# Clase Nº 10

## Recursividad



# Definición

- Un objeto se define como ***recursivo*** cuando una parte de él está formada por el objeto mismo.
- La recursividad puede aplicarse en diversos aspectos de la vida cotidiana, tales como las imágenes, el idioma y también en la programación.

# Imágenes recursivas



# Definiciones recursivas

- **Un *identificador* es un nombre definido por el programador para denominar una variable o función.**
- **Algunas de las reglas que deben cumplir los identificadores pueden expresarse en forma recursiva.**

© Lic. Ricardo Thompson

# Definiciones recursivas

**Un identificador en Python es:**

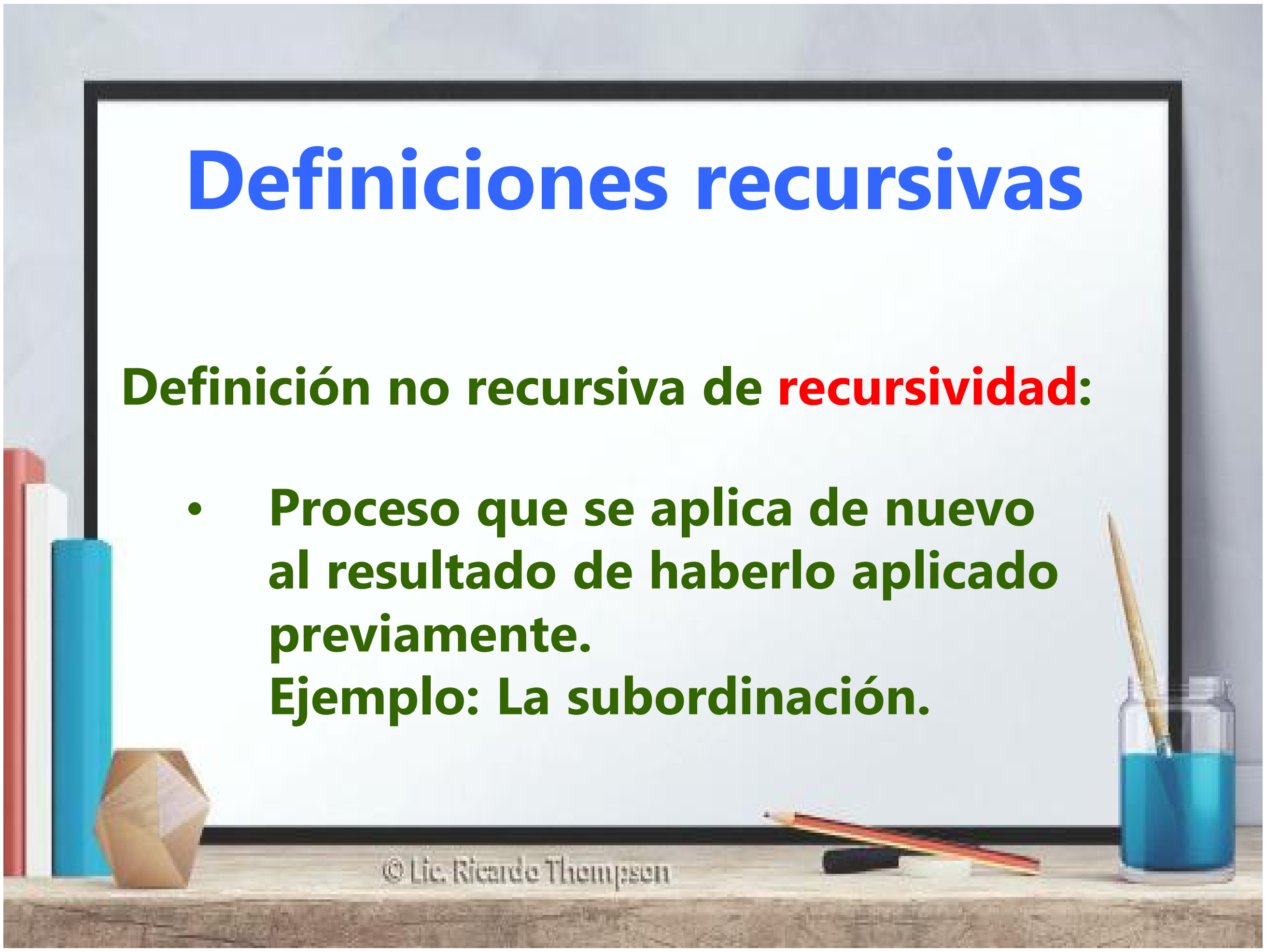
- **Una letra o guión bajo.**
- **Un identificador seguido por una letra, número o guión bajo.**



# Definiciones recursivas

**Definición recursiva de recursividad:**

- **Véase *recursividad***


Definiciones recursivas
Definición no recursiva de recursividad:
• Proceso que se aplica de nuevo al resultado de haberlo aplicado previamente.
Ejemplo: La subordinación.
© Lic. Ricardo Thompson


Funciones recursivas
• La recursividad aplicada a la programación se manifiesta en forma de funciones en las que una parte del trabajo lo realiza la misma función.
• En otras palabras, son funciones que se invocan a si mismas.
© Lic. Ricardo Thompson

# Función factorial

fact(4) = 4 * 3 * 2 * 1

fact(3) = 3 * 2 * 1 ➔ fact(4) = 4 * fact(3)

fact(2) = 2 * 1 ➔ fact(3) = 3 * fact(2)



# Función factorial

Generalizando:

- fact(n) = n * fact(n-1)
- fact(0) = 1 (por convención)

© Lic. Ricardo Thompson

# Función factorial

```
def fact(n):
    if n==0:
        return 1
    else:
        return n * fact(n-1)

# Programa principal
a=int(input("Ingrese un número entero: "))
print("El factorial de", a, "es", fact(a))
```



# Función factorial

Prueba de escritorio para n = 4

```
def fact(n):
    if n==0:
        return 1
    else:
        return n * fact(n-1)
```

| n | fact(n) |
|---|---|
| 4 | 4 * fact(3) |
| 3 | 3 * fact(2) |
| 2 | 2 * fact(1) |
| 1 | 1 * fact(0) |
| 0 | 1 |

# Función factorial

**Prueba de escritorio para n = 4**

```
def fact(n):
   if n==0:
      return 1
   else:
      return n * fact(n-1)
```

| n | fact(n) |
|---|---|
| 4 | 4 * fact(3) = 24 |
| 3 | 3 * fact(2) = 6 |
| 2 | 2 * fact(1) = 2 |
| 1 | 1 * fact(0) = 1 |
| 0 | 1 |



# Función factorial

```
def fact(n):
   if n==0:
      return 1
   else:
      return n * fact(n-1)

a=int(input("Ingrese un número entero: "))
print("El factorial de", a, "es", fact(a))
```

- `if n==0: return 1` → **Caso base**
- `else: return n * fact(n-1)` → **Caso recursivo**

© Lic. Ricardo Thompson

# Función factorial

- El **caso recursivo** es donde se realizan las llamadas recursivas. Suele ser el más común, es decir el que se ejecuta la mayoría de las veces.

- El **caso base** es donde se realiza una *salida no recursiva*. Suele ser único, o limitado a pocas alternativas.



# Función factorial

**Prueba de escritorio para n = -1**

```
def fact(n):
   if n==0:
      return 1
   else:
      return n * fact(n-1)
```

| n | fact(n) |
|---|---|
| -1 | -1 * fact(-2) |
| -2 | -2 * fact(-3) |
| -3 | -3 * fact(-4) |
| -4 | -4 * fact(-5) |
| | [ . . . ] |

# Función factorial

**Prueba de escritorio para n = -1**

**RecursionError:**
**maximum recursion depth exceeded in comparison**



# "Divide y Vencerás"

- **Es una técnica que ayuda a determinar si un problema es adecuado para recibir una solución recursiva.**
- **Consiste en particionar el problema global en problemas más pequeños, y volverlos a particionar hasta llegar a una solución elemental.**

# Potencia de un Nº natural

- $2^4 = 2 * 2 * 2 * 2$
- $2^3 = 2 * 2 * 2$ ➔ $2^4 = 2 * 2^3$
- $2^2 = 2 * 2$ ➔ $2^3 = 2 * 2^2$



# Potencia de un Nº natural

**Generalizando:**

- $a^b = a * a^{b-1}$
- $a^0 = 1$ (por convención)

# Potencia de un Nº natural

```
def potencia(a, b):
    if b==0:
        return 1
    else:
        return a * potencia(a, b-1)
```



# Teorema fundamental

*"Todo problema que admita*
*una solución iterativa*
*admite también*
*una solución recursiva*
*y viceversa"*

# Iteraciones y recursividad

**Ejemplo: Imprimir los números del 1 al 100**

```
def imprimir(n):
    if n>0:
        imprimir(n-1)
        print(n, end=" ")
```

**...que se invoca como:**

```
imprimir(100)
```



# Iteraciones y recursividad

**Prueba de escritorio para n = 100**

```
def imprimir(n):
    if n>0:
        imprimir(n-1)
        print(n, end=" ")

# Programa principal
imprimir(100)
```

| n | Imprime |
|---|---|
| 100 | |
| 99 | |
| 98 | |
| | [ . . . ] |
| 2 | |
| 1 | |
| 0 | |

# Iteraciones y recursividad

**Prueba de escritorio para n = 100**

```
def imprimir(n):
    if n>0:
        imprimir(n-1)
        print(n, end=" ")

# Programa principal
imprimir(100)
```

| n | Imprime |
|---|---|
| 100 | 100 |
| 99 | 99 |
| 98 | 98 |
| | [ . . . ] |
| 2 | 2 |
| 1 | 1 |
| 0 | |



# Iteraciones y recursividad

**¿Qué ocurre si invertimos las líneas del if?**

```
def imprimir(n):
    if n>0:
        imprimir(n-1)
        print(n, end=" ")
```

# Ventajas y desventajas

**Ventajas**

- Elegancia
- Simplicidad

**Desventajas**

- Lentitud
- Consumo de memoria



# Fibonacci

- Es una sucesión infinita de números naturales.
- Comienza con 0 y 1.
- A partir de allí cada término se calcula sumando los dos anteriores:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55…

# Fibonacci

**Formulación recursiva:**

- fib(n) = fib(n-1) + fib(n-2)
- fib(0) = 0
- fib(1) = 1

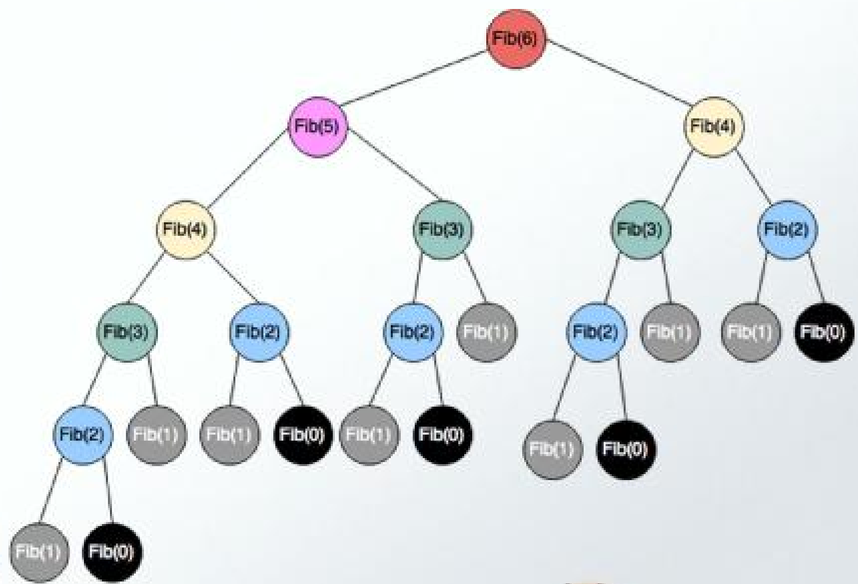
Casos base



# Fibonacci

```
def fib(n):
    if n==0 or n==1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

# Fibonacci






# Listas y Recursividad

- **Muchas operaciones sobre listas se realizan a través de ciclos.**
- **Como los ciclos pueden ser reemplazados por llamadas recursivas, es posible implementar cualquiera de estas operaciones a través de recursividad.**

# Listas y Recursividad

```
def imprimirlista(lista, inicio=0):
    # Versión 1, con dos parámetros
    if inicio<len(lista):
        print(lista[inicio], end=" ")
        imprimirlista(lista, inicio+1)
```

...que se invoca como:

```
imprimirlista(lista)
```



# Listas y Recursividad

```
def imprimirlista(lista):
    # Versión 2, con un solo parámetro y rebanadas
    if len(lista)>0:
        print(lista[0], end=" ")
        imprimirlista(lista[1: ])
```

...que se invoca como:

```
imprimirlista(lista)
```

# Listas y Recursividad

```
def buscarmayor(lista, inicio=0):
    # Versión 1, con dos parámetros
    if inicio<len(lista)-1:
        actual = lista[inicio]
        mayor = buscarmayor(lista, inicio+1)
        return actual if actual>mayor else mayor
    else:
        return lista[-1]  # Último elemento
```

...que se invoca como:

```
maximo = buscarmayor(lista)
```



# Listas y Recursividad

```
def buscarmayor(lista):
    # Versión 2, con un solo parámetro y rebanadas
    if len(lista)>1:
        actual = lista[0]
        mayor = buscarmayor(lista[1: ])
        return actual if actual>mayor else mayor
    else:
        return lista[0]  # Único elemento
```

...que se invoca como:

```
maximo = buscarmayor(lista)
```

# Programa completo

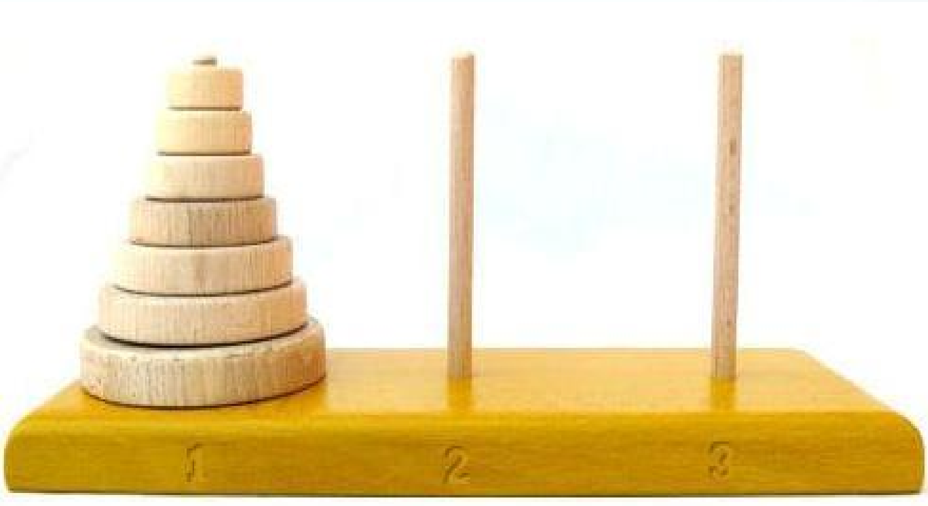
```
def buscarmayor(lista, inicio=0):
    if inicio<len(lista)-1:
        actual = lista[inicio]
        mayor = buscarmayor(lista, inicio+1)
        return actual if actual>mayor else mayor
    else:
        return lista[-1]  # Último elemento

def imprimirlista(lista, inicio=0):
    if inicio<len(lista):
        print(lista[inicio], end=" ")
        imprimirlista(lista, inicio+1)

# Programa principal
lista = [2,7,5,4,9,0,8,6]
imprimirlista(lista)
print()
maximo = buscarmayor(lista)
print("El mayor elemento de la lista es", maximo)
```



# Las Torres de Hanoi

- Es un pasatiempo que se presentó en Europa en 1883.
- El entretenimiento intenta reproducir una tarea que, según la leyenda, vienen desarrollando los monjes del templo de Brahma en la India.

# Las Torres de Hanoi





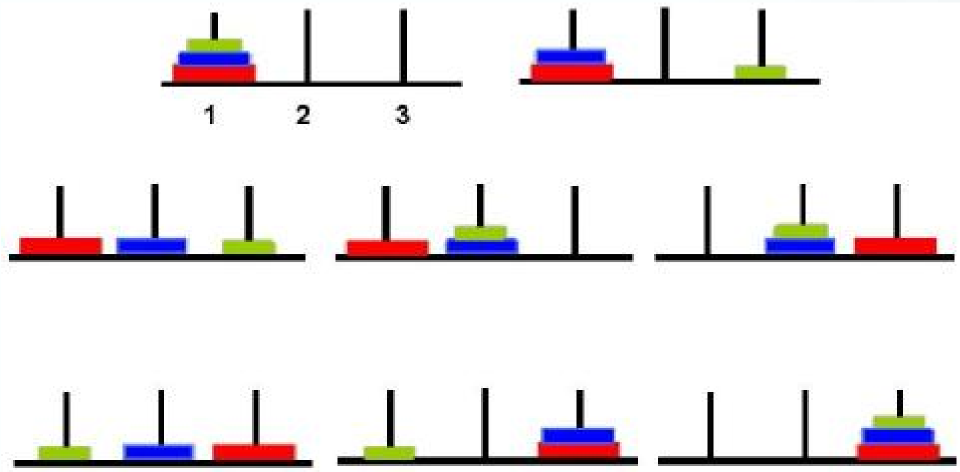

# Las Torres de Hanoi

- **El objetivo del juego consiste en trasladar la torre de 64 discos desde la aguja 1 a la aguja 3, respetando sólo dos reglas básicas:**


Las Torres de Hanoi
1. No se puede mover más de un disco por vez.
2. No se puede colocar un disco de mayor tamaño encima de otro de menor tamaño.
© Lic. Ricardo Thompson


Las Torres de Hanoi
• La tarea es tan larga que aún hoy los monjes continúan con ella.
• Según la leyenda, cuando terminen de trasladar la pirámide habrá llegado *el fin del mundo*.
© Lic. Ricardo Thompson

# Las Torres de Hanoi






# Las Torres de Hanoi

## Estrategia de resolución (1)


Las Torres de Hanoi
Estrategia de resolución (2)
© Lic. Ricardo Thompson


Las Torres de Hanoi
Estrategia de resolución (3)
© Lic. Ricardo Thompson

# Las Torres de Hanoi

## Estrategia de resolución (4)



# Las Torres de Hanoi

```
def mover(n, origen, destino, aux):
    if n>0:
        mover(n-1, origen, aux, destino)
        print("Muevo un disco de", origen, "a", destino)
        mover(n-1, aux, destino, origen)

# Programa principal
discos=int(input("Cantidad de discos? "))
mover(discos, 1, 3, 2)
```

# Las Torres de Hanoi

- La cantidad óptima de movimientos está dada por la fórmula $2^n - 1$, donde n es la cantidad de discos.
- Si n = 64 → $2^{64} - 1$ =

**18.446.744.073.709.551.615**

(≈ 18.4 trillones de movimientos)



# Las Torres de Hanoi

**A razón de 1 movimiento por segundo ésto requiere 585.000 millones de años, que equivale a 100 veces la edad del universo.**

# Para tener en cuenta

- Nunca debe verificarse el caso base mediante while o for.
- Las variables locales tienen una utilidad acotada.
- Es necesario utilizar parámetros adicionales y el valor de retorno para comunicar valores entre distintas llamadas recursivas.



# Ejercitación

- Práctica 8: Completa