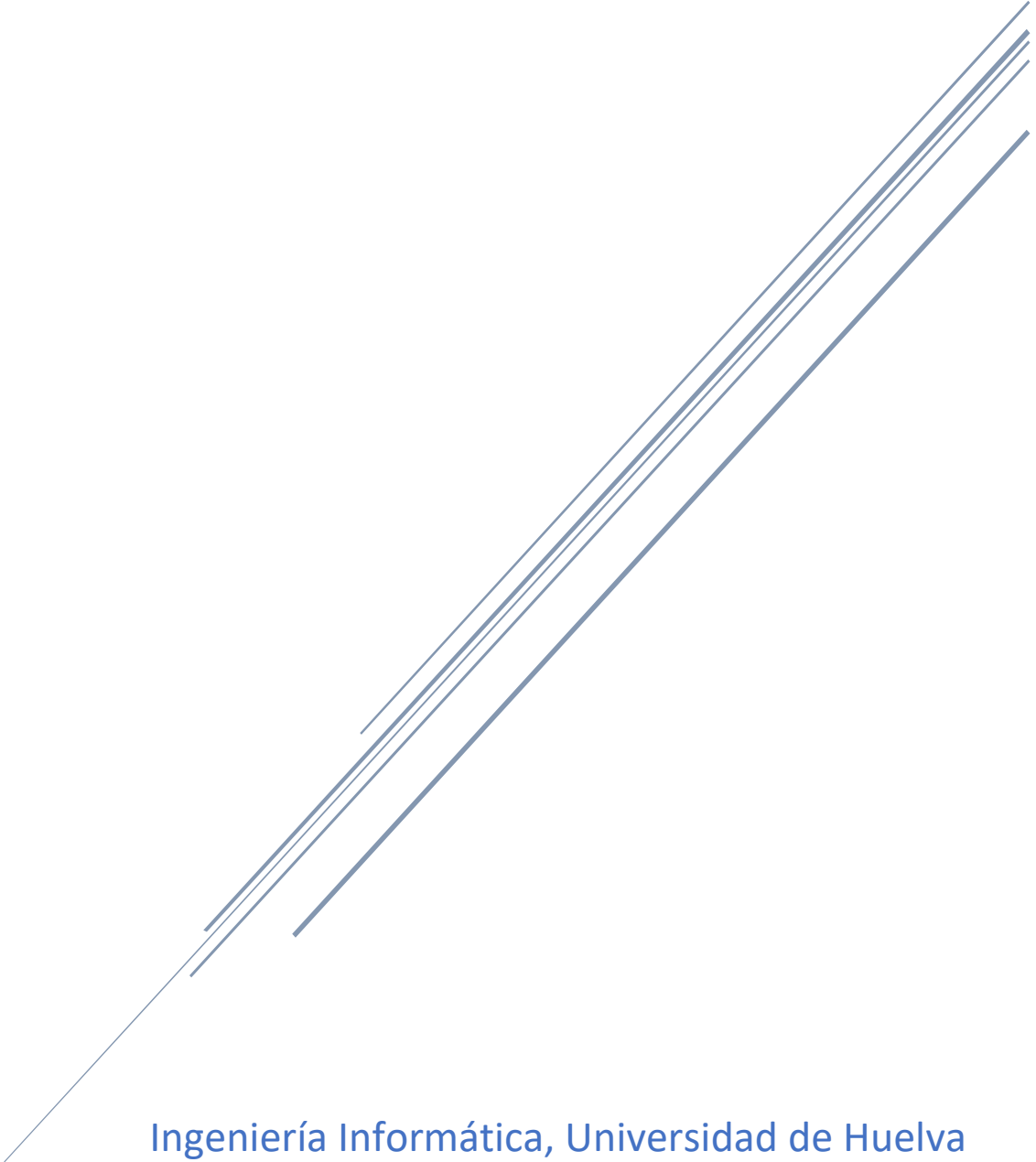


# PRÁCTICA 2: FUNCIONES EN HASKELL

Alba Márquez Rodríguez



Ingeniería Informática, Universidad de Huelva  
Modelos Avanzados de Computación

### Ejercicio 1: Nsobrek

Sea la función nsobrek tal que nsobrek n k es el número de combinaciones de n elementos tomados de k en k; es decir

$$\binom{n}{k} = \frac{n!}{k! (n-k)!}$$

Se pide: definir la función en Haskell en el mayor número de variantes que hemos visto en clase.

Guardas:

```
-- Guardas

factorial_guardas n
  | n == 0 = 1
  | n > 0 = n * factorial_guardas(n-1)
  | otherwise = error "valor negativo"

nsobrek_guardas n k
  | n > k = (factorial_guardas n) / ((factorial_guardas k) * factorial_guardas (n-k))
  | otherwise = error "n debe ser mayor que k"
```

If-then else:

```
-- if then else

factorial_if n = if (n == 0) then 1 else n*factorial_if(n-1)

nsobrek_if n k =
  if (n > k) then (factorial_if n) / ((factorial_if k) * factorial_if (n-k))
  else error "n debe ser mayor que k"
```

Case:

```
-- case

factorial_case n = case (n == 0) of
  True -> 1
  False -> n*factorial_if(n-1)

nsobrek_case n k = case (n > k) of
  True -> (factorial_case n) / ((factorial_case k) * factorial_case (n-k))
  False -> error "n debe ser mayor que k"
```

Primero ya que debemos calcular el factorial de varios números realizamos esta función para llamarla en la función principal.

En la función principal aseguramos que n > k ya que en el caso contrario el factorial de n-k será erróneo. Una vez se haya tenido esto en cuenta se calculan los factoriales y se operan tal y como se muestra en la fórmula de nsobrek.

```
Main> nsobrek 3 2
3.0
Main> nsobrek 4 2
6.0
```

## Actividad 2

Definir la función raíces tal que raíces a b c es la lista de las raíces de la ecuación  $ax^2 + bx + c = 0$ .

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Se pide: definir la función en Haskell en el mayor número de variantes que hemos visto en clase.

Guardas:

```
-- Guardas
raices_guardas::(Floating x, Ord x) => x -> x -> x -> [x]
raices_guardas a b c
  | (a == 0) && (b /= 0) = [(-c)/b]
  | (a == 0) && (b == 0) = error "No hay variable X"
  | (b*b - 4*a*c) < 0 = error "La raíz es negativa"
  | otherwise = unico [(-b + sqrt (b*b - 4*a*c))/(2*a), (-b - sqrt (b*b - 4*a*c))/(2*a)]
  where
    unico [] = []
    unico list = (head list):unico (filter (\x -> x /= (head list)) list)
```

If:

```
-- if
raices_if::(Floating x, Ord x) => x -> x -> x -> [x]
raices_if a b c =
  if (a == 0) && (b /= 0) then [(-c)/b]
  else if (a == 0) && (b == 0) then error "No hay variable X"
  else if (b*b - 4*a*c) < 0 then error "La raíz es negativa"
  else unico [(-b + sqrt (b*b - 4*a*c))/(2*a), (-b - sqrt (b*b - 4*a*c))/(2*a)]
  where
    unico [] = []
    unico (h:hs) = h:unico (filter (\x -> x /= h) (hs))
```

Para calcular la raíz cuadrada primero se verifica el cálculo más básico, q es el caso en el que  $a = 0$ . Luego en el caso en el que pueda salir un valor no válido como que sea del tipo  $0x^2 + 0x + c = c$ . Es decir, ninguna ecuación que resolver.

O bien, en el caso en el que la raíz fuese negativa y entonces no se podría calcular.

Tras verificar estas condiciones se procede al cálculo de la raíz, que se devuelve en una lista que ya la raíz puede tener valor positivo o negativo y hay que hacer dos cálculos.

### Actividad 3: Sucesión de Fibonacci

Los números de Fibonacci quedan definidos por las ecuaciones

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

Esto produce los siguientes números:

- $f_2 = 1$
- $f_3 = 2$
- $f_4 = 3$
- $f_5 = 5$
- $f_6 = 8$
- $f_7 = 13$
- $f_8 = 21$

y así sucesivamente.

Se pide: definir la función en Haskell en el mayor número de variantes que hemos visto en clase.

Local:

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

Guardas:

```
-- Guardas
fib_guardas :: Integer -> Integer
fib_guardas n
  | n == 0 = 0
  | n == 1 = 1
  | n > 1 = fib_guardas (n-1) + fib_guardas (n-2)
  | otherwise = error "valor negativo"
```

If-then else:

```
-- if then else
fib_if :: Integer -> Integer
fib_if n =
  if (n == 0) then 0
  else if (n == 1) then 1
  else if (n < 0) then error "n debe positivo"
  else fib_if (n-1) + fib_if (n-2)
```

Se realiza recursividad, esta terminará cuando  $n$  sea igual a 0 o 1. Así que estas son las primeras condiciones, en el caso en el que  $n$  no sea ni 0 o 1 habrá que seguir realizando esta recursividad sobre  $n-1$  y  $n-2$ .

## Actividad 4: Comprobar la pertenencia a una lista usando una función recursive

### Pertenece a [b]

Ejemplos:

Pertenece 3 [2,3,5] == True

Pertenece 4 [2,3,5] == False

Se pide: definir la función en haskell en el mayor número de variantes que hemos visto en clase.

Guardas:

```
-- Guardas
pertenece_guardas n list
| list == [] = False
| (head list) == n = True -- si el primer elemento de la lista es n, true, else rec
| otherwise = pertenece_guardas n (tail list)
```

If-else-then:

```
-- if then else
pertenece_if n list =
  if list == [] then False
  else if (head list) == n then True -- si el primer elemento de la lista es n, true
  else pertenece_if n (tail list)
```

Se crea una recursividad que se terminará en el caso en el que el elemento n se encuentre en la cabeza de la lista o la lista quede vacía.

Si no se cumple ninguna de estas condiciones hará recursividad sobre la cola de la lista (todos los elementos menos el primero, que es con el que se habrá comparado para saber si terminar la recursividad o continuarla).