Clase 6: Testing, Depuración y más Recursión

Testing

Son acciones que permiten probar la funcionalidad de alguna parte de programa.

Depuración: es el proceso de detectar y corregir errores en un programa como errores de sintaxis, de nombre, de tipo, etc., y errores lógicos.

Errores lógicos: son aquellos que causan que nuestro programa no calcule lo deseado.

Testing de funciones

Aspectos generales:

- La función debe cumplir con el contrato estipulado en la receta de diseño
    - Parámetros
    - Tipo de dato de retorno
    - Los tests de la función deben abarcar suficientes casos para verificar que la función está correctamente programada
- Buen testing para funciones condicionales debe incluir tests para:
    - Todos los casos de borde.
    - Al menos un test por cada caso representativo.

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)

Ejemplos:

Problema 1: Suma de progresión aritmética

$$\sum_{i=x}^{y} i = x + (x+1) + \ldots + y$$

Ejemplo:

$$\sum_{i=3}^{5} i = 3 + 4 + 5 = 12$$

```
#suma: int int -> int
#Calcula x + (x+1) + ... + y
#ej: suma(3,5) debe ser 12
def suma(x,y):
    assert (type(x)==int) and (type(y)==int)
    if x>y:
        return 0
    else:
        return suma(x,y-1) + y

assert suma(3,5)==12 #caso general
assert suma(3,3)==3  #caso especial
assert suma(5,3)==0  #caso base
```

Problema 2: Números de Fibonacci

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n > 1 \end{cases}$$

Ejemplo:

$\{0, 1, 1, 2, 3, 5, 8, 13\} \Rightarrow f(7) = 13$

```
# fibonacci: int -> int
# calcula el n-esimo numero de la sucession de Fibonacci
# ejemplo: fibonacci(7) debe dar 13
def fibonacci(n):
    assert type(n)==int and n>=0
    if n<2:
        # caso base
        return n
    else:
        # caso recursivo
        return fibonacci(n-1) + fibonacci(n-2)

# test:
assert fibonacci(0) == 0
assert fibonacci(1) == 1
assert fibonacci(2) == 1
assert fibonacci(7) == 13
```

Problema 3: Repetir un string n veces

```
#repetir: str int -> str
#string x repetido n veces
#ej: repetir("ja",3) debe ser "jajaja“
def repetir(x,n):
    assert type(x)==str and type(n)==int and n>=0
    if n==0:
        return ""
    else:
        return x + repetir(x,n-1)

assert repetir("ja",3)=="jajaja" #caso general
assert repetir("ja",0)==""       #caso base
assert repetir("ja",1)=="ja"     #caso especial
```

```
repetir("hola",10)
```

Problema 4: Cálculo de una distancia (problema de punto flotante)

Leer capítulo 7 en detalle

```
#distancia: num num num num -> float
#distancia entre puntos (x0,y0) y (x1,y1)
#ej: distancia(1,0,4,0) debe ser 3.0
def distancia(x0,y0,x1,y1):
    dx = x1 - x0
    dy = y1 - y0
    return (dx**2 + dy**2)**0.5

#Tests
assert distancia(1,0,4,0)==3.0

#resultado con 4 decimales de precisión
assert abs(distancia(0,1,1,0)-1.4142)< 0.0001
```

Haz doble clic (o ingresa) para editar

```
>>> d1 = distancia(0.1, 0.2, 0.2, 0.1)
>>> d1
```

```
     0.14142135623730953
```

```
>>> d2 = distancia(1,2,2,1)
>>> d2
```

```
    1.4142135623730951
```

```
>>> 10*d1 == d2
```

```
    False
```

Se produce un error de redondeo de punto flotante, para corregirlo debemos definir un parámetro ε (epsilon) de tolerancia o cercanía mínima entre dos valores para poder decir que estos son iguales:

```
# cerca: num num num -> bool
# retorna True si x es igual a y con
# precision epsilon
def cerca(x, y, epsilon):
    diff = x - y
    return abs(diff) < epsilon
```

```
>>> cerca(10*d1,d2,0.0001)
```

```
    True
```

Problema propuesto: Combinatoria

$$\binom{x}{y} = \frac{x!}{y!(x-y)!}$$

$$\binom{x}{y} = \binom{x-1}{y} + \binom{x-1}{y-1}$$

Caso base:

$$\binom{x}{y} = 1$$

si es que $x = y$ ó $y = 0$

Hacer función `comb(x,y)` que calcule $\binom{x}{y}$ incluyendo receta de diseño y tests para varios casos representativos.

Puede hacerlo de dos maneras:

1. **Sin recursión**: usando la función `factorial(x)` con la primera fórmula.
2. **Con recursión**: Sin usar la función factorial, directamente usando la segunda relación $\binom{x}{y} = \binom{x-1}{y} + \binom{x-1}{y-1}$ y el caso base.

De la primera manera es fácil pero no usa recursión. Le recomendamos mucho intentar hacerlo de la segunda manera, usando recursión. :-)