Clase 5: Recursión

(Leer Capítulo 6 del Apunte)

En programación se habla de funciones recursivas cuando la definición de estas depende de sí mismas.

Se basa en el principio de la *inducción*:

- Caso base
- Caso n => n+1

Ejemplo de función recursiva: Potencia

$$
\begin{aligned}
2^4 &= 2 \cdot 2^{4-1} \\
&= 2 \cdot 2^3 \\
&= 2 \cdot (2 \cdot 2^2) \\
&= 2 \cdot (2 \cdot (2 \cdot 2^1)) \\
&= 2 \cdot (2 \cdot (2 \cdot (2 \cdot 2^0))) \\
&= 2 \cdot (2 \cdot (2 \cdot (2 \cdot 1))) \\
&= 16
\end{aligned}
$$

$$
a^b = \begin{cases} 1 & \text{si } b = 0 \\ a \cdot a^{b-1} & \text{si } b > 0 \end{cases}
$$

```
# potencia: num int -> num
# calcula el valor de una potencia de base elevado a exponente
# para exponentes enteros positivos
# ejemplo: potencia (4,5) debe dar 1024
def potencia(base, exponente):
    if exponente == 0:
        return 1
    else:
        return base*(potencia(base, exponente-1))

# test
assert potencia(4,5) == 1024
assert potencia(2,4) == 16
assert potencia(-1,5) == -1
assert potencia(3,0) == 1
```

```
potencia(-2,7)
```

- **propuesto:** modificar para permitir tanto exponentes positivos como negativos.

Características de las funciones recursivas

Es una función que se define en términos de sí misma es una ***función recursiva***. Estas funciones siempre deben tener:

- un caso base
- un caso recursivo
- cada llamada debe disminuir el tamaño del problema (converger al caso base)
- **Problema:** Contar dígitos de un entero. Ej: 245 tiene 3 dígitos, -4 tiene un dígito.
- **Definición:**
  - digitos(n) = 1 + digitos(n/10), para n>= 10
  - digitos(n) = 1, para |n|<10

```
#digitos: int->int
#cuenta digitos de un número entero
#ej: dígitos(245) debe ser 3
#ej: digitos(4) debe ser 1
def digitos(n):
    if abs(n) < 10 :
        return 1
    else:
        return 1 + digitos(n//10)
# tests
assert digitos(245)==3
assert digitos(-4)==1


digitos(-2456)
```

Ejemplo de función recursiva: Factorial

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Escriba la función `factorial` que reciba un entero y retorne su factorial

```
# factorial: int -> int
# calcula el valor factorial de n
# ejemplo: factorial(4) debe dar 24
def factorial(n):
    if n == 0:
        return 1
```

```
    else:
        return n * factorial(n-1)


# test
assert factorial(4) == 24
assert factorial(2) == 2
assert factorial(8) == 40320
```

```
factorial(0)
```

```
>>> factorial(10)
```

¿Cómo se ejecuta (desenrolla) una Función Recursiva?

Usualmente, para diseñar (programar) una función recursiva nos conviene razonar en términos de **caso base** y **caso recursivo**.

A veces puede ser útil *seguir la ejecución* de la función **paso a paso**, mirando qué se va calculando, y así entener lo que sucede en la llamada.

Entonces, ¿cómo se *ejecuta* una función recursiva? Para verlo, podemos imaginarnos que

- La primera invocación de la función "ocurre" en la parte del código donde está la función,
- Cada invocación (o llamada) a la función produce una **copia** de dicha función dentro del computador,
- Esta nueva **copia** se ejecuta, posiblemente generando nuevas copias en secuencia al llamarse recursivamente (decimos que *desenrolla* la recursión).
- Al final, la última copia **cae en el caso base** y retorna un valor, el cual es usado por la copia que hizo la invocación (la penúltima) para calcular su propio valor y retornarlo, *enrollando* nuevamente las llamadas recursivas.
- El valor devuelto es el valor retornado por la primera copia.

Veamos visualmente la ejecución de la función factorial.

¿Cómo podemos asegurarnos que los valores que reciben las funciones estén en un determinado rango?

- Usamos `assert`

```
# factorial: int -> int
# calcula el valor factorial de n
```

```
# ejemplo: factorial(4) debe dar 24
def factorial(n):
    assert (type(n)==int) and (n>=0), "Factorial no esta definido para n no entero, ni
    if n == 0:
        return 1
    else:
        return n*(factorial(n-1))

# test
assert factorial(4) == 24
assert factorial(2) == 2
assert factorial(8) == 40320
```

```
factorial(-1)
```

```
factorial(4)
```

Podemos usar lo aprendido para hacer un programa interactivo que lea una lista indeterminada de números e imprima su factorial

```
#calculaListaFactoriales: None -> None
#calcular factorial de lista de numeros ingresada por el teclado
#  (la lista termina con el valor "fin")
#ej: factoriales()

def calculaListaFactoriales():
    respuesta=input('n ?')
    if respuesta=="fin":
      return

    # caso no es igual a fin
    n = int(respuesta)
    factorial_n = factorial(n)
    print(str(n)+'!='+str(factorial_n))
    calculaListaFactoriales()
```

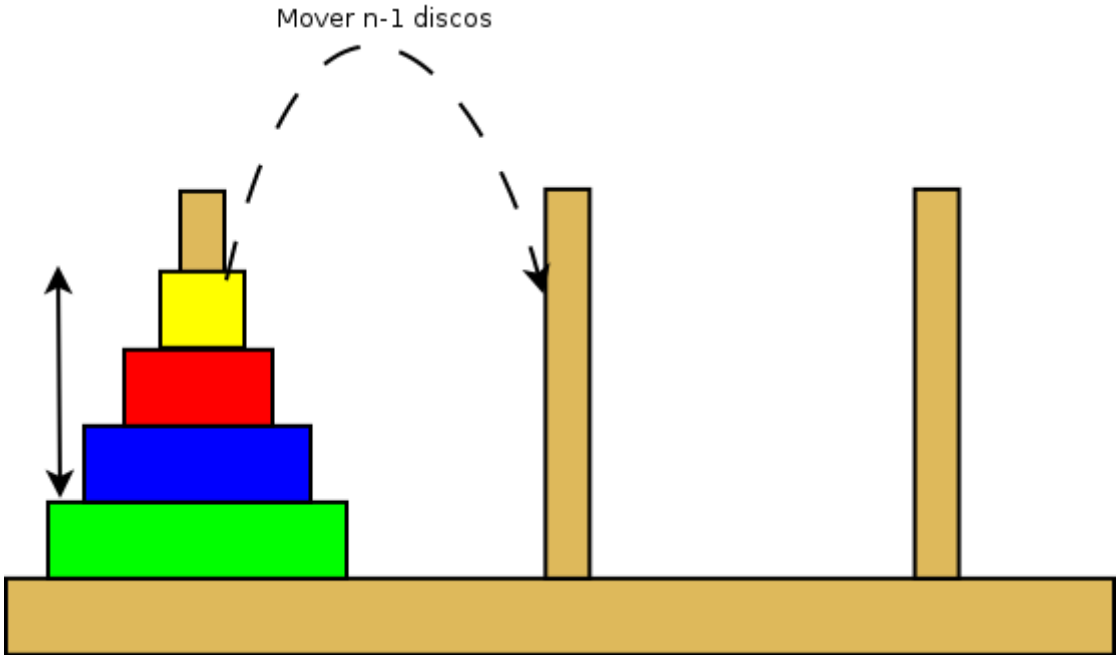
```
calculaListaFactoriales()
```

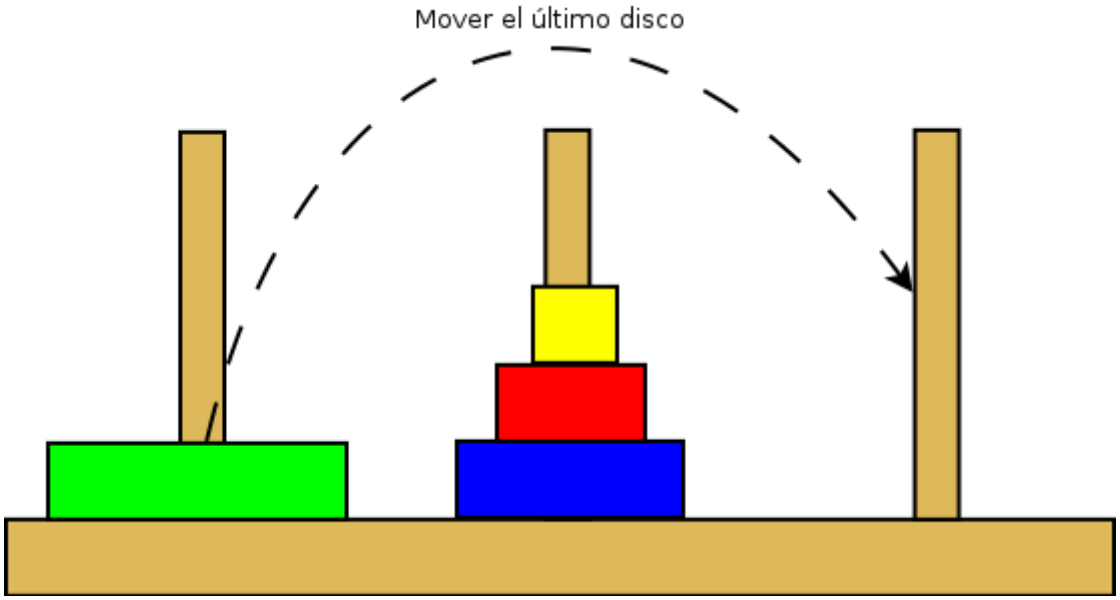
Ejemplo más complicado: Torres de Hanoi

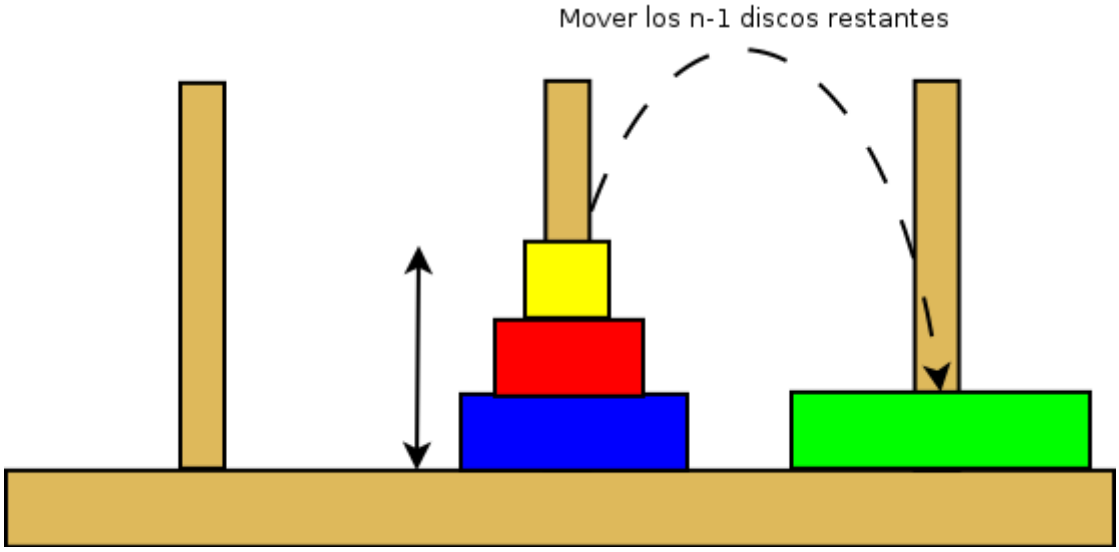
Problema matemático que consiste en mover todos los discos desde una vara a otra, bajo las siguientes restricciones:

- Sólo 1 disco puede ser movido a la vez.
- No puede haber un disco más grande encima de uno más pequeño.
- Un movimiento consiste en mover un disco en la cima de una pila de discos hacia otra pila de discos puestos en otra vara.

Se nos pide crear una función llamada $hanoi()$ que reciba como parámetro el número de discos de la torre y entregue como salida la cantidad de movimientos que debieron realizarse.







- Para mover el disco más grande de una vara a otra, necesitamos mover los $n-1$ discos anteriores a otra vara, lo cual nos toma $hanoi(n-1)$ movimientos.
- Luego, debemos mover el disco más grande de su vara a la desocupada, esto nos toma 1 movimiento.
- A continuación, debemos volver a mover los $n-1$ discos restantes para que queden encima del disco grande que acabamos de mover. Esto nuevamente nos toma $hanoi(n-1)$ movimientos.
- En total, necesitamos $2 \times hanoi(n-1)+1$ movimientos para n discos.

```
# hanoi: int -> int
# calcula el numero de movimientos necesarios para mover
# una torre de n discos desde una vara a otra
# usando 3 varas y siguiendo las restricciones del puzzle de hanoi
# ejemplo: hanoi(0) debe dar 0, hanoi(1) debe dar 1, hanoi(2) debe dar 3
def hanoi(n):
    if n<2:
        return n
    else:
        return 1+2*hanoi(n-1)
# test
assert hanoi(0)==0
assert hanoi(1)==1
assert hanoi(2)==3
assert hanoi(4)==15
assert hanoi(5)==31
```

Hanoi Animado:

Si quiere visualizar cómo se *moverían los discos uno a uno* al seguir el procedimiento anterior (moviendo los discos), mire la animación en http://www.cs.armstrong.edu/liang/animation/web/TowerOfHanoi.html. Pruebe con 2 discos, luego con 3 discos, y luego con más.

Cómo escribir funciones recursivas

- **Son funciones condicionales**
- Pensar de forma inductiva:
  - **caso base**
  - **caso inductivo**: siempre debe reducir el problema y llevar hacia el caso base
  - podemos asumir que el caso inductivo devolverá el valor correcto.
  - debemos pensar qué es lo que le falta al caso inductivo para obtener el caso final.

Receta de diseño

1. Escribir varios ejemplos de uso de la función, incluir parámetro y resultado
   - 2 potencia 4 = 6
   - 2 potencia 0 = 1
   - 2 potencia 3 = 8
2. Decidir cuál de los argumentos va a tener descomposición recursiva.
   - El argumento con el procedimiento recursivo es la potencia

$$a \cdot a^{b-1}$$

3. Entender cuál es el caso base para el argumento.
   - Cuando la potencia es igual a 0, el resultado es 1.
4. Ocupar la receta de diseño normal tomando en cuenta que es una función condicional, donde una rama es el caso base y el otro el caso recursivo.