



Programación  
Licenciatura en Agroinformática  
Licenciatura en Bioinformática

# Tema 08: Recursividad

# Funciones dentro de funciones

- Al definir una función, es posible incorporar sentencias que llamen a otras funciones.
- La función original continúa luego de obtener los resultados de la función a la que llama.

# Funciones dentro de funciones

- Ejemplo: cálculo del desvío estándar de una lista de valores.

```
array = [1, 3, 5, 7]  
desv = desvio(array)  
print(desv)
```

# Funciones dentro de funciones

- Ejemplo: cálculo del desvío estándar de una lista de valores.

```
array = [1, 3, 5, 7]  
desv = desvio(array)
```



```
prom = media(array)  
# mas acciones ...  
return sd
```



```
print(desv)
```

# Funciones dentro de funciones

- Ejemplo: cálculo del desvío estándar de una lista de valores.

```
array = [1, 3, 5, 7]  
desv = desvio(array)
```


```
print(desv)
```

Programa principal




```
prom = media(array)
```


```
# mas acciones ...  
return sd
```



Función “desvio”



```
suma, cant = 0, 0  
for i in array:  
    # acciones...  
return promedio
```



Función “media”

# Recursividad

- Una función es aquella que se llama a sí misma.

```
r = func_recurv(var)
```



```
r2 = func_recurv(var2)  
return r2
```



```
r3 = func_recurv(var3)  
return r3
```



...

# Cálculo del factorial

- El factorial de un entero no negativo  $n$ , escrito  $n!$  (y pronunciado  $n$  factorial), es el producto

$$n! = n * (n - 1) * (n - 2) * \dots * 1$$

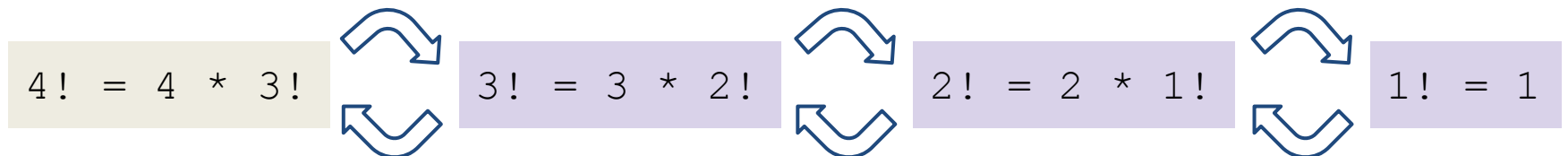
- Por ejemplo:

$$5! = 5 * 4 * 3 * 2 * 1$$

$$4! = 4 * 3 * 2 * 1$$

- También puede escribirse como:

$$n! = n * (n - 1)!$$



# Componentes de la recursividad

- Todo algoritmo recursivo tiene que tener, al menos, 2 componentes:
  - Componente recursivo.
  - Componente base.
- El componente o caso base es un caso que tiene una solución directa (no recursiva).

$$f(n) \begin{cases} 1 & n \leq 1 \\ n * f(n - 1) & n > 1 \end{cases}$$



# Recursión vs Iteración



- Por definición, todo problema que pueda resolverse mediante recursión se puede resolver por iteración.
- Las soluciones recursivas suelen ser más lentas y demandar más memoria, pero suelen ser más fáciles de comprender, implementar y depurar.
- Si una solución de un problema se puede expresar iterativa o recursivamente con igual facilidad, es preferible la solución iterativa.

# Implementando función "factorial"

```
def factorial_recursivo (n):  
    if n <= 1:      # Caso base  
        return 1  
    else:           # Caso recursivo  
        resultado = n * factorial_recursivo(n-1)  
        return resultado  
  
fact_rec = factorial_recursivo(10)  
print("Recursivo: ", fact_rec)
```

```
Recursivo: 3628800
```

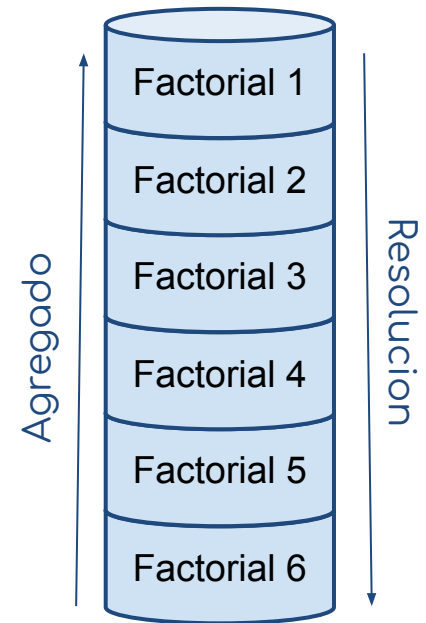
# Implementando función "factorial"

```
def factorial_recursivo (n):  
    if n <= 1:      # Caso base  
        return 1  
    else:           # Caso recursivo  
        resultado = n * factorial_recursivo(n-1)  
        return resultado  
  
def factorial_iterativo (n):  
    resultado = 1  
    for i in range (2,n+1):  
        resultado *= i  
    return resultado  
  
fact_rec = factorial_recursivo(10)  
fact_iter = factorial_iterativo(10)  
print("Recursivo: ", fact_rec, "Iterativo", fact_iter)
```

```
Recursivo: 3628800 Iterativo 3628800
```

# Límite de recursividad

- Por cada llamada dentro de una función recursiva, se agrega un valor a una **pila de recursión**.
- Esta pila mantiene el orden en que se deben resolver las distintas llamadas a la función recursiva.
- Por defecto, en Python, esta cola acepta acepta 1000 valores (luego da error).



# Precauciones

- Siempre debe haber, al menos, un caso base, sino la llamada recursiva se hace **infinita**.
- Si la llamada anterior requiere un resultado de una siguiente, siempre debe haber un *return*.
- Si se trabaja con listas (u otros tipos de datos abstractos), los resultados pueden almacenarse en ellos (sin necesidad de retornarlos)

# Ejemplo 1

- Implemente la operación “multiplicación” como una suma recursiva.

Pseudocódigo:

Recibir dos números: multiplicando y multiplicador

si multiplicador == 1:

    devolver multiplicando

sino:

    calcular multiplicando + f(multiplicando, multiplicador-1)

# Ejemplo 1

```
def mult (multiplicando, multiplicador):  
    print("multiplicando: ", multiplicando)  
    print("multiplicador: ", multiplicador)  
    if multiplicador == 1:  
        return multiplicando  
    else:  
        return multiplicando + mult(multiplicando, multiplicador-1)  
  
resultado = mult(4,6)  
print (resultado)
```

# Ejemplo 1

```
(base) matias@rfgenom002:~$ python test.py
multiplicando: 4
multiplicador: 6

multiplicando: 4
multiplicador: 5

multiplicando: 4
multiplicador: 4

multiplicando: 4
multiplicador: 3

multiplicando: 4
multiplicador: 2

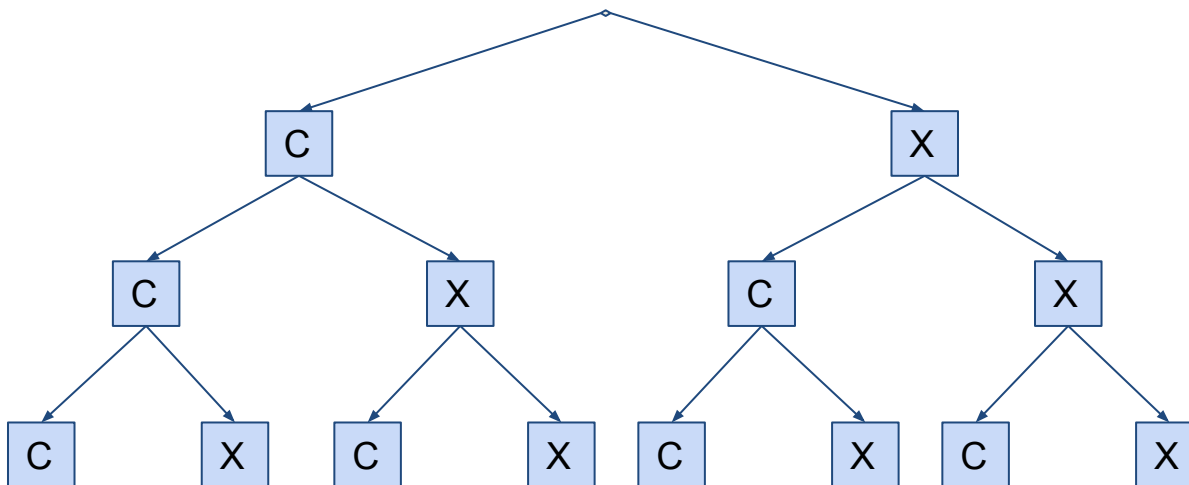
multiplicando: 4
multiplicador: 1
```



# Ejemplo 2

- Imprimir todas las combinaciones posibles de caras y cecas en un número de lanzamientos definido por el usuario.

Por ejemplo, para 3 lanzamientos, las posibilidades son:



CCC  
CCX  
CXC  
CXX  
XCC  
XCX  
XXC  
XXX

# Ejemplo 2

Pseudocódigo:

Recibir un número de lanzamientos y un string con combinaciones hasta el momento

si lanzamientos == 1:

    cara = string+"C"

    print(cara)

    ceca = string+"X"

    print(ceca)

sino:

    combinaciones(lanzamientos-1, string+"C")

    combinaciones(lanzamientos-1, string+"X")

# Ejemplo 2

```
def combinaciones(lanzamientos, lista, string = ""):
    if lanzamientos == 0:
        lista.append(string)
    else:
        combinaciones(lanzamientos-1, lista, string+"C")
        combinaciones(lanzamientos-1, lista, string+"X")

valor = int(input("Ingrese numero de lanzamientos: "))
lista = []
combinaciones(valor, lista)
print(lista)
```

# Ejemplo 2

```
(base) matias@Matias-Notebook:~$ python test.py  
CCC  
CCX  
CXC  
CXX  
XCC  
XCX  
XXC  
XXX
```