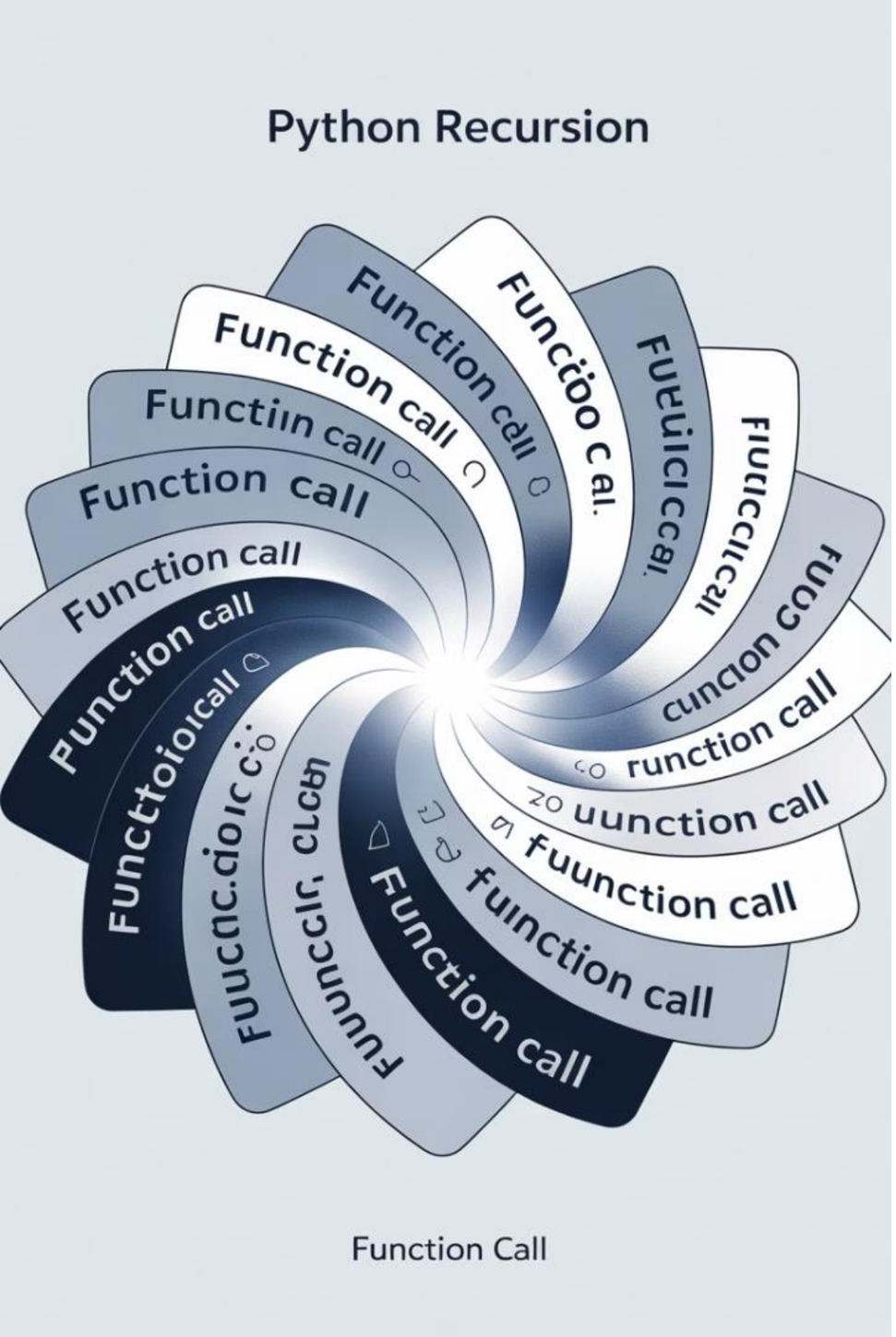



# Guía Completa sobre Recursividad en Python

La recursividad es una técnica fundamental en programación que permite a una función llamarse a sí misma para resolver problemas complejos. En esta presentación, exploraremos en profundidad cómo funciona la recursividad en Python, sus aplicaciones prácticas, ventajas y consideraciones importantes.

Aprenderemos desde ejemplos básicos hasta implementaciones más avanzadas, y entenderemos por qué la recursividad es una herramienta poderosa en el arsenal de cualquier programador.

# ¿Qué es la Recursividad?

## Definición

La **recursividad** es una técnica de programación en la cual una **función se llama a sí misma** para resolver un problema. Cada llamada recursiva resuelve una **parte del problema**, y el conjunto de llamadas sucesivas permiten alcanzar la solución final.

## Componentes Clave

- Caso base (condición de corte)
- Paso recursivo

El caso base detiene la recursión, evitando un proceso infinito. El paso recursivo es donde la función se llama a sí misma con un problema más pequeño o modificado.

# Funcionamiento de la Recursividad

### Definición del Problema

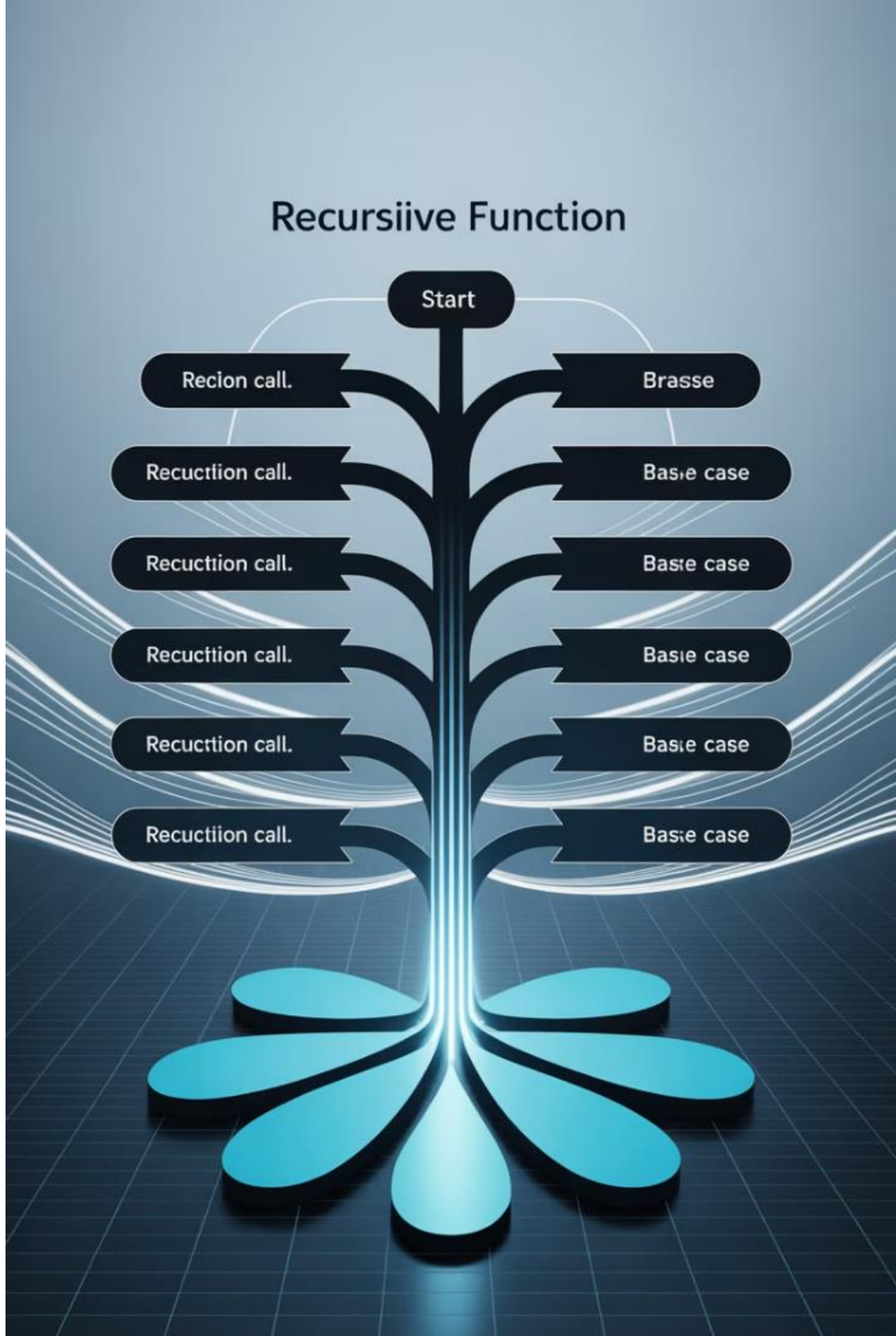
Se identifica un problema que puede dividirse en casos más simples del mismo tipo.

### Caso Base

Se establece una condición que **detiene la recursión**, representando la solución directa para el caso más simple.

### Paso Recursivo

La función **se llama a sí misma** con un problema más pequeño, acercándose gradualmente al caso base.



# Ejemplo Simple: Cuenta Regresiva

## Código

```
def cuenta_regresiva(n):
 if n == 0:
   print("¡Despegue!")
 else:
   print(n)
   cuenta_regresiva(n - 1)

cuenta_regresiva(5)
```

Este ejemplo muestra cómo una función recursiva puede implementar una cuenta regresiva, llamándose a sí misma con valores decrecientes hasta alcanzar el caso base (n = 0).

## Ejecución

Cuando llamamos a **cuenta_regresiva(5)**, la función imprime "5" y luego se llama a sí misma con n = 4. Este proceso continúa hasta que n = 0, momento en el que imprime "¡Despegue!" y termina la recursión.

# Diagrama Conceptual de la Recursividad

### cuenta_regresiva(3)

Imprime 3 y llama a cuenta_regresiva(2)

### cuenta_regresiva(2)

Imprime 2 y llama a cuenta_regresiva(1)

### cuenta_regresiva(1)

Imprime 1 y llama a cuenta_regresiva(0)

### cuenta_regresiva(0)

Imprime "¡Despegue!" y termina

# Ejemplo Clásico: Factorial

## Definición Matemática

La función factorial se define matemáticamente así:

- factorial(0) = 1
- factorial(n) = n × factorial(n-1)

## Implementación en Python

```
def factorial(n):
 if n == 0:
   return 1
 else:
   return n * factorial(n - 1)

print(factorial(5)) # Resultado: 120
```

# Comparación: Recursión vs Bucle

### Implementación Recursiva

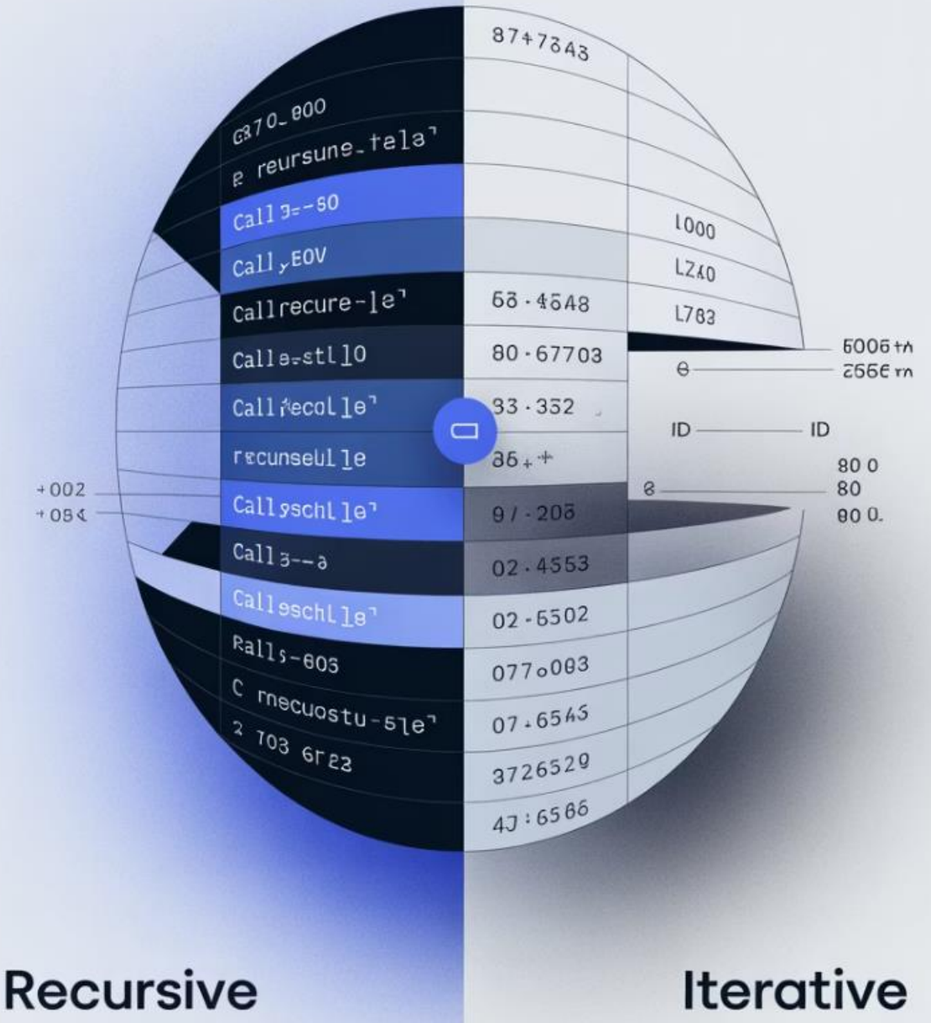
```
def factorial(n):
 if n == 0:
   return 1
 else:
   return n * factorial(n - 1)
```

Más elegante y cercana a la definición matemática, pero puede consumir más memoria debido al uso de la pila de llamadas.

### Implementación Iterativa

```
def factorial_iterativo(n):
 resultado = 1
 for i in range(1, n + 1):
   resultado *= i
 return resultado
```

Más eficiente en términos de memoria, ya que no utiliza la pila de llamadas de manera extensiva.




# Consideraciones Importantes

## Caso Base Obligatorio

La **recursión debe tener un caso base claro**, o se producirá un error de recursión infinita (RecursionError). Sin una condición de parada, la función seguirá llamándose indefinidamente.

## Límite de Profundidad

Python tiene un **límite de profundidad recursiva** (por defecto es 1000 llamadas). Este límite protege contra desbordamientos de pila pero restringe la complejidad de los problemas que pueden resolverse.

## Consumo de Memoria

Cada llamada recursiva consume memoria en la pila de llamadas. Para problemas grandes, esto puede llevar a un uso excesivo de recursos comparado con soluciones iterativas.

# Ejemplo Útil: Suma de una Lista

### Problema

Sumar todos los elementos de una lista utilizando recursividad.

### Implementación

Dividir la lista en el primer elemento y el resto, sumando recursivamente.

### Resultado

La suma se calcula progresivamente a medida que las llamadas recursivas se completan.

### Solución

Caso base: lista vacía = 0. Paso recursivo: primer elemento + suma del resto.

```
def suma_lista(lista):
 if not lista:
   return 0
 else:
   return lista[0] + suma_lista(lista[1:])

print(suma_lista([1, 2, 3, 4])) # Resultado: 10
```

# Ejemplo Más Complejo: Fibonacci

### Definición

Cada número es la suma de los dos anteriores: F(n) = F(n-1) + F(n-2)

### Implementación

Casos base: F(0)=0, F(1)=1. Recursión para n>1.

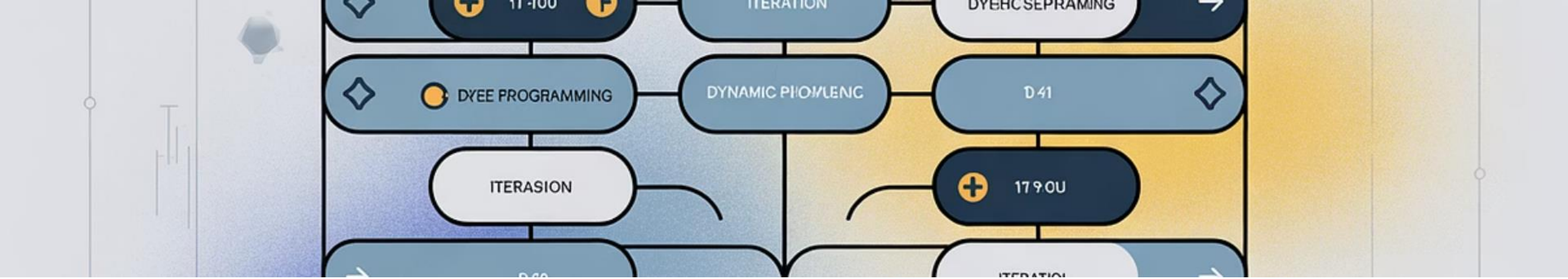
### Limitación

Ineficiente para valores grandes debido a cálculos repetidos.

```
def fibonacci(n):
 if n == 0 or n == 1:
   return n
 else:
   return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(6)) # Resultado: 8
```

**Nota:** Esta versión no es eficiente para valores grandes. Se pueden aplicar técnicas como **memoización** para mejorarla.




# ¿Cuándo Usar Recursividad?

## Estructuras Jerárquicas

Ideal para trabajar con árboles, directorios de archivos y otras estructuras anidadas donde cada elemento puede contener elementos similares.

## Problemas Divisibles

Cuando un problema puede dividirse en subproblemas más pequeños del mismo tipo, como en algoritmos "divide y vencerás".

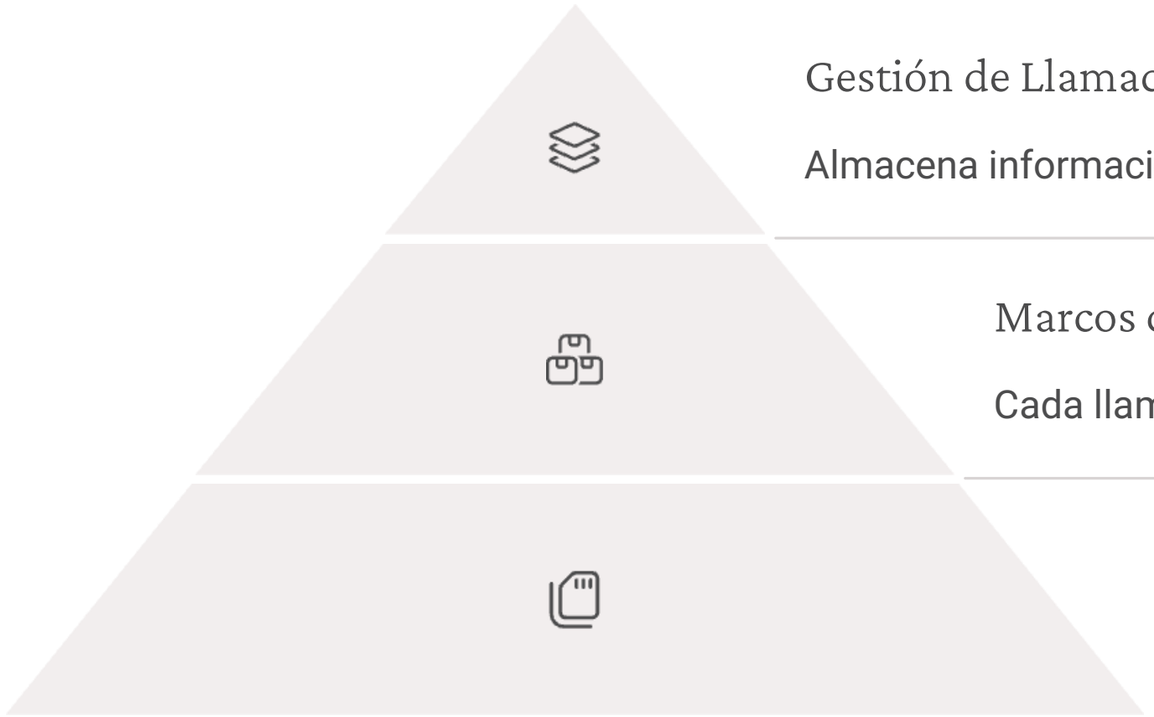
## Elegancia del Código

Cuando la solución recursiva es significativamente más clara y concisa que la iterativa, mejorando la legibilidad y mantenimiento.

## Algoritmos de Búsqueda

Para recorrer estructuras complejas como grafos o realizar búsquedas en profundidad.

# El Stack o Pila de Llamadas



### Gestión de Llamadas

Almacena información de cada llamada a función

### Marcos de Ejecución

Cada llamada crea un nuevo marco en la pila

### Memoria Limitada

El tamaño de la pila tiene un límite físico

Cuando se llama a una función en Python, el programa guarda la **información de esa llamada en una pila**, conocida como **call stack**. Cada vez que una función **se llama a sí misma**, se agrega un nuevo **marco de ejecución** al tope de la pila. Cuando una función termina, ese marco se **elimina** de la pila.

# Recursos Adicionales



Para profundizar en el tema de la recursividad, recomendamos explorar el artículo **"Visualizing Recursion"** disponible en el siguiente enlace:

https://medium.com/swlh/visualizing-recursion-6a81d50d6c41

Este recurso ofrece visualizaciones interactivas que ayudan a comprender mejor cómo funciona la recursividad en la práctica, complementando los conceptos teóricos que hemos explorado en esta presentación.