Ejercicios de Recursividad en Python — Soluciones

5 ejercicios con explicación, complejidad y código probado

1) Factorial (Básico)

Implementa una función recursiva factorial(n) que calcule n! para n ≥ 0. Se prohíben ciclos; usa caso base y paso recursivo.

Complejidad (resumen):

$T(n) = T(n-1) + O(1) \Rightarrow O(n)$ tiempo, $O(n)$ profundidad.

Código:

```
from typing import List, Union, Dict

def factorial(n: int) -> int:
    """
    Calcula n! recursivamente.
    Precondición: n >= 0
    """
    if n < 0:
        raise ValueError("factorial: n debe ser >= 0")
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)

# Pruebas
assert factorial(0) == 1
assert factorial(1) == 1
assert factorial(5) == 120
```

2) Fibonacci (Básico–Intermedio)

Implementa fibo(n) para el n-ésimo número de Fibonacci (0-index). Incluye versión simple (exponencial) y versión recursiva con memoización.

Complejidad (resumen):

Versión simple: O(φ^n) tiempo. Con memoización: O(n) tiempo, O(n) espacio.

Código:

```
from typing import Dict

def fibo(n: int) -> int:
    """
    Regresa el n-ésimo número de Fibonacci (0-index)
recursivamente (versión simple).
    Precondición: n >= 0
    Nota: Esta versión es exponencial; se incluye por claridad
teórica.
    """
    if n < 0:
        raise ValueError("fibo: n debe ser >= 0")
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibo(n - 1) + fibo(n - 2)

def fibo_memo(n: int, memo: Dict[int, int] | None = None) -> int:
    """
    Fibonacci recursivo con memoización (O(n)).
    """
    if n < 0:
        raise ValueError("fibo_memo: n debe ser >= 0")
    if memo is None:
        memo = {0: 0, 1: 1}
    if n in memo:
        return memo[n]
    memo[n] = fibo_memo(n - 1, memo) + fibo_memo(n - 2, memo)
    return memo[n]

# Pruebas
assert fibo(0) == 0 and fibo(1) == 1 and fibo(7) == 13
assert fibo_memo(0) == 0 and fibo_memo(1) == 1 and fibo_memo(30)
== 832040
```

3) Palíndromo (Intermedio)

Implementa es_palindromo(s) que verifique si s es palíndromo de forma exacta (sensible a espacios y mayúsculas). Se incluye variante normalizada que ignora espacios y mayúsculas.

Complejidad (resumen):

$T(n) = T(n-2) + O(1) \Rightarrow O(n)$ tiempo, $O(n)$ profundidad.

Código:

```
def es_palindromo(s: str) -> bool:
    """
    Verifica recursivamente si s es palíndromo (exacto; sensible
a espacios y mayúsculas).
    """
    if len(s) <= 1:
        return True
    if s[0] != s[-1]:
        return False
    return es_palindromo(s[1:-1])

def es_palindromo_normalizado(s: str) -> bool:
    t = "".join(ch.lower() for ch in s if not ch.isspace())
    return es_palindromo(t)

# Pruebas
assert es_palindromo("reconocer") is True
assert es_palindromo("python") is False
assert es_palindromo("") is True
assert es_palindromo("a") is True
assert es_palindromo_normalizado("Anita lava la tina") is True
```

4) Máximo de una lista (Intermedio)

Implementa maximo(lst) que devuelva el máximo de una lista no vacía sin usar max() ni ciclos. Divide en cabeza y resto; compara recursivamente.

Complejidad (resumen):

$T(n) = T(n-1) + O(1) \Rightarrow O(n)$ tiempo, $O(n)$ profundidad.

Código:

```
from typing import List

def maximo(lst: List[int]) -> int:
    """
    Devuelve el máximo de una lista no vacía recursivamente.
    Precondición: len(lst) >= 1
    """
    if not lst:
        raise ValueError("maximo: la lista no debe estar vacía")
    if len(lst) == 1:
        return lst[0]
    max_resto = maximo(lst[1:])
    return lst[0] if lst[0] >= max_resto else max_resto

# Pruebas
assert maximo([3]) == 3
assert maximo([3, 10, -2, 7]) == 10
assert maximo([-5, -1, -9]) == -1
```

5) Aplanar lista anidada (Intermedio–Avanzado)

Implementa aplanar(x) para aplanar una lista potencialmente anidada de enteros, preservando el orden. Sin ciclos; usa recursión sobre cabeza y resto.

Complejidad (resumen):

Visita cada elemento al menos una vez. O(N) sobre el número total de items; tener en cuenta que concatenaciones pueden costar en Python.

Código:

```
from typing import List, Union

Anidado = List[Union[int, "Anidado"]]

def aplanar(x: Anidado) -> List[int]:
    """
    Aplana una lista potencialmente anidada de enteros,
preservando el orden.
    Ejemplo: [1, [2, [3,4], 5], [], 6] -> [1,2,3,4,5,6]
    """
    if not x:
        return []
    cabeza, resto = x[0], x[1:]
    if isinstance(cabeza, int):
        return [cabeza] + aplanar(resto)
    return aplanar(cabeza) + aplanar(resto)

# Pruebas
assert aplanar([]) == []
assert aplanar([1,2,3]) == [1,2,3]
assert aplanar([1,[2,[3,4],5],[],6]) == [1,2,3,4,5,6]
```