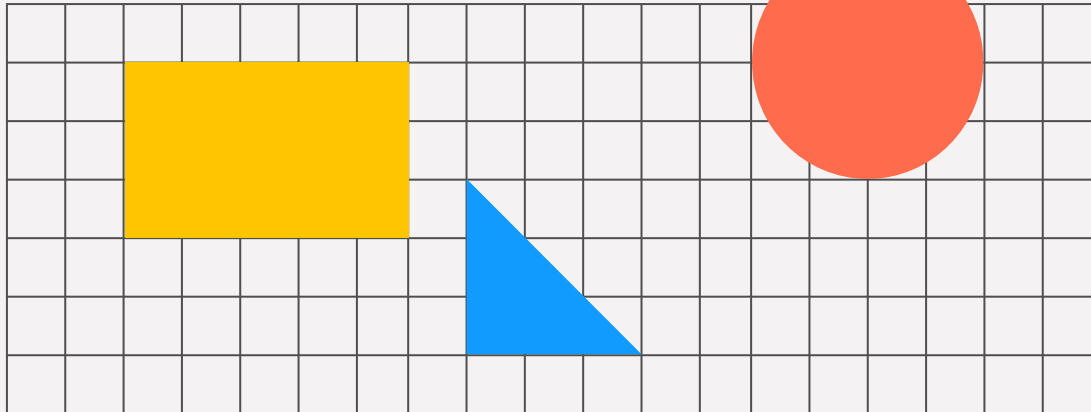



# Unidad 5

## Recursividad





La **recursividad** es una técnica de programación en la que **una función se llama a sí misma** para resolver un problema más pequeño del mismo tipo.

Es como cuando te mirás en un espejo frente a otro espejo y ves una imagen dentro de otra imagen... dentro de otra imagen...

Una función recursiva tiene **dos partes importantes**:

1. **Caso base (condición de parada)**: evita que la función se llame infinitamente.
2. **Llamada recursiva**: la función se llama a sí misma con un nuevo valor más cerca del caso base.



## ¿Qué pasa si NO hay caso base?

La función se llamaría a sí misma para siempre → se produce un **desbordamiento de pila (stack overflow)** y el programa se cae... o **EXPLOTA!!!!**



## ¿Cuándo usar recursividad?









Usá recursividad cuando:

- El problema se puede **dividir en subproblemas más pequeños**.
- No se necesita guardar muchos resultados intermedios (aunque esto puede optimizarse con técnicas como "memoización").
- Te conviene escribir menos código y más elegante (por ejemplo, para árboles, fractales, estructuras jerárquicas).



## Cuidados con la recursividad

1. Siempre debe tener un **caso base** que detenga las llamadas.
2. Puede ser **más lenta** que las soluciones iterativas si no se optimiza.
3. Consume **más memoria** por las llamadas anidadas en la pila.

Criterio	Recursividad	Estructura Repetitiva
 <b>Concepto</b>	Una función se llama a sí misma para resolver un problema.	Se repite un bloque de código usando <code>for</code> , <code>while</code> o <code>do-while</code> .
 <b>Código más legible</b>	Sí, en problemas como factorial, torres de Hanoi, árboles.	No tanto para problemas muy recursivos.
 <b>Uso de memoria</b>	Alto: cada llamada ocupa espacio en la pila (stack).	Bajo: usa una sola variable de control.
 <b>Velocidad (performance)</b>	Más lento si hay muchas llamadas (por sobrecarga de stack).	Más rápido en la mayoría de los casos.
 <b>Riesgo de desbordamiento</b>	Sí, puede causar Stack Overflow si no tiene un caso base correcto.	No hay riesgo de desbordamiento por sí solo.
 <b>Adecuado para</b>	Problemas recursivos por naturaleza: árboles, combinaciones, fractales.	Cálculos repetitivos como sumas, multiplicaciones, bucles comunes.
 <b>Fácil de entender al inicio</b>	No, puede ser confuso para principiantes.	Sí, es más directo.
 <b>Conversión a iterativo</b>	A veces difícil (ej: algoritmos con múltiples caminos como backtracking).	Fácil de codificar y entender.

```
#include <stdio.h>
//Contar hacia atrás
void contarAtras(int n) {
    if (n == 0) {
        printf(";Listo!\n"); // Caso base
    } else {
        printf("%d\n", n);    // Mostrar el número actual
        contarAtras(n - 1);   // Llamada recursiva
    }
}

int main() {
    contarAtras(5); // Comenzamos desde 5
    return 0;
}
```

```
#include <stdio.h>
//Factorial de un número
int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1; // Caso base
    } else {
        return n * factorial(n - 1); // Llamada recursiva
    }
}

int main() {
    int num = 4;
    printf("El factorial de %d es %d\n", num, factorial(num));
    return 0;
}
```



```
#include <stdio.h>
//Serie de Fibonacci
int fibonacci(int n) {
    if (n == 0) return 0; // Caso base
    if (n == 1) return 1; // Caso base
    return fibonacci(n - 1) + fibonacci(n - 2); // Recursión
}

int main() {
    int i;
    for (i = 0; i < 8; i++) {
        printf("%d ", fibonacci(i));
    }
    return 0;
}
```