Universidad Nacional de San Martín

Licenciatura en Ciencia de Datos

Algoritmos II

Introducción a recursión

-La **recursión** es una **técnica de programación** en la que **una función se llama a sí misma** para resolver un problema. Se utiliza para **descomponer** problemas complejos en **subproblemas** más pequeños y manejables.

```
funcion resolver(problema)
    si problema es simple entonces
        devolver solucion
    sino
        dividir problema en subproblema1..N
        resolver(subproblema1)
        resolver(subproblema2)
        ..
        resolver(subproblemaN)
        combinar_soluciones
        devolver solución
    finSi
finFuncion
```

Introducción a recursión (cont)

- **Caso Base**: Es la condición que **termina las llamadas recursivas**. Sin un caso base, la recursión continuaría indefinidamente, causando un **desbordamiento de la pila** (stack overflow).

-**Llamada Recursiva:** Es cuando una función **se llama a sí misma** con argumentos modificados, **acercándose al caso base**.

-Para implementar una solución recursiva las **instancias de recursión o subestructuras recursivas** deben **relacionarse** mediante **un ORDEN BIEN FUNDADO**: cualquier subconjunto de **elementos ordenados** debe tener un **elemento mínimo**.

Introducción a la recursión: ejemplo

Ejemplo factorial: El factorial de un número (n!) se define como el producto de todos los enteros positivos hasta n. Por definición 0! = 1.

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

Ejercicio

Definir lo siguiente:

a. Una función recursiva digitos, que dado un número entero, retorne su cantidad de dígitos.

b. Una función recursiva reversa_num que, dado un número entero, retorne su imagen especular. Por ejemplo: reversa_num(345) = 543

c. Una función recursiva suma_digitos que, dado un número entero, retorne la suma de sus dígitos.

d. Una función recursiva que retorne los dos valores anteriores a la vez como un par, aprovechando la recursión. (retorne tanto la suma de los dígitos como el número invertido a la vez)

Tipos de recursión

LA RECURSIÓN PUEDE PRESENTARSE:

1) EN **PROCEDIMIENTOS O FUNCIONES**
2) EN **ESTRUCTURAS DE DATOS**

Clasificación según referencias

Recursión simple

-Existe una **única llamada recursiva** en una **función** / una **estructura de datos** se compone con un **elemento del mismo tipo** de datos que se está definiendo

Ejemplo:

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return factorial(n-1) * n
```

cuando se llega al **caso base factorial(1)**, donde aún quedan **apilados todos los casos previos** ya que **no finalizan** hasta se aplique la **multiplicación por n**

Clasificación según referencias

Recursión múltiple

-Las **referencias a sí mismo superan** la **unidad**. Casos particulares: **recursión doble** o **recursión triple** si son dos o tres referencias recursivas.

Ejemplo:

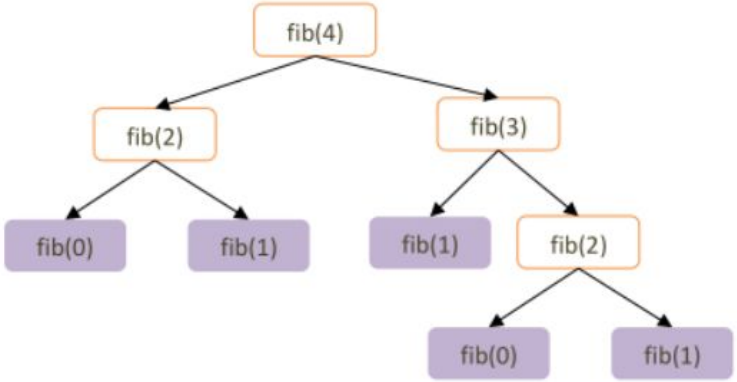
```
def fibonacci(n: int) -> int:
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2) #suma de numeros anteriores
```

Clasificación según referencias

Recursión múltiple (cont)

-En general es **muy ineficiente** ya que se genera un **árbol de invocaciones**, a diferencia de la **recursión simple** que es una **secuencia de invocaciones**.

-En casos donde se **invoquen varias veces una misma instancia recursiva** y se respete la **transparencia referencial**, se puede usar **MEMOIZATION: almacena el resultado en memoria para evitar recomputarlo** en próximas invocaciones.



Clasificación según dirección

- La **recursión ocurre o no** en la **misma función/estructura** de datos.

Recursión directa

-Realiza su **invocación** a sí misma **dentro de su propio cuerpo.** (lo que vimos hasta ahora)

Clasificación según dirección

Recursión indirecta

-Una **operación vuelva a ser invocada** a así misma **a partir de invocaciones intermediarias** a otras operaciones dentro de la instancia recursiva:

operacion1 -> operacion2 -> operacion3 -> operacion1

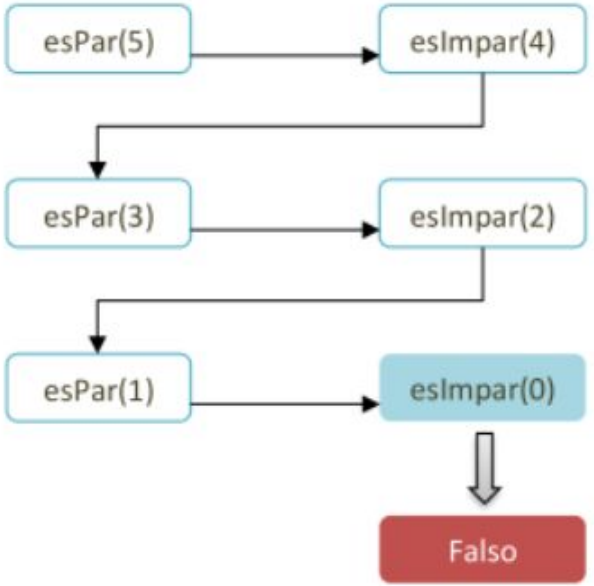
```
def operacion1(n):
    if n > 0:
        print(f"operacion1, n = {n}")
        operacion2(n - 1)
def operacion2(n):
    if n > 0:
        print(f"operacion2, n = {n}")
        operacion3(n - 1)
def operacion3(n):
    if n > 0:
        print(f"operacion3, n = {n}")
        operacion1(n - 1)

# Llamada inicial
operacion1(3)
operacion1, n = 3
operacion2, n = 2
operacion3, n = 1
```

Recursión mutua (cont)

-Caso especial **Recursión Mutua**: intervienen **sólo dos componentes.**

```
def es_par(n: int) -> bool:
  return n == 0 or es_impar(n - 1)
def es_impar(n: int) -> bool:
  return False if n == 0 else es_par(n - 1)
print(es_par(10))    # True
print(es_par(9))     # False
print(es_impar(4))   # False
print(es_impar(7))   # True
```



Clasificación según visibilidad

Recursión explícita

-Una función se llama a sí misma **directamente en su bloque de código.** Ej: fibonacci, factorial.

Recursión implícita

-La **invocación inicial** se realiza mediante una **función** que **no** tiene **recursión**, que **invoca a otra** que **sí presenta**.

Ejemplo:

```
def mostrar_paridad(n: int) -> None:
    if es_par(n):
        print(f'{n} es par')
    else:
        print(f'{n} es impar')
```

Ejercicio

Adaptar la solución propuesta con recursión mutua para determinar si un número es par o impar pero permitiendo aceptar también números negativos.

```
def es_par(n: int) -> bool:
  return n == 0 or es_impar(n - 1)
def es_impar(n: int) -> bool:
  return False if n == 0 else es_par(n - 1)
print(es_par(10))    # True
print(es_par(9))     # False
print(es_impar(4))   # False
print(es_impar(7))   # True
```

Ejercicio

Definir la operación procedimiento pares, que dado un número entero, muestre todos los pares de números enteros positivos que son suma del número entero dado. Por ejemplo, 5 = (1 , 4), (2, 3).

Ejercicio

Definir la función desde_hasta recursiva que dados dos números enteros retorne una lista de números consecutivos donde el primer elemento de la lista resultante sea el primer elemento dado, y el último elemento de la lista resultante sea el segundo elemento dado.

Redefinir las funciones sumatoria y factorial utilizando desde_hasta.