Recursión

"Life can only be understood backwards; but it must be lived forwards." — Søren Kierkegaard

¿Qué significa recursión?

Según el diccionario Merriam Webster, la recursión es «*la determinación de una sucesión de elementos (como números o funciones) por operación en uno o más elementos anteriores de acuerdo con una regla o fórmula que involucra un número finito de pasos*».

Una manera elegante de decir «*un programa que se llama a sí mismo*». Siendo un poco más formales, escribamos la siguiente definición:

Definición. La **recursión** es una forma de definir un objeto o un proceso, definiendo explícitamente su forma más simple y definiendo sus formas más complejas con respecto a formas más simples.

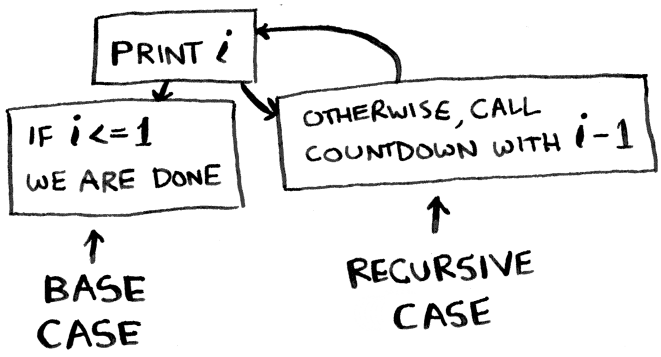
Definición. Un **algoritmo recursivo** es un algoritmo que expresa la solución de un problema en términos de una llamada a sí mismo. La llamada a sí mismo se conoce como llamada recursiva.

⚠ La programación recursiva se basa en un conjunto relativamente simple de principios para resolver problemas (que de otro modo podrían ser desafiantes), al dividir el problema en las piezas más pequeñas posibles y luego usar esas piezas como bloques de construcción para la eventual respuesta.

Un método recursivo se divide en dos aspectos distintos y extremadamente importantes.

1) La llamada recursiva, que (lo adivinó) es la (s) línea (s) de código responsables de que el método se llame a sí mismo.

2) El caso base, que es un condicional que permitirá que la cadena de llamadas infinitas se detenga.



⚠ El aspecto más importante del caso base es que el problema no puede ser más simple y, por lo tanto, la respuesta es casi trivial.

Ejemplo 1. La suma de los enteros en un `array`.

Solución iterativa

```
a = array([1,2,3,4])

def iterative_sum(a):
    total=0
    i=0

    while(i<len(a)):
        total+=a[i]
        i+=1
    return total
```

Solución recursiva

```
a = array([1,2,3,4])

def recursive_sum(a):
    #caso base
    if len(a)==0:
        return 0
    #caso recursivo
    return a[0]+recursive_sum(a[1:])
```

⚠️ Es frecuente que los algoritmos recurrentes sean más ineficientes en tiempo que los iterativos aunque suelen ser mucho más breves en espacio.

Ejemplo 2. La definición explícita del **factorial** de un entero es:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

Podemos convertir esta definición a una *definición recursiva* de la siguiente manera:

Caso base: $0! = 1$
Caso recursivo: $n! = (n-1)! \cdot n$

⚠️ Calcular $(n-1)!$ es un **subproblema** que podemos resolver para calcular $n!$

La función factorial se puede implementar de forma recursiva en Python 🐍 como:

```
def factorial(n):
    #caso base
    if n==0:
        return 1
    #caso recursivo
    else:
        return factorial(n-1)*n
```

Si llamamos a la función:

```
factorial(4)
    factorial(3)                    #push stack n=4
        factorial(2)                #push stack n=3
            factorial(1)            #push stack n=2
                factorial(0)        #push stack n=1
                return 1
            return 1*1              #pop stack n=1
        return 1*1*2                #pop stack n=2
    return 1*1*2*3                  #pop stack n=3
return 1*1*2*3*4                    #pop stack n=4
```

⚠️ Es necesaria una estructura de datos para recordar los valores de las variables cada vez que se llama a la función. Dicha estructura recibe el nombre de pila o *stack*; requiere memoria de la computadora y tiene un límite. Si excedemos el límite ocurre un desbordamiento de pila o *stack overflow*.

⚠️ Claramente vemos que n toma los valores en orden inverso al que fueron ingresando, cuando va de regreso de las llamadas recursivas; *First-In Last-Out* (*FILO*).

Ejercicio 3. Escriba una función que imprima un número entero positivo en su representación binaria.

👀 Observaciones:

- si un número es par, termina en 0 y si es impar termina en 1; es nuestro caso base
- si un número se divide por 2, el cociente (en binario) se escribe igual que el dividendo sin el último dígito a la derecha; es nuestro caso recursivo

Por ejemplo, $(1011)_2/2 = (101)_2$

```
def imprime_binario(n):
    #caso recursivo
    if n>=2:
        imprime_binario(int(n/2))
        print(n%2)
    #caso base
    else:
        print(n)
```

⚠️ El caso base no es necesariamente lo primero que se escribe en el código, depende de la estructura lógica que siga su programa.

Si llamamos a la función:

```
imprime_binario(10)
    imprime_binario(5)          #push n=10
        imprime_binario(2)      #push n=5
            imprime_binario(1)  #push n=2
            print(1)
        print(0)                #pop n=2
    print(1)                    #pop n=5
print(0)                        #pop n=10

Out[x]: 1010    #la representación en binario de 10
```

⚠️ La recursión debemos usarla cuando simplifique sustancialmente los procedimientos y «valga la pena» pagar el precio de menor rendimiento.

Ejercicio 1: Recursividad

Ejercicio 4. Escriba una función recursiva que calcule el valor de b^n, $b > 0$, $n \geq 0 \in Z$.

Ejercicio 5. Alguien compra una pareja de conejos. Luego de un mes esos conejos son adultos. Después de dos meses esa pareja de conejos da a luz a otra pareja de conejos. Al tercer mes la primera pareja da a luz a otra pareja de conejos y sus primeros hijos se vuelven adultos. Cada mes qué pasa, cada pareja de conejos adultos da a luz a una nueva pareja de conejos y una pareja de conejos tarda un mes en crecer. Escriba una función recursiva que regrese cuántos conejos adultos se tienen pasados n meses.

Requierimiento para medir *performance*:

```
import time
...
start_time=time.time()
print(F(40))
print(time.time()-start_time)
```