Introducción a la Programación

Algoritmos y Estructuras de Datos I

Segundo 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:

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

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 sóla 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 *debugeando* el código y queremos saber dónde esta 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: $\mathbb{Z}$) : $\mathbb{Z}$ {
  `requiere:` $\{ n \geq 0 \}$
  `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):

- $n = 0 => (resultado = 0)$
- $n = 1 => (resultado = 1)$

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:

- $n = 0 => (resultado = 0)$
- $n = 1 => (resultado = 1)$
- $n >= 2 => (resultado = fib(n-1) + fib(n-2))$

Posible solución Ej1

Lo planteamos en Haskell:

```
fibonacci :: Integer -> Integer
fibonacci n | n == 0 = ...
            | n == 1 = ...
            | n >=2 = ...
```

Otra forma de resolverlo ...

Posible solución Ej1

Lo planteamos en Haskell usando guardas:

```haskell
fibonacci :: Integer -> Integer
fibonacci n | n == 0 || n == 1 = n
            | n >= 2 = (fibonacci (n-1)) +
                       (fibonacci (n-2))
```

Posible solución Ej1

Lo planteamos en Haskell usando guardas:

```
fibonacci :: Integer -> Integer
fibonacci n | n == 0 || n == 1 = n
            | n >= 2 = (fibonacci (n-1)) +
                       (fibonacci (n-2))
```

Esta no es la única forma de implementar la función en Haskell. Veamos otras

Posible solución Ej1

La podemos definir también usando pattern matching:

```haskell
fibonacci :: Integer -> Integer
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = (fibonacci (n-1)) +
              (fibonacci (n-2))
```

Posible solución Ej1

La podemos definir también usando pattern matching:

```
fibonacci :: Integer -> Integer
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = (fibonacci (n-1)) +
              (fibonacci (n-2))
```

- ¿Qué pasa si introducimos n=-1 en nuestra función?
- ¿Debemos preocuparnos por este caso?

Ej 2

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

problema parteEntera (x: $\mathbb{R}$) : $\mathbb{Z}$ {
  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

```haskell
parteEntera :: Float -> Int
parteEntera x | x < 1 = 0
              | otherwise = 1 + parteEntera (x -1)
```

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:` $\{\ n \geq 0 \land 1 \leq i \leq cantDigitos(n)\ \}$
  `asegura:` $\{\ resultado = (n \text{ div } 10^{cantDigitos(n)-i}) \text{ mod } 10\ \}$
}

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

Demos algunos ejemplos para asegurarnos que comprendimos la especificación

- cantDigitos $0 = ?$
- cantDigitos $12 = ?$
- cantDigitos $123 = ?$

Demos algunos ejemplos para asegurarnos que comprendimos la especificación

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

- iesimoDigito 468 0 $= (468$ div $10^{cantDigitos(468)-0})$ mod $10 = (468$ div $10^3)$ mod $10 = (468$ div $1000)$ mod $10 =$ 0
- iesimoDigito 468 1 $= (468$ div $10^{cantDigitos(468)-1})$ mod $10 = (468$ div $10^{3-1})$ mod $10 = (468$ div $10^2)$ mod $10 =$ 4
- iesimoDigito 468 2 $= (468$ div $10^{cantDigitos(468)-2})$ mod $10 = (468$ div $10^{3-2})$ mod $10 = (468$ div $10^1)$ mod $10 =$ 6
- iesimoDigito 468 3 $= (468$ div $10^{cantDigitos(468)-3})$ mod $10 = (468$ div $10^{3-3})$ mod $10 = (468$ div $10^0)$ mod $10 =$ 8

Notemos que el requiere indica que: $n \geq 0 \wedge 1 \leq i \leq cantDigitos(n)$, por lo tanto el primer caso con i = 0 no es válido. Y tampoco sería válido usar i = 4 porque 468 tiene 3 dígitos.

Posible solución iesimoDigito

```
-- usando recursión
iesimoDigito :: Int -> Int -> Int
iesimoDigito x i | i == cantidadDeDigitos x = digitoUnidades x
                 | otherwise = iesimoDigito (eliminarUnidades x) i


cantidadDeDigitos :: Int -> Int
cantidadDeDigitos x | x < 10 = 1
                    | otherwise = 1 + cantidadDeDigitos (eliminarUnidades x)

eliminarUnidades :: Int -> Int
eliminarUnidades x = div x 10

digitoUnidades :: Int -> Int
digitoUnidades x = mod x 10

-- alternativa sin recursión en iesimoDigito
iesimoDigito2 :: Int -> Int -> Int
iesimoDigito2 n i = digitoUnidades (n `div` 10^(cantidadDeDigitos n - i))
```

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

```
esCapicua :: Int -> Bool
esCapicua n = n < 10 || (primero == ultimo && esCapicua (sacarPrimeroYultimo n))
             where primero =  (iesimoDigito n 1)
                   ultimo = mod n 10

-- Resultado es 0 si n es menor a 100
sacarPrimeroYUltimo :: Int -> Int
sacarPrimeroYUltimo n = sacarUnidades (sacarNumeroMasSignificativo n)

sacarNumeroMasSignificativo :: Int -> Int
sacarNumeroMasSignificativo n = mod n (10^(cantidadDeDigitos n - 1))
```