Algoritmos recursivos

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.

El código 1a implementa una versión recursiva de la función Factorial N → N
El código 1b implementa una versión iterativa de la misma función.

Generalmente, si la primera llamada al subprograma se plantea sobre un problema de tamaño u orden n, cada nueva ejecución recurrente del mismo se planteará sobre problemas, de igual naturaleza que el original, pero de un tamaño menor que n.

De esta forma, al ir reduciendo progresivamente la complejidad del problema que resolver, llegará un momento en que su resolución sea más o menos trivial (o, al menos, suficientemente manejable como para resolverlo de forma no recursiva).

En esa situación diremos que estamos ante un **caso base** de la recursividad.

```
def factorial(num):
  if num < 0:
    raise Exception("Debe ser un entero positivo o 0")
  resultado = 1
  if num > 0:
    resultado = num * factorial(num-1)
  print(num,resultado)
  return resultado
```

Código 1ª: factorial recursivo

```
def factorialIter(n):
  ret = n
  while n > 1:
    n = n - 1
    ret = ret * n
  return ret
```

Código 1b: factorial iterativo

Las claves para construir un subprograma recursivo son:

(1) Cada llamada recursiva se debería definir sobre un problema de menor complejidad (algo más fácil de resolver).

(2) Ha de existir al menos un caso base para evitar que la recurrencia sea infinita.

Si ocurren (1) y (2) tenemos garantizado que el programa finalizara en algún momento.

En el caso de `factorialRecur(), cada llamada recursiva se realiza con un problema de tamaño n-1.`

El caso base ocurre cuando n = 1.

Iterativo Vs Recursivo

Las versiones recursivas suelen ser más breves(menos líneas de código), pero más lentas (tiempo de ejecución).

Y las versiones iterativas suelen ser más declarativas (esta claro como se resuelve el problema), pero se ven en problemas a la hora de recorrer estructuras de naturaleza recursiva, por ejemplo, un árbol binario.

Por lo tanto, no hay una regla general que indique cuando resolver un problema de una manera u otra.

Tipos de recursión

Hay dos tipos de recursión.
Una recursión es lineal, si existe una única llamada recursiva, por ejemplo, en `factorialRecur()`.
La recursión es no lineal si existen dos o más llamadas recursivas, como por ejemplo, en fibonacciRecur() (Ver Código 2a).

```
def fibo(n):
  if n < 0:
    raise Exception("Debe ser un entero positivo o 0")
  if n == 0:
    return 0
  if n == 1:
    return 1
  if n > 1:
    return fibo(n-1) + fibo(n-2)
```

Código 2a: recursión no lineal

Ejercicio0: Tardara mucho fiboRecur?

Ejercicio1: Como demostrar que fiboRecur() finaliza?
(1)Todas las llamadas recursivas se invocan con un valor < n
(2)Tiene un caso no recursivo en el cual el programa finaliza.

Ejercicio2: Como será la versión iterativa?

```
def fiboIter(n):
  n0 = 0
  n1 = 1
  n2 = n0 + n1

  if n==0:
    return n0
  if n==1:
    return n1
  if n>2:
    for i in range(2,n+1):
      n2 = n1 + n0
      n0 = n1
      n1 = n2
  return n2
```

Código 2b: fibonacci iterativo

Observaciones:

El código 2a y el código 2b hacen lo mismo?

Lamentablemente no hay una formula general, para demostrar que dos códigos hacen lo mismo.

Afortunadamente para fibo, si se puede demostrar que ambos códigos son equivalentes. Pero la demostración escapa al alcance de la materia.

Sin embargo, en la clase de complejidad demostraremos que versión es la que tarda más tiempo en finalizar.

Ejemplo2: Búsqueda binaria en vectores ordenados

```
#devuelvo el índice dentro de lista o -1
#PRE: lista debe estar ordenada, j=len(lista)-1
def busquedaBinaria(lista,elem,i,j):
  if i>j:
    return -1
  medio = (i+j) // 2
  if lista[medio] == elem:
    return medio
  if lista[medio] < elem:
    return busquedaBinaria(lista,elem,medio+1,j)
  if lista[medio] > elem:
    return busquedaBinaria(lista,elem,i,medio-1)

lista = [1,2,5,7,11,13]
# buscamos el 13, tiene indice 5
print(busquedaBinaria(lista,13,0,len(lista)-1))
# buscamos el 20, retornamos - 1
print(busquedaBinaria(lista,20,0,len(lista)-1))
```