# 0. Práctica Recursión

TUIA - Programación 2

2025 - 2C

## Recursión

### Ejercicio 0

Escriba una función recursiva `factorial` que reciba un numero natural `n` y calcule su factorial `n!`.

```
>>> factorial(1)
1
>>> factorial(3)
6
>>> factorial(7)
5040
```

### Ejercicio 1

Escriba una función recursiva que calcule el *n-ésimo* número triangular (i.e. el número `1 + 2 + 3 + ... + n`).

### Ejercicio 2

Escriba una función recursiva `fibonacci` que calcule el *n-ésimo* número de Fibonacci.

```
>>> fibonacci(0)
0
>>> fibonacci(4)
3
>>> fibonacci(8)
21
```

### Ejercicio 3

1. Escriba una función recursiva `print_string` que reciba como parámetro un número natural `n` y una string `s` y escriba por pantalla `n` veces el valor de la string `s`.

```
>>> print_string(0, "Test")
>>> print_string(3, "Test")
Test
Test
Test
```

2. Escriba una función recursiva `repeat_string` que reciba como parámetro un número natural `n` y una string `s` y devuelva el valor de `n` concatenaciones de la string `s`.

```
>>> repeat_string(0, "Test")
''
>>> repeat_string(4, "Test")
'TestTestTestTest'
```

## Ejercicio 4

Convierta la siguiente función iterativa a una definición recursiva.

```
def iterativa(l: list[int]) -> int:
    c = 1
    for i in l:
        c = c * i
    return c
```

## Ejercicio 5

Escriba una función recursiva `find_max` que encuentre el mayor elemento de una lista de números naturales.

```
>>> find_max([])
0
>>> find_max([9])
9
>>> find_max([1, 3, 2])
3
```

## Ejercicio 6

Escriba una función recursiva `power` que reciba dos números naturales `a` y `b` y calcule la potencia $a^b$.

```
>>> power(3, 0)
1
>>> power(4, 2)
16
```

## Ejercicio 7

Escriba una función recursiva `count_digits` que calcule la cantidad de dígitos de un número natural.

```
>>> count_digits(0)
1
>>> count_digits(13)
2
>>> count_digits(85041)
5
```

## Ejercicio 8

Escriba una función `reverse_string` que compute el reverso de una string. Implementar una definición iterativa y una recursiva.

**Nota**: No utilice la función *built-in* `reversed` en su solución, ni el método `reversed`.

```
>>> reverse_string("")
''
>>> reverse_string("Hola")
'aloH'
```

## Ejercicio 9

Escriba una función recursiva `replicate(l, n)` que replique `n` veces cada elemento en la lista `l`.

```
>>> replicate([1, 5, 4], 2)
[1, 1, 5, 5, 4, 4]
```

## Ejercicio 10

Escriba una función recursiva `is_palindrome` que compute si una string es un palíndromo.

**Nota**: Un palíndromo es una secuencia de caracteres que se lee igual hacia adelante que hacia atrás.

```
>>> is_palindrome("")
True
>>> is_palindrome("a")
True
>>> is_palindrome("ejemplo")
False
>>> is_palindrome("sometemos")
True
```

## Ejercicio 11

Considere la siguiente función recursiva:

```
def mystery(a: int, b: int) -> int:
    if (b == 0):
        return a
    return mystery(2 * a, b - 1)
```

1. ¿Qué muestra por pantalla el siguiente código? Intente deducirlo sin ejecutarlo.

```
>>> print(mystery(7, 3))
```

2. ¿Cuántas veces se invoca recursivamente `mystery` en la llamada del item anterior?
3. De manera general: ¿qué muestra por pantalla la llamada `f(x, 3)` para un `x` cualquiera? y ¿qué muestra por pantalla la llamada `f(x, y)` para `x`, `y` cualesquiera?

## Ejercicio 12

Escriba una función recursiva `potencia(a: int, b: int)` que reciba dos números enteros (incluyendo negativos) `a` y `b` y calcule la potencia $a^b$. Para esto, reutilice la función `power` definida en el Ejercicio 6.

```
>>> potencia(4, -1)
0.25
>>> potencia(-8, -2)
0.015625
```

## Ejercicio 13

Escriba una función `average` que calcule el promedio de una lista de números. Para esto, defina una función recursiva auxiliar `aux` que calcule tanto la cantidad y la suma de los elementos de la lista.

```
def aux(lista: list[float]) -> tuple[int, float]:
    # Completar

def average(lista: list[float]) -> float:
    # Completar
```

## Ejercicio 14

1. Escribir una función recursiva `posicion(a: str, b: str) -> int` que calcule la posición de la primer ocurrencia de `b` como substring en `a`.

```
>>> posicion("Wubba Lubba Dub Dub!", "ubba")
1
>>> posicion("Ñam fri fruli fali fru", "fru")
8
```

**Nota**: Puede ser de utilidad el método de string `str.startswith` que indica si una cadena comienza con una subcadena dada. Puede leer más sobre este método en la documentación oficial: https://docs.python.org/3/library/stdtypes.html#str.startswith

2. Escribir una función recursiva `posicion_maybe(a: str, b: str) -> (int|None)` con el mismo comportamiento que la función `posicion` del item anterior, que devuelva `None` en el caso que la substring `b` no se encuentre en `a`.

```
>>> print(posicion_maybe("Usted es el Jefe de los Minisuper?", "Si"))
None
```

## Ejercicio 15

La implementación recursiva del cálculo del número de Fibonacci es ineficiente porque muchas de las ramas calculan reiteradamente los mismos valores.

La técnica de `memoización` o `cache` se utiliza para optimizar los tiempos de cómputo mediante el almacenamiento de los resultados en memoria evitando así recalcularlos.

**Nota**: https://es.wikipedia.org/wiki/Memoizaci%C3%B3n

1. Escriba una función recursiva `fibonacci` que calcule el *n-ésimo* número de Fibonacci, almacenando los valores calculados en un diccionario definido de manera global.

```
memo: dict[int, int] = { 0: 0, 1: 1 }

def fibonacci(n: int) -> int:
    # Completar
```

2. Escriba una función recursiva `fibonacci_memo` que se comporte igual que la función `fibonacci` del item anterior. En vez de utilizar un diccionario global para la memoización, el mismo debe ser pasado como un argumento de la función recursiva `fibonacci_memo`.

```
def fibonacci_memo(n: int, memo: dict[int, int]) -> tuple[int, dict[int, int]]:
    # Completar
```

```
def fibonacci(n: int) -> int:
    ret, memo = fibonacci_memo(n, { 0: 0, 1: 1 })
    return ret
```