

# Introducción a la Programación

## Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2025

Departamento de Computación - FCEyN - UBA

Práctica 4: Recursión sobre números enteros

# Repaso de la teórica

Vamos a implementar la función factorial, entre todos como repaso de lo visto de recursión en la teórica.

Recordemos:

$$n! = \prod_{k=1}^n k$$

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

# Posible solución para factorial

```
factorial :: Int -> Int
factorial n
  | n == 0 = 1
  | n > 0 = n * factorial (n-1)
```

# Comentando el código

Para poder escribir texto que no sea ejecutado por Haskell, pero que podamos ver y leer en el código, podemos utilizar los comentarios. Se pueden agregar comentarios de una sola línea:

```
factorial :: Int -> Int
--Este es un comentario, no interrumpe la ejecución
factorial n
    | n == 0 = 1
    | n > 0 = n * factorial (n-1)
```

# Comentando el código

Si queremos dejar comentarios de varias líneas, por ejemplo cuando estamos *debuggando* el código y queremos saber dónde está el error, podemos comentar toda una función para que no se ejecute:

```
{--  
factorial :: Int -> Int  
factorial n  
    | n == 0 = 1  
    | n > 0 = n * factorial (n-1)  
--}
```

## Ej 1

Implementar la función `fibonacci`: `Integer -> Integer` que devuelve el *i*-ésimo número de Fibonacci. Recordar que la secuencia de Fibonacci se define como:

$$fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fib(n-1) + fib(n-2) & \text{en otro caso} \end{cases}$$

```
problema fibonacci (n: ℤ) : ℤ {  
  requiere: { n ≥ 0 }  
  asegura: { resultado = fib(n) }  
}
```

# Ej 1

Implementar la función `fibonacci`: `Integer -> Integer` que devuelve el *i*-ésimo número de Fibonacci. Recordar que la secuencia de Fibonacci se define como:

$$fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fib(n-1) + fib(n-2) & \text{en otro caso} \end{cases}$$

Podemos comenzar pensando cual es el caso base (o mejor dicho, los casos base):

- $n = 0 \Rightarrow (resultado = 0)$
- $n = 1 \Rightarrow (resultado = 1)$

## Ej 1

Implementar la función `fibonacci`: `Integer -> Integer` que devuelve el *i*-ésimo número de Fibonacci. Recordar que la secuencia de Fibonacci se define como:

$$fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fib(n-1) + fib(n-2) & \text{en otro caso} \end{cases}$$

Y luego consideramos el paso recursivo:

- ▶  $n = 0 \Rightarrow (resultado = 0)$
- ▶  $n = 1 \Rightarrow (resultado = 1)$
- ▶  $n \geq 2 \Rightarrow (resultado = fib(n-1) + fib(n-2))$



# Posible solución Ej1

## Posible solución

Se mostrará en clase

# Posible solución Ej1

Lo planteamos en Haskell usando guardas:

Posible solución

Se mostrará en clase

# Posible solución Ej1

La podemos definir también usando pattern matching:

## Posible solución

Se mostrará en clase

## Ej 2

Implementar una función `parteEntera :: Float -> Integer` que calcule la parte entera de un número real.

```
problema parteEntera (x: ℝ) : ℤ {  
  requiere: {  $x \geq 0$  }  
  asegura: {  $resultado \leq x < resultado + 1$  }  
}
```

## Ej 2

Probemos con algunos ejemplos:

`parteEntera` 8.124 = ?

`parteEntera` 1.999999 = ?

`parteEntera` 0.12 = ?

## Ej 2

Probemos con algunos ejemplos:

`parteEntera 8.124 = 8`

`parteEntera 1.999999 = 1`

`parteEntera 0.12 = 0`

# Posible Solución Ej 2

## Posible solución

Se mostrará en clase

## Ej 7

Implementar la función `iesimoDigito :: Integer -> Integer -> Integer` que dado un  $n \in \mathbb{Z}$  mayor o igual a 0 y un  $i \in \mathbb{Z}$  mayor o igual a 1 y menor o igual a la cantidad de dígitos de  $n$ , devuelve el  $i$ -ésimo dígito de  $n$ .

```
problema iesimoDigito (n:  $\mathbb{Z}$ , i:  $\mathbb{Z}$ ) :  $\mathbb{Z}$  {  
  requiere: {  $n \geq 0 \wedge 1 \leq i \leq \text{cantDigitos}(n)$  }  
  asegura: {  $\text{resultado} = (n \text{ div } 10^{\text{cantDigitos}(n)-i}) \bmod 10$  }  
}
```

```
problema cantDigitos (n:  $\mathbb{Z}$ ) :  $\mathbb{Z}$  {  
  requiere: {  $n \geq 0$  }  
  asegura: {  $n = 0 \rightarrow \text{resultado} = 1$  }  
  asegura: {  
     $n \neq 0 \rightarrow (n \text{ div } 10^{\text{resultado}-1} > 0 \wedge n \text{ div } 10^{\text{resultado}} = 0)$  }  
}
```



# Demos algunos ejemplos para asegurarnos que comprendimos la especificación

- ▶ `cantDigitos 0 = ?`
- ▶ `cantDigitos 12 = ?`
- ▶ `cantDigitos 123 = ?`

## Demos algunos ejemplos para asegurarnos que comprendimos la especificación

- ▶  $\text{cantDigitos } 0 = 1$
- ▶  $\text{cantDigitos } 12 = (12 \text{ div } 10^{res-1} > 0 \wedge 12 \text{ div } 10^{res} = 0) = 2$
- ▶  $\text{cantDigitos } 123 = (123 \text{ div } 10^{res-1} > 0 \wedge 123 \text{ div } 10^{res} = 0) = 3$

## Y ejemplos con iesimoDigito?

- ▶ `iesimoDigito 468 0 = ?`
- ▶ `iesimoDigito 468 1 = ?`
- ▶ `iesimoDigito 468 2 = ?`
- ▶ `iesimoDigito 468 3 = ?`

# Y ejemplos con iesimoDigito?

## Posible solución

Se mostrará en clase

## Posible solución

Se mostrará en clase

## Ej 9

Especificar e implementar una función `esCapicua :: Integer -> Bool` que dado  $n \in \mathbb{N}_{\geq 0}$  determina si  $n$  es un número capicúa. Se puede considerar que la entrada no tiene ceros.

## Ej 9

Especificar e implementar una función `esCapicua :: Integer -> Bool` que dado  $n \in \mathbb{N}_{\geq 0}$  determina si  $n$  es un número capicúa. Se puede considerar que la entrada no tiene ceros. Algunos ejemplos...

- ▶ `esCapicua 1 = ?`
- ▶ `esCapicua 23 = ?`
- ▶ `esCapicua 22 = ?`
- ▶ `esCapicua 56765 = ?`
- ▶ `esCapicua 5689 = ?`

## Ej 9

Especificar e implementar una función `esCapicua :: Integer -> Bool` que dado  $n \in \mathbb{N}_{\geq 0}$  determina si  $n$  es un número capicúa. Se puede considerar que la entrada no tiene ceros.

Algunos ejemplos...

- ▶ `esCapicua 1 = True`
- ▶ `esCapicua 23 = False`
- ▶ `esCapicua 22 = True`
- ▶ `esCapicua 56765 = True`
- ▶ `esCapicua 5689 = False`



# Posible Solución esCapicua

## Posible solución

Se mostrará en clase