Guía de Ejercicios 10: Recursión algorítmica

Objetivos:

- Presentar la recursión algorítmica como una forma de encarar la resolución de problemas.
- Identificar los casos base y recursivo en problemas sencillos sobre enteros y listas.

Ejercicio 1. Escribir funciones recursivas para resolver los siguientes problemas.

(a) `pot2(n)`: Dado un entero `n` ≥ 0, calcular 2^{n}. Ejemplos: `pot2(0)` $\to 1$; `pot2(7)` $\to 128$.

(b) `pot_a(a, n)`: Dados dos enteros `n` ≥ 0 y `a`, calcular a^{n}. Ejemplos: `pot_a`$(3, 0) \to 1$; `pot_a`$(-2, 7) \to -128$. Además, en particular `pot_a`$(0, 0)$ debe devolver 1.

(c) `producto(n, m)`: Dados dos enteros `n` $\geq 0, m \geq 0$, calcular `n` $*$ `m`. Sugerencia: hacer la recursión sobre el parámetro `n`, dejando fijo el valor de `m`.

(d) `es_par(n)`: Determinar si un entero `n` ≥ 0 es par (o sea, debe devolver `True` si es par y `False` en caso contrario). Sugerencia: pensar cómo podría ayudar restarle 2 a `n`.

Ejercicio 2. Escribir funciones recursivas para resolver los siguientes problemas.

(a) `productoria(xs)`: Dada una lista no vacía de enteros `xs`, calcular el resultado de multiplicar todos los números de `xs`.

(b) `cantidad_ocurrencias(x, xs)`: Dada una lista de enteros `xs` y un entero `x`, devolver la cantidad de veces que aparece `x` en `xs`.

(c) `max_pos(xs)`: Dada una lista no vacía de enteros `xs`, devolver la posición del elemento más grande. En caso de empate, devolver la posición de la primera aparición.

(d) `contar_coincidencias(xs)`: Dada una lista de enteros `xs`, contar cuántas veces es cierto que la `i`-ésima posición tiene el número `i` (es decir, cuántas veces `xs[i]==i`).

(e) `sumar_posiciones_pares(xs)`: Dada una lista de enteros, sumar los elementos en sus posiciones pares. Ejemplo: `sumar_posiciones_pares([1,2,9,4,3])` devuelve $13 = 1 + 9 + 3$.

Ejercicio 3. La sucesión de Fibonacci es una sucesión en la cual cada término se define como la suma de los dos anteriores, comenzando con 0 y 1. A continuación se observan los primeros diez términos de la sucesión:

$$F_0 = 0; F_1 = 1; F_2 = 1; F_3 = 2; F_4 = 3; F_5 = 5; F_6 = 8; F_7 = 13; F_8 = 21; F_9 = 34; \ldots$$

(a) Escribir una función recursiva `fibonacci(n)` que dado un entero `n` ≥ 0, devuelva el término F_n de la sucesión de Fibonacci.

(b) Ejecutar `fibonacci(50)`. ¿Por qué tarda tanto? (Probablemente sea necesario cortar la ejecución después de un tiempo.) Para buscar una respuesta (informal), hacer un seguimiento de la evaluación de un ejemplo chico, como `fibonacci(6)`.

Ejercicio 4. Escribir una función recursiva `dict_longitudes`, que dada una lista `xs` de strings devuelva un diccionario de strings a enteros, que tenga los elementos de `xs` como claves y sus respectivas longitudes como valores.

```
def dict_longitudes(xs:list[str]) -> dict[str,int]:
    ''' Requiere: nada.
        Devuelve: un dict con los elementos de xs como claves,
                  y sus respectivas longitudes como valores.
    '''
    ### COMPLETAR ###
```

Los siguientes ejemplos describen el comportamiento esperado. Se muestra el valor pasado al parámetro `xs` y el valor que debe retornar la ejecución de `dict_longitudes(xs)`.

`xs`	Diccionario devuelto (no importa el orden de las claves)
`[]`	`{}`
`['árbol', 'flor', '', 'césped']`	`{'árbol': 5, 'flor': 4, '': 0, 'césped': 6}`
`['hola', 'hola', 'chau', 'chau']`	`{'hola': 4, 'chau': 4}`

Ejercicio 5. Escribir qué se imprime por pantalla luego de cada `print` en las líneas indicadas con `(1)`, `(2)`, `(3)`, etc.

```
def f(xs:list[int]) -> str:
    if len(xs)==0:
        return ''
    else:
        n:int = xs[0]
        s:str = f(xs[1:])
        return str(n) + s*n

print(f([]))             # (1)
print(f([1]))            # (2)
print(f([2,1]))          # (3)
print(f([3,1]))          # (4)
print(f([1,2,3]))        # (5)
print(f([3,2,1]))        # (6)
print(f([0,99999,1]))    # (7)
```

Justificar las respuestas, mostrando paso a paso cómo se evalúan las expresiones.

Sugerencia: Cuando sea posible, utilizar la resolución de expresiones anteriores para justificar la resolución de las siguientes.

Ejercicio 6. Escribir el código de la siguiente función **recursiva** `espejar`, de manera que cumpla con la especificación dada y pase los casos de test definidos abajo.

```
def espejar(xs:list[int]) -> list[int]:
  ''' Requiere: nada.
      Devuelve: una nueva lista que contiene los mismos elementos
                que xs, pero en el orden inverso.'''
  ### COMPLETAR ###

import unittest
class TestlistaEspejada(unittest.TestCase):
  def test_espejar(self):
    self.assertEqual(espejar([]), [])
    self.assertEqual(espejar([-3]), [-3])
    self.assertEqual(espejar([1,2,2,5,9]), [9,5,2,2,1])
    self.assertEqual(espejar([1,2,5,3]), [3,5,2,1])
```

Ejercicio 7. Determinar qué se imprime por pantalla en las líneas indicadas con `(N)` en el siguiente código. Justificar las respuestas y mostrar paso a paso cómo se resuelven las expresiones.

```
def f(n:int) -> str:
  if n==0:
    vr = str(n)
  else:
    vr = str(n) + f(n-1) + str(n)
  return vr

def g(xs:list[int], s:str) -> str:
    if len(xs) == 0:
        return ''
    elif len(xs) == 1:
        return str(xs[0])
    else:
        med:int = len(xs) // 2
        return g(xs[:med], s) + s + g(xs[med:], s)

def h(xs:list[str], s:str) -> bool:
    if len(xs) == 0:
        return False
    elif len(xs) == 1:
        return xs[0]==s
    else:
        med:int = len(xs) // 2
        return h(xs[:med], s) or h(xs[med:], s)

print(f(1))    # (1)
print(f(3))    # (2)

print(g([], 'hola'))            # (3)
print(g([3, 6, 2, 1], ','))     # (4)
print(g([1, 2, 3], '+'))        # (5)

print(h(['a', 'b', 'c', 'a', 'e'], 'a'))  # (6)
print(h(['a', 'b', 'c', 'a', 'e'], 'b'))  # (7)
print(h(['a', 'b', 'c', 'a', 'e'], 'x'))  # (8)
```