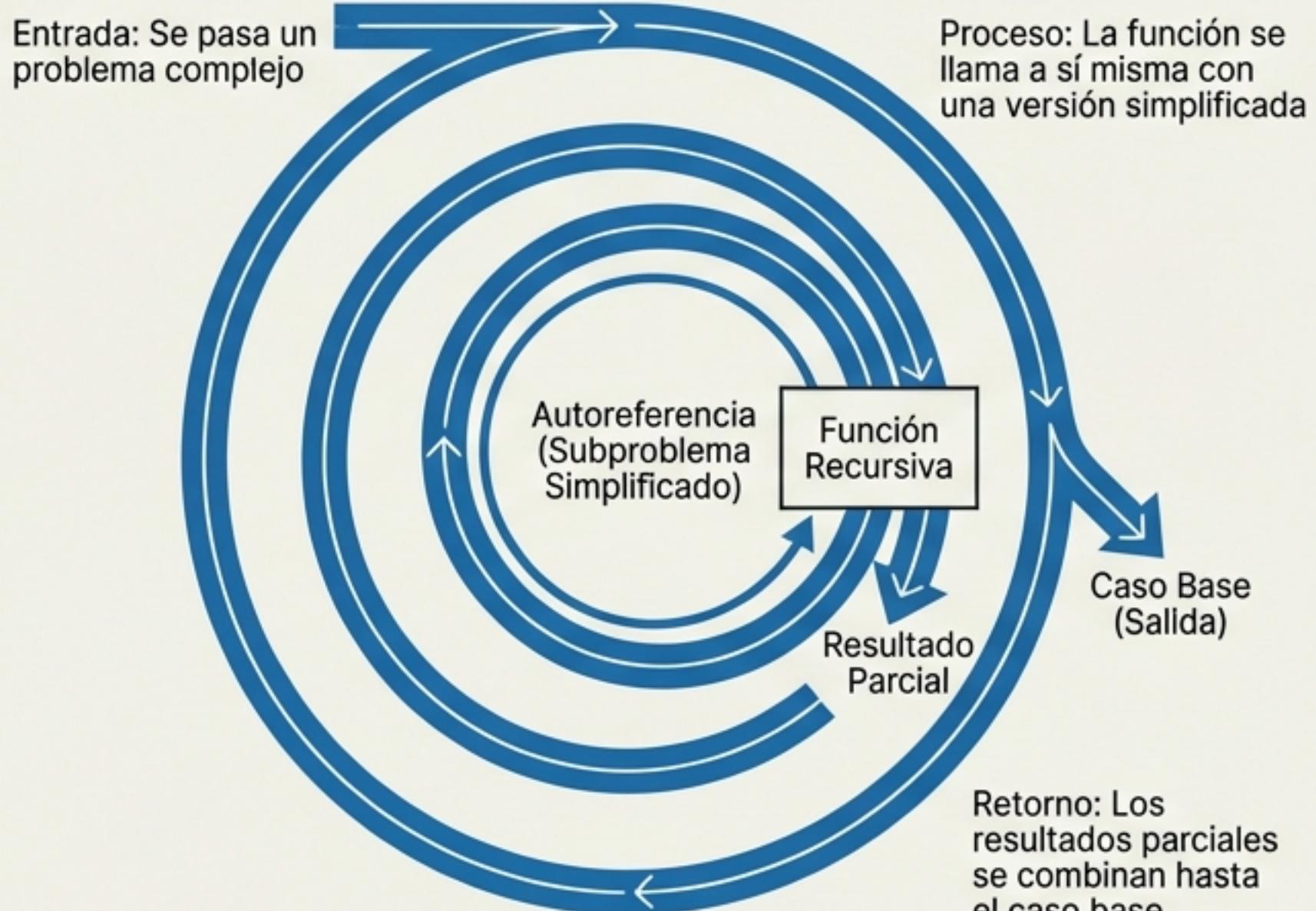
## PENSAMIENTO HUMANO (ITERACIÓN)



No existe una solución única. El programador debe elegir:

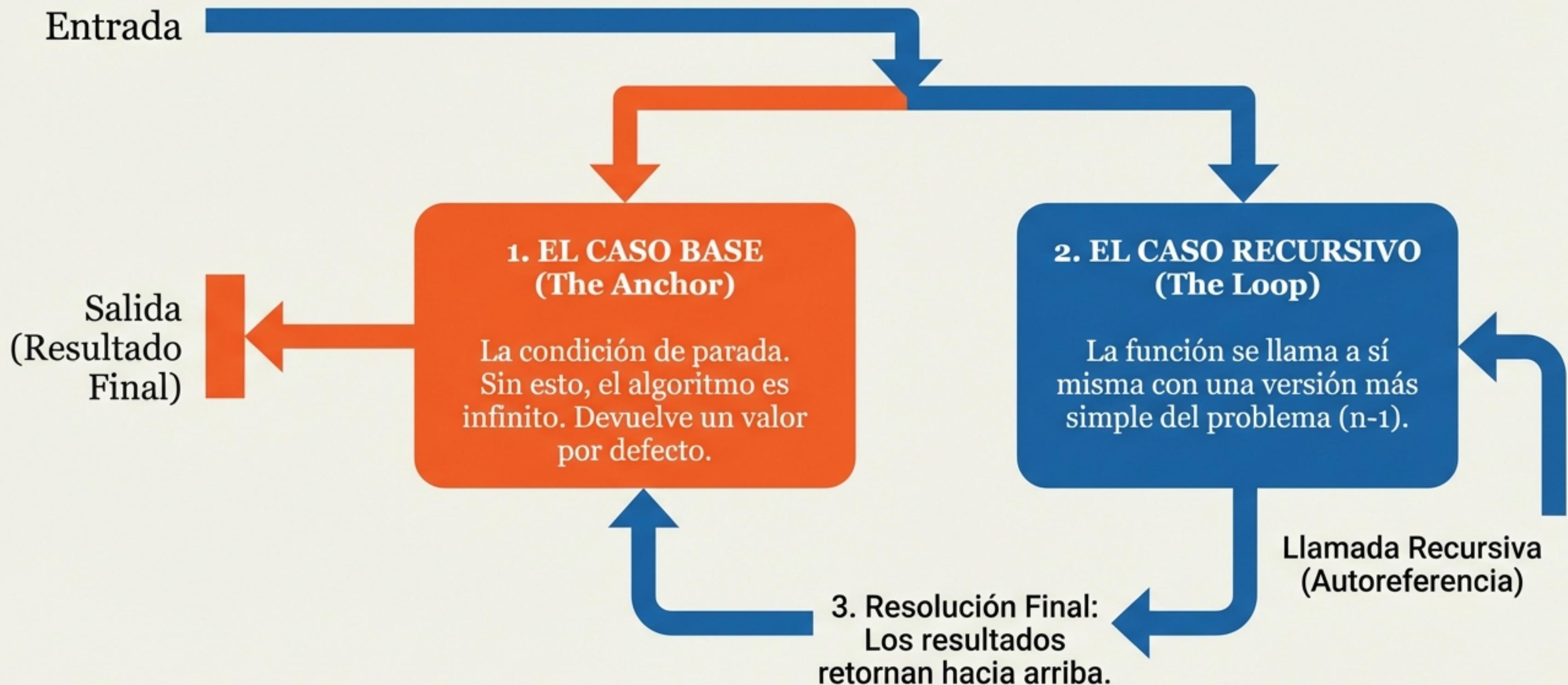
Iteración	RecurSividad
Natural para la mente humana (bucles).	Elegancia matemática y simplificación de problemas complejos.
Control manual del estado.	Autoreferencia (la función se llama a sí misma).

## PENSAMIENTO MATEMÁTICO (RECURSIVIDAD)

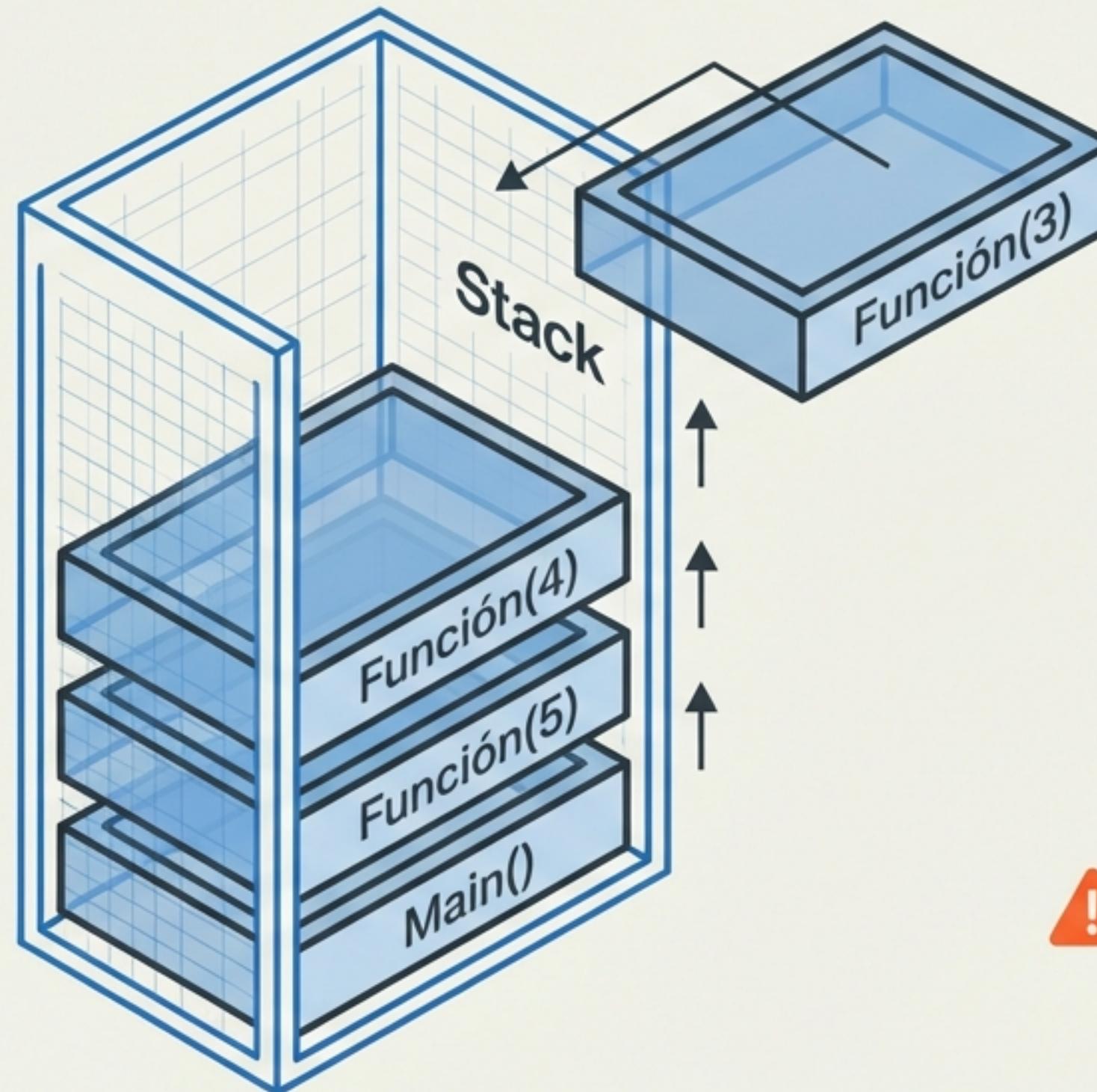


⚠ Advertencia: Depurar soluciones recursivas puede ser complejo.

# ANATOMÍA DE UNA FUNCIÓN RECURSIVA



# BAJO EL CAPÓ: LA PILA DE LLAMADAS (CALL STACK)



Mecánica de Memoria: Cada llamada crea una estructura intermedia en la 'Pila' (Stack).

El Límite Físico: La memoria es finita. La profundidad de la recursión depende de la RAM disponible.



El Riesgo: **Stack Overflow**  
(Desbordamiento). Ocurre cuando las llamadas superan la capacidad del contenedor.

# EL DILEMA: ELEGANCIA VS. EFICIENCIA



## VENTAJAS (PROS)

- Código más limpio y cercano a la definición matemática.
- Ideal para Estructuras Arbóreas: Navegación de nodos y hojas donde los bucles son confusos.



## DESVENTAJAS (CONS)

- Ineficiencia: Genera un gran número de llamadas.
- Abuso de Memoria: Cada llamada duplica variables en la Pila.
- Overhead: Gestión de frames costosa para la CPU.

# EJEMPLO CLÁSICO 1: FACTORIAL (RECURSIVIDAD LINEAL)

## DEFINICIÓN MATEMÁTICA

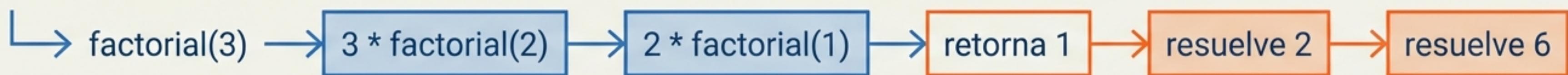
$F(n) = 1, \text{ si } n = 0$  (Caso Base)

$F(n) = n \times f(n-1), \text{ si } n \neq 0$   
(Caso Recursivo)

## IMPLEMENTACIÓN

```
funcion factorial(n)
    si n=1 entonces devolver 1
    sino devolver n * factorial(n-1)
```

-- Conexión Caso Base

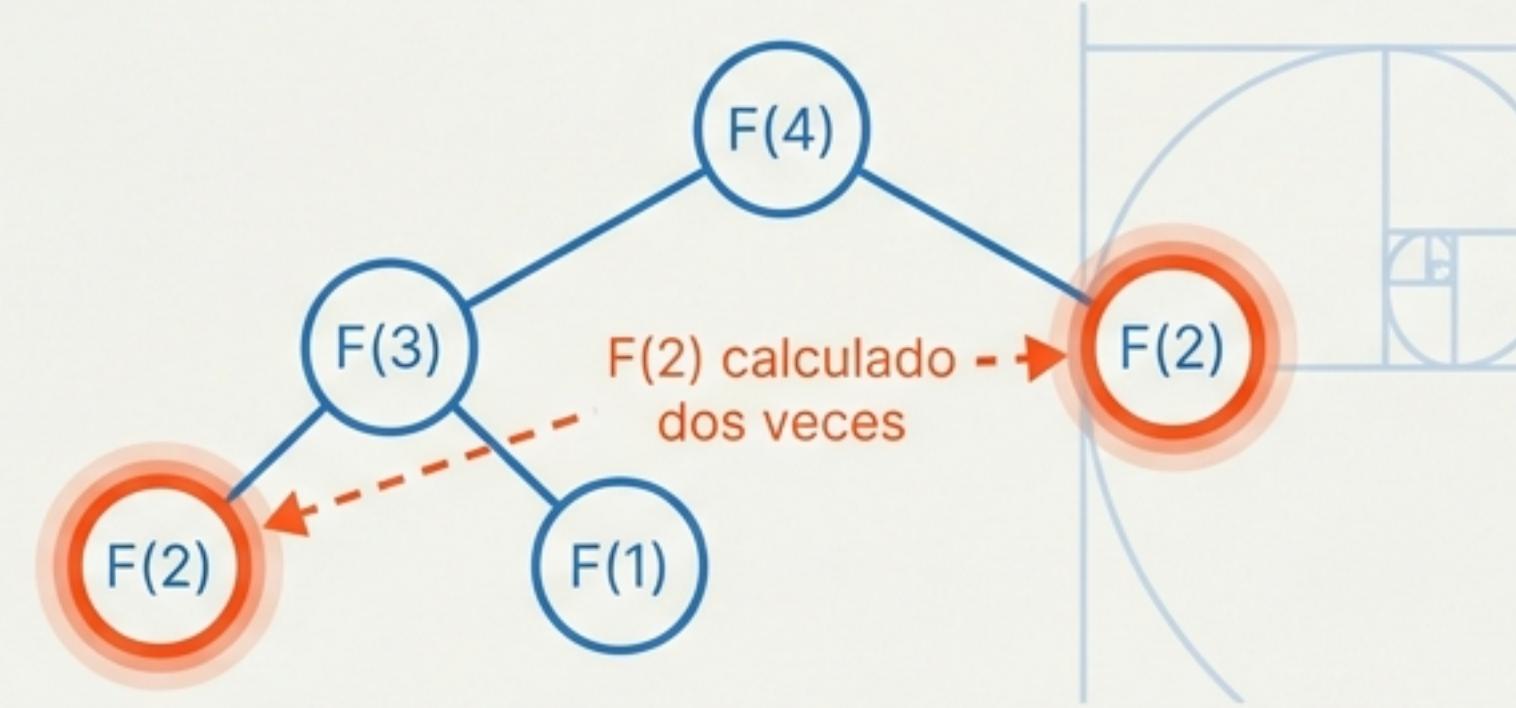


## EJEMPLO CLÁSICO 2: FIBONACCI (RECURSIVIDAD MÚLTIPLE)

Definición: Una espiral natural con DOS casos base.

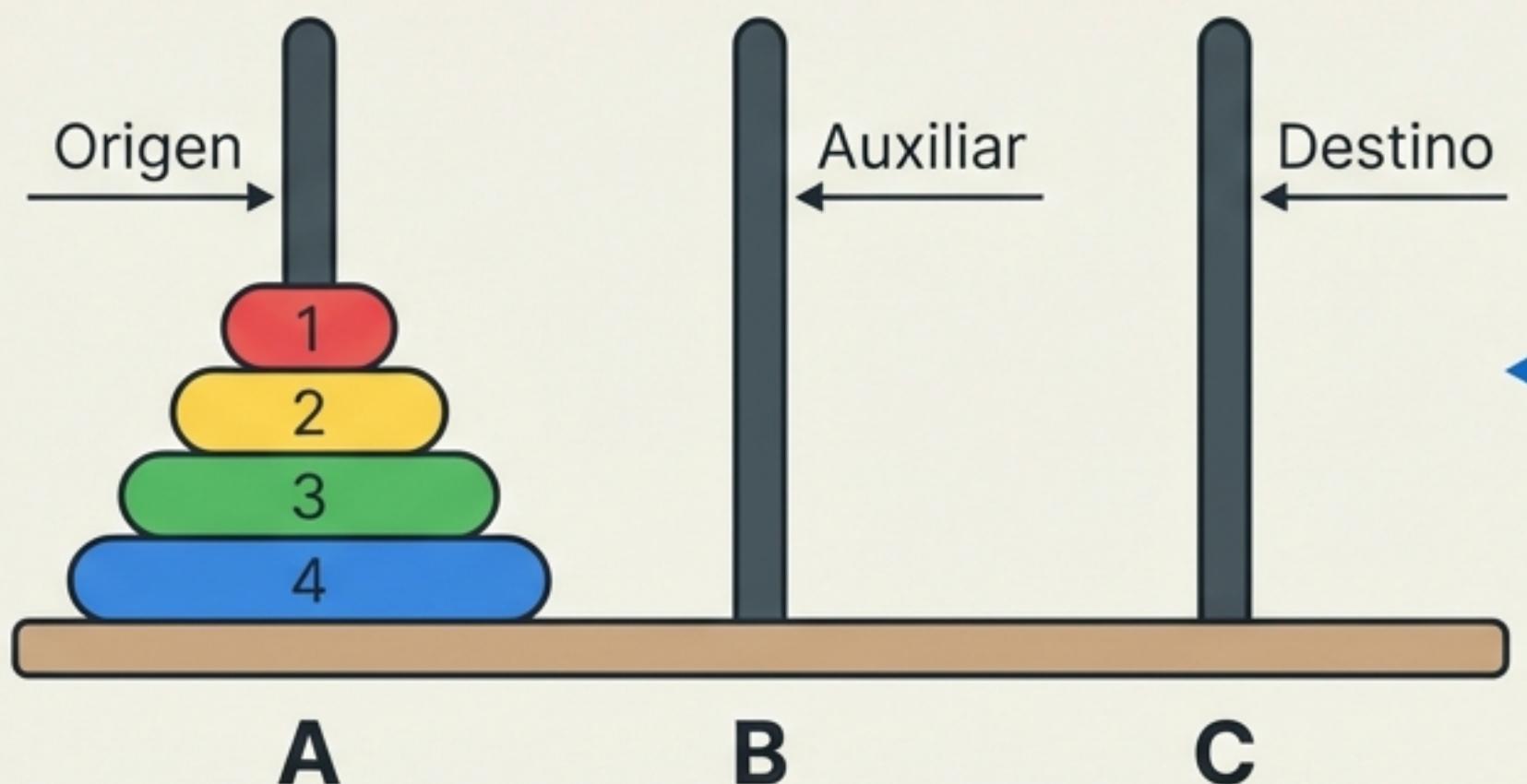
```
si n=0 retorna 0  
si n=1 retorna 1  
sino retorna fib(n-1) + fib(n-2)
```

### El árbol de llamadas (Ineficiencia)



Riesgo: Crecimiento exponencial de llamadas (Recursividad Múltiple).

# CASO PRÁCTICO: TORRES DE HANOI



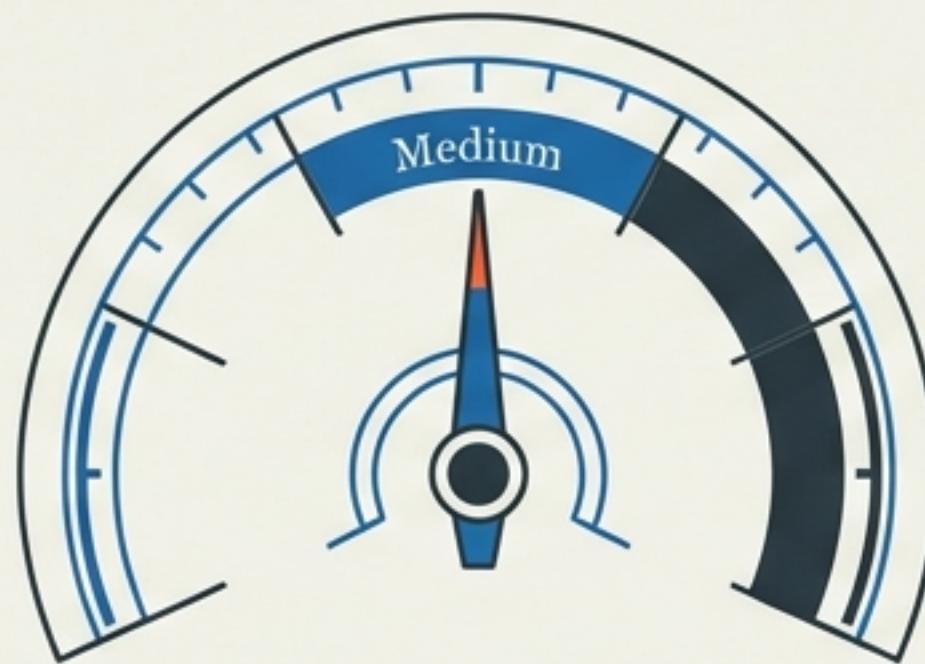
Algoritmo Recursivo (3 Pasos):

1. Mover  $n-1$  discos de Origen -> Auxiliar
2. Mover disco mayor de Origen -> Destino
3. Mover  $n-1$  discos de Auxiliar -> Destino

Caso Base: Si discos == 1, mover directo.

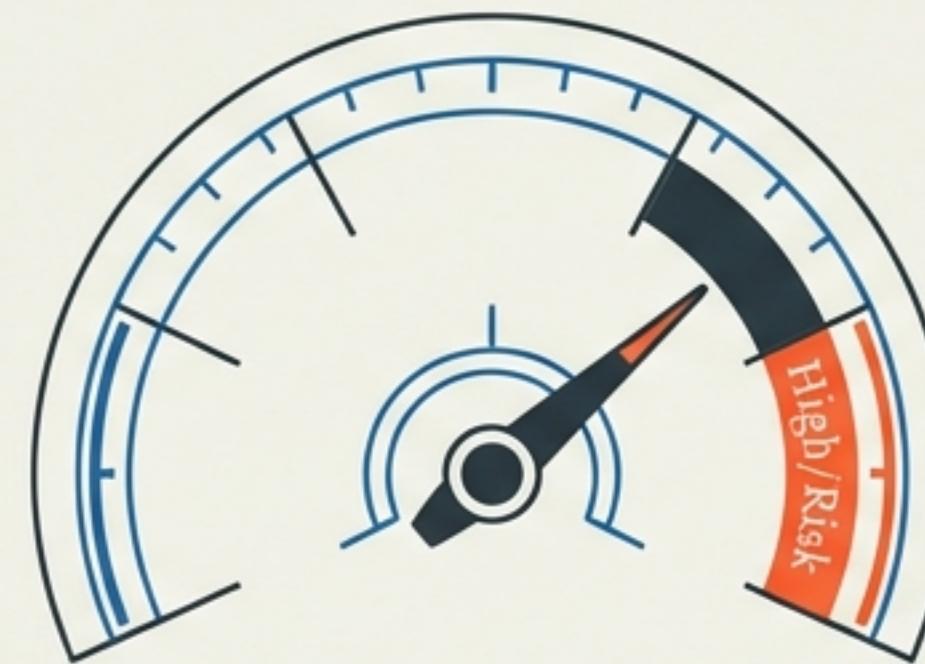
Problema trivial con recursividad, extremadamente complejo con iteración.

# EL COSTE DE LA SOLUCIÓN RECURSIVA



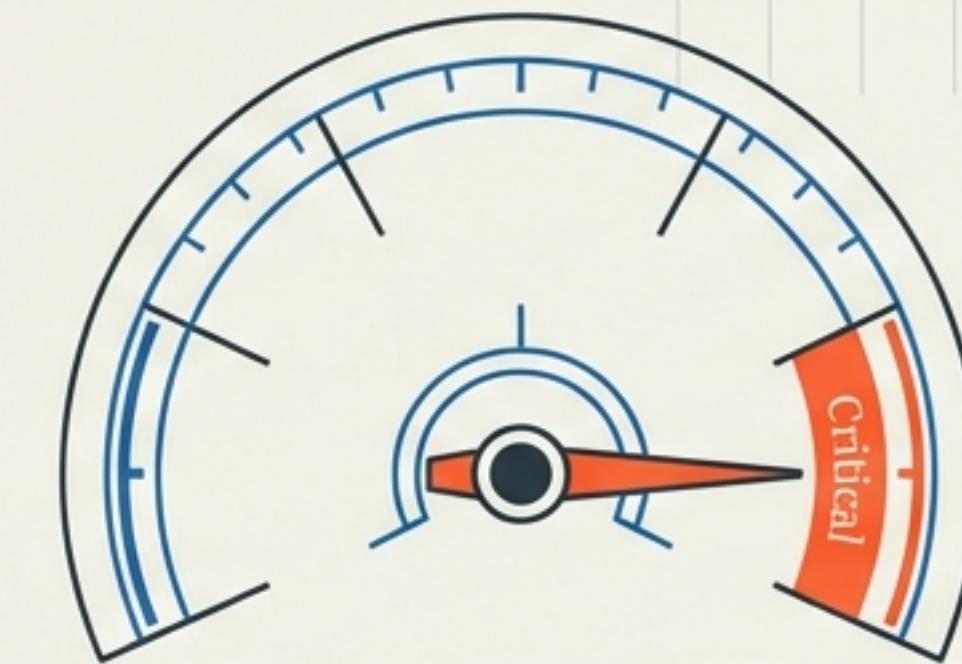
**ESFUERZO DE DESARROLLO**

Diseño elegante, pero depuración (debugging) difícil.



**COMPLEJIDAD TEMPORAL**

¿El algoritmo termina a tiempo?  
Crítico en sistemas real-time.

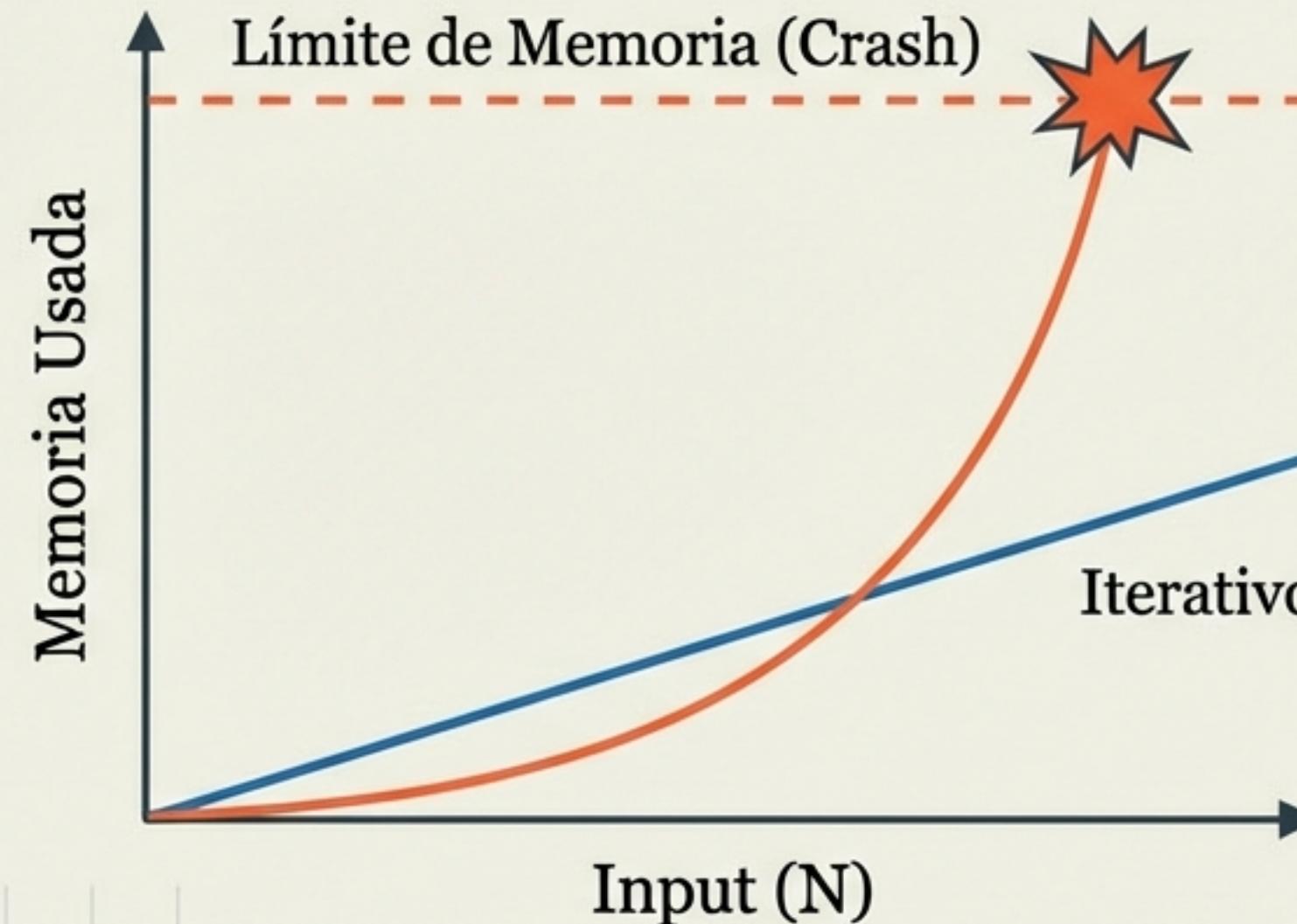


**COMPLEJIDAD ESPACIAL**

Uso intensivo de memoria  
(Stack Frames).

Regla de decisión: Si hay límites estrictos de hardware (sistemas embebidos), evita la recursividad.

# ZONA DE PELIGRO: STACK OVERFLOW



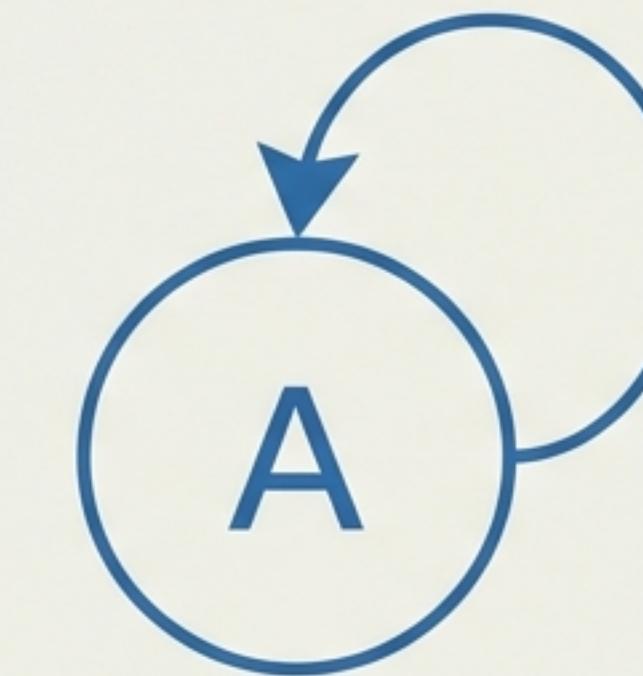
## Causas del Fallo Crítico:

1. Caso Base inalcanzable (Bucle Infinito).
2. Input (N) excesivo para la RAM física.
3. Ineficiencia por recálculo (Fibonacci sin optimizar).

“Al usuario no le importa la elegancia del código si el programa se cierra inesperadamente.”

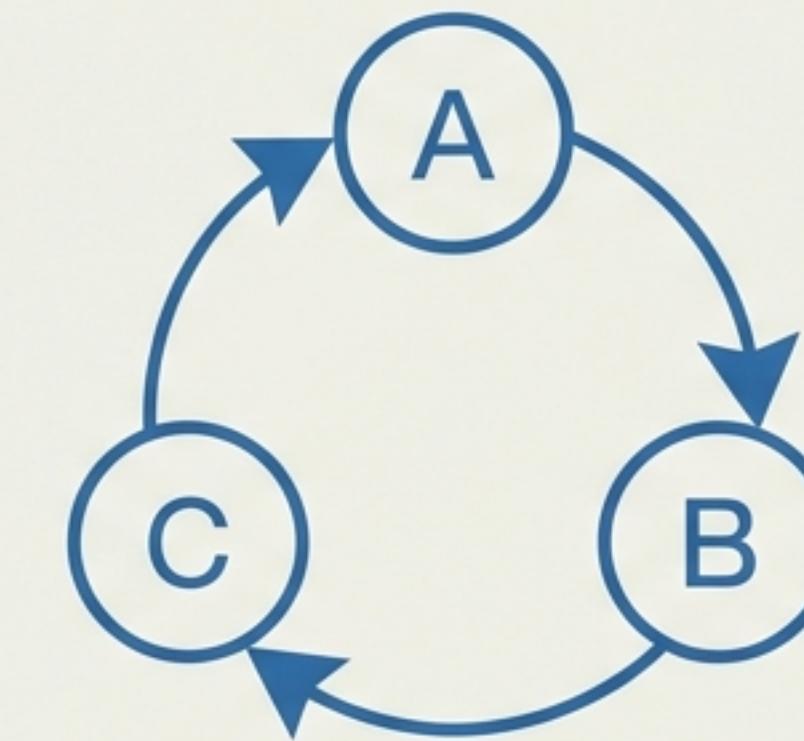
# TAXONOMÍA DE LA RECURSIVIDAD (I)

## RECURSIVIDAD DIRECTA



La función A llama a la función A.  
(Ej: Factorial).

## RECURSIVIDAD INDIRECTA

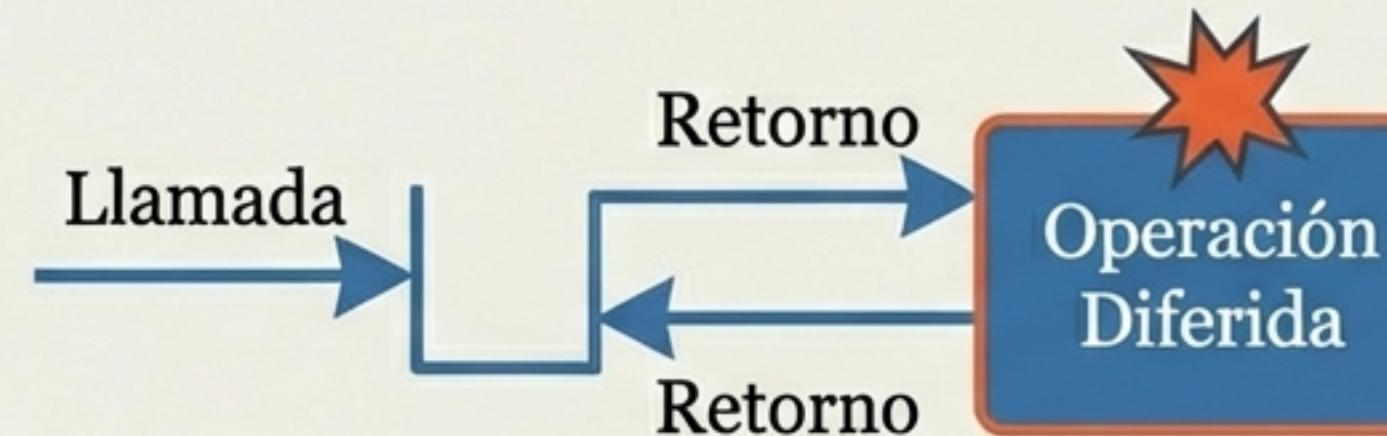


El Bucle Invisible.  
A llama a B, B llama a C, C llama a A.

⚠ Difícil de detectar y depurar.

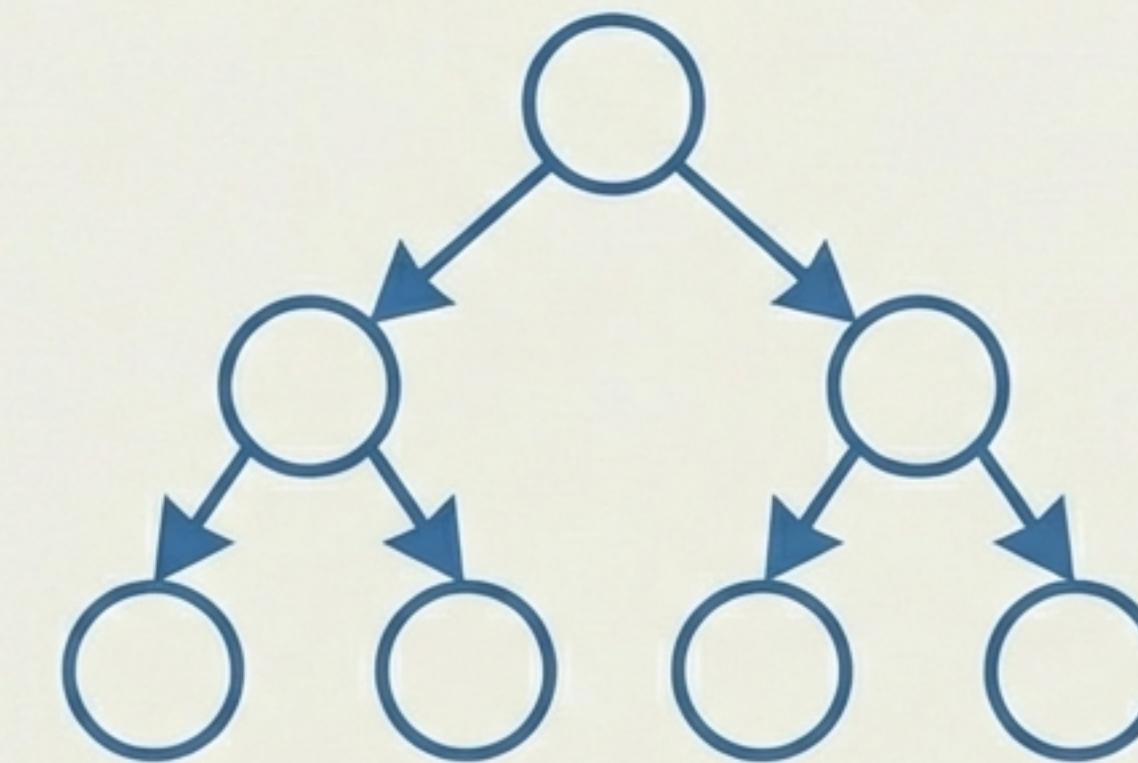
# TAXONOMÍA DE LA RECURSIVIDAD (II)

## RECURSIVIDAD EN AUMENTO



Las operaciones se ejecutan después del retorno (post-proceso).

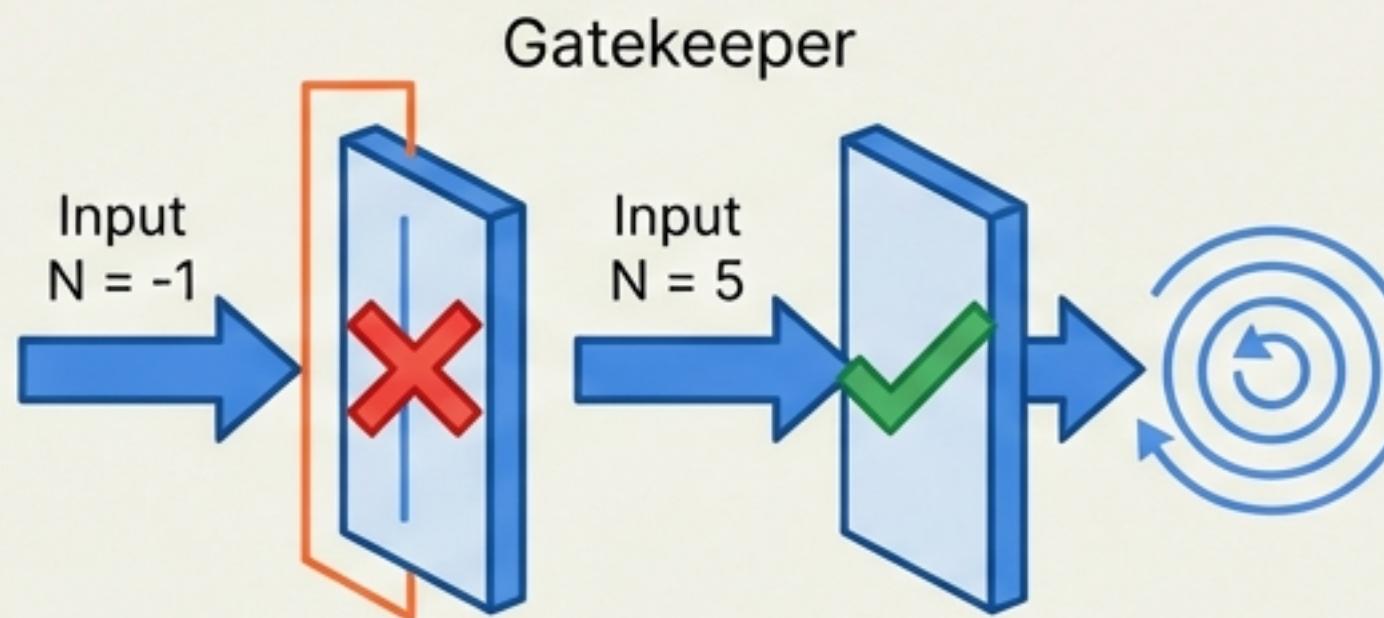
## RECURSIVIDAD MÚLTIPLE



Más de una llamada por paso (Ej: Fibonacci).

⚠ Riesgo alto de desbordamiento exponencial.

# ROBUSTEZ: VALIDACIÓN DE ENTRADAS

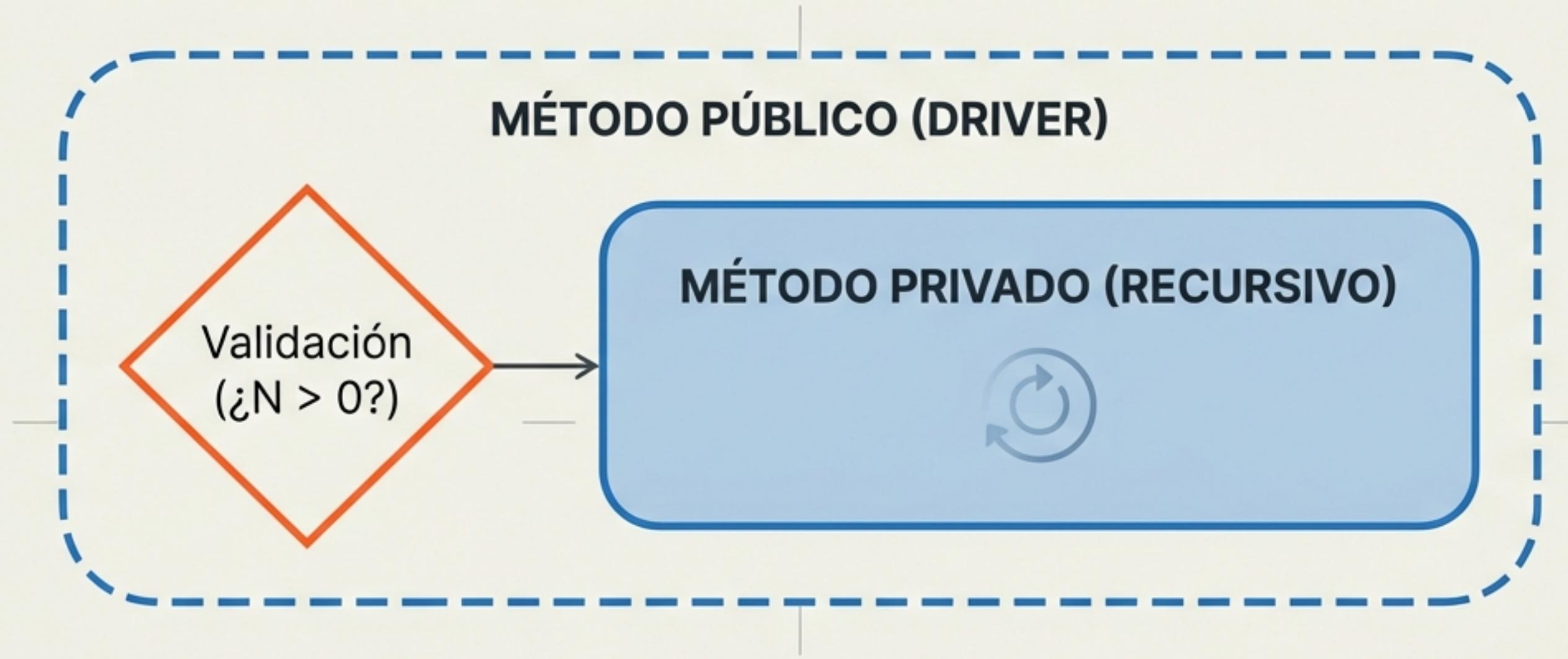


Escenario de Error: Si pasamos  $N = -1$  a una función factorial, el Caso Base ( $N=1$ ) nunca se alcanza  $\rightarrow$  Stack Overflow.

```
si (n < 1) {  
    imprimir('Error: Entrada inválida');  
} sino {  
    iniciar_recursion(n);  
}
```

Objetivo: Proteger la estabilidad de la máquina.

# PATRÓN DE OPTIMIZACIÓN: USO DE DRIVERS



## Problema:

Validar inputs dentro de la recursión es ineficiente (se repite en cada vuelta).

## Solución:

El Driver valida una sola vez y llama a la lógica pura.

## Estructura:

- Driver(n): Gestiona errores y configuración.
- Recursion(n): Solo contiene el algoritmo puro (Base + Loop).

# RESUMEN ESTRATÉGICO

## DEFINICIÓN

Solución elegante que divide un problema en versiones más pequeñas. Requiere Caso Base y Caso Recursivo.

## RIESGOS

Vigilar el Stack Overflow. Cuidado con la Recursividad Múltiple (Fibonacci) en inputs grandes.

## MEJORES PRÁCTICAS

Usar el patrón Driver. Validar siempre las entradas antes de recursar.

## CUÁNDO USAR

Perfecto para árboles y problemas no lineales (Hanoi). Evitar en bucles simples si la memoria es limitada.