Exercitium (curso 2018–19)

Ejercicios de programación funcional con Haskell

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 1 de diciembre de 2018

> “The chief goal of my work as an educator and author is to help people learn to write *beautiful programs*.”
>
> Donald Knuth en Computer programming as an art

Índice general

Introducción

Este libro es una recopilación de las soluciones de los ejercicios propuestos en el blog Exercitium [1] durante el curso 2018–19.

El principal objetivo de Exercitium es servir de complemento a la asignatura de Informática [2] de 1º del Grado en Matemáticas de la Universidad de Sevilla.

Con los problemas de Exercitium, a diferencias de los de las relaciones [3], se pretende practicar con los conocimientos adquiridos durante todo el curso, mientras que con las relaciones están orientadas a los nuevos conocimientos.

Habitualmente de cada ejercicio se muestra distintas soluciones y se compara sus eficiencias.

La dinámica del blog es la siguiente: cada día, de lunes a viernes, se propone un ejercicio para que los alumnos escriban distintas soluciones en los comentarios. Pasado 7 días de la propuesta de cada ejercicio, se cierra los comentarios y se publica una selección de sus soluciones.

Para conocer la cronología de los temas explicados se puede consultar el diario de clase [4].

En el libro se irán añadiendo semanalmente las soluciones de los ejercicios del curso.

El código del libro se encuentra en GitHub [5]

[1] https://www.glc.us.es/~jalonso/exercitium
[2] https://www.cs.us.es/~jalonso/cursos/i1m-18
[3] https://www.cs.us.es/~jalonso/cursos/i1m-18/ejercicios/ejercicios-I1M-2018.pdf
[4] https://www.glc.us.es/~jalonso/vestigium/category/curso/i1m/i1m2018
[5] https://github.com/jaalonso/Exercitium2018

Ejercicio 1

Listas equidigitales

Ejercicio propuesto el 9–11–18

Una lista de números naturales es equidigital si todos sus elementos tienen el mismo número de dígitos.

Definir la función

```
equidigital :: [Int] -> Bool
```

tal que `(equidigital xs)` se verifica si `xs` es una lista equidigital. Por ejemplo,

```
equidigital [343,225,777,943]   ==  True
equidigital [343,225,777,94,3]  ==  False
```

Soluciones

```haskell
-- 1ª definición
-- =============

equidigital :: [Int] -> Bool
equidigital xs = todosIguales (numerosDeDigitos xs)

-- (numerosDeDigitos xs) es la lista de los números de dígitos de
```

```haskell
-- los elementos de xs. Por ejemplo,
--    numerosDeDigitos [343,225,777,943]   ==  [3,3,3,3]
--    numerosDeDigitos [343,225,777,94,3]  ==  [3,3,3,2,1]
numerosDeDigitos :: [Int] -> [Int]
numerosDeDigitos xs = [numeroDeDigitos x | x <- xs]

-- (numeroDeDigitos x) es el número de dígitos de x. Por ejemplo,
--    numeroDeDigitos 475  ==  3
numeroDeDigitos :: Int -> Int
numeroDeDigitos x = length (show x)

-- (todosIguales xs) se verifica si todos los elementos de xs son
-- iguales. Por ejemplo,
--    todosIguales [3,3,3,3]    ==  True
--    todosIguales [3,3,3,2,1]  ==  False
todosIguales :: Eq a => [a] -> Bool
todosIguales (x:y:zs) = x == y && todosIguales (y:zs)
todosIguales _        = True

-- 2ª definición
-- =============

equidigital2 :: [Int] -> Bool
equidigital2 []     = True
equidigital2 (x:xs) = and [numeroDeDigitos y == n | y <- xs]
    where n = numeroDeDigitos x

-- 3ª definición
-- =============

equidigital3 :: [Int] -> Bool
equidigital3 (x:y:zs) = numeroDeDigitos x == numeroDeDigitos y &&
                        equidigital3 (y:zs)
equidigital3 _        = True
```

Ejercicio 2

Distancia de Hamming

Ejercicio propuesto el 10–11-18

La distancia de Hamming entre dos listas es el número de posiciones en que los correspondientes elementos son distintos. Por ejemplo, la distancia de Hamming entre romaz ”loba.es 2 (porque hay 2 posiciones en las que los elementos correspondientes son distintos: la 1ª y la 3ª).

Definir la función

```
distancia :: Eq a => [a] -> [a] -> Int
```

tal que (distancia xs ys) es la distancia de Hamming entre xs e ys. Por ejemplo,

```
distancia ”romano” ”comino”  ==  2
distancia ”romano” ”camino”  ==  3
distancia ”roma”   ”comino”  ==  2
distancia ”roma”   ”camino”  ==  3
distancia ”romano” ”ron”     ==  1
distancia ”romano” ”cama”    ==  2
distancia ”romano” ”rama”    ==  1
```

Comprobar con QuickCheck si la distancia de Hamming tiene la siguiente propiedad: distancia(xs,ys) = 0 si, y sólo si, xs = ys y, en el caso de que no se verifique, modificar ligeramente la propiedad para obtener una condición necesaria y suficiente de distancia(xs,ys) = 0.

Soluciones

```haskell
import Test.QuickCheck

-- 1ª definición:
distancia :: Eq a => [a] -> [a] -> Int
distancia xs ys = length [(x,y) | (x,y) <- zip xs ys, x /= y]

-- 2ª definición:
distancia2 :: Eq a => [a] -> [a] -> Int
distancia2 [] _ = 0
distancia2 _ [] = 0
distancia2 (x:xs) (y:ys) | x /= y    = 1 + distancia2 xs ys
                         | otherwise = distancia2 xs ys

-- La propiedad es
prop_distancia1 :: [Int] -> [Int] -> Bool
prop_distancia1 xs ys =
  (distancia xs ys == 0) == (xs == ys)

-- La comprobación es
--    ghci> quickCheck prop_distancia1
--    *** Failed! Falsifiable (after 2 tests and 1 shrink):
--    []
--    [1]
--
-- En efecto,
--    ghci> distancia [] [1] == 0
--    True
--    ghci> [] == [1]
--    False
--
-- La primera modificación es restringir la propiedad a lista de igual
-- longitud:
prop_distancia2 :: [Int] -> [Int] -> Property
prop_distancia2 xs ys =
  length xs == length ys ==>
  (distancia xs ys == 0) == (xs == ys)

-- La comprobación es
```

```haskell
--    ghci> quickCheck prop_distancia2
--    *** Gave up! Passed only 33 tests.

-- Nota. La propiedad se verifica, pero al ser la condición demasiado
-- restringida sólo 33 de los casos la cumple.

-- La segunda restricción es limitar las listas a la longitud de la más
-- corta:
prop_distancia3 :: [Int] -> [Int] -> Bool
prop_distancia3 xs ys =
  (distancia xs ys == 0) == (take n xs == take n ys)
  where n = min (length xs) (length ys)

-- La comprobación es
--    ghci> quickCheck prop_distancia3
--    +++ OK, passed 100 tests.
```

Ejercicio 3

Último dígito no nulo del factorial

Ejercicio propuesto el 13–11–18

El factorial de 7 es $7! = 1 * 2 * 3 * 4 * 5 * 6 * 7 = 5040$. Por tanto, el último dígito no nulo del factorial de 7 es 4.

Definir la función

```
ultimoNoNuloFactorial :: Integer -> Integer
```

tal que (ultimoNoNuloFactorial n) es el último dígito no nulo del factorial de n. Por ejemplo,

```
ultimoNoNuloFactorial  7  == 4
ultimoNoNuloFactorial 10  == 8
ultimoNoNuloFactorial 12  == 6
ultimoNoNuloFactorial 97  == 2
ultimoNoNuloFactorial  0  == 1
```

Comprobar con QuickCheck que si n es mayor que 4, entonces el último dígito no nulo del factorial de n es par.

Solución

```haskell
import Test.QuickCheck

-- 1ª definición
-- =============

ultimoNoNuloFactorial :: Integer -> Integer
ultimoNoNuloFactorial n = ultimoNoNulo (factorial n)

-- (ultimoNoNulo n) es el último dígito no nulo de n. Por ejemplo,
--    ultimoNoNulo 5040  ==  4
ultimoNoNulo :: Integer -> Integer
ultimoNoNulo n
  | m /= 0    = m
  | otherwise = ultimoNoNulo (n `div` 10)
  where m = n `rem` 10

-- (factorial n) es el factorial de n. Por ejemplo,
--    factorial 7  ==  5040
factorial :: Integer -> Integer
factorial n = product [1..n]

-- 2ª definición
-- =============

ultimoNoNuloFactorial2 :: Integer -> Integer
ultimoNoNuloFactorial2 n = ultimoNoNulo2 (factorial n)

-- (ultimoNoNulo2 n) es el último dígito no nulo de n. Por ejemplo,
--    ultimoNoNulo 5040  ==  4
ultimoNoNulo2 :: Integer -> Integer
ultimoNoNulo2 n = read [head (dropWhile (=='0') (reverse (show n)))]

-- Comprobación
-- ============

-- La propiedad es
prop_ultimoNoNuloFactorial :: Integer -> Property
prop_ultimoNoNuloFactorial n =
```

```
  n > 4 ==> even (ultimoNoNuloFactorial n)

-- La comprobación es
--    ghci> quickCheck prop_ultimoNoNuloFactorial
--    +++ OK, passed 100 tests.
```

Ejercicio 4

Diferencia simétrica

Ejercicio propuesto el 14–11–18

La diferencia simétrica de dos conjuntos es el conjunto cuyos elementos son aquellos que pertenecen a alguno de los conjuntos iniciales, sin pertenecer a ambos a la vez. Por ejemplo, la diferencia simétrica de 2,5,3 y 4,2,3,7 es 5,4,7.

Definir la función

```
diferenciaSimetrica :: Ord a => [a] -> [a] -> [a]
```

tal que (diferenciaSimetrica xs ys) es la diferencia simétrica de xs e ys. Por ejemplo,

```
diferenciaSimetrica [2,5,3] [4,2,3,7]    ==  [4,5,7]
diferenciaSimetrica [2,5,3] [5,2,3]      ==  []
diferenciaSimetrica [2,5,2] [4,2,3,7]    ==  [3,4,5,7]
diferenciaSimetrica [2,5,2] [4,2,4,7]    ==  [4,5,7]
diferenciaSimetrica [2,5,2,4] [4,2,4,7]  ==  [5,7]
```

Soluciones

```
import Data.List

-- 1ª definición
```

```haskell
diferenciaSimetrica :: Ord a => [a] -> [a] -> [a]
diferenciaSimetrica xs ys =
  sort (nub ([x | x <- xs, x `notElem` ys] ++ [y | y <- ys, y `notElem` xs]))

-- 2ª definición
diferenciaSimetrica2 :: Ord a => [a] -> [a] -> [a]
diferenciaSimetrica2 xs ys =
  sort (nub (union xs ys \\ intersect xs ys))

-- 3ª definición
diferenciaSimetrica3 :: Ord a => [a] -> [a] -> [a]
diferenciaSimetrica3 xs ys =
  [x | x <- sort (nub (xs ++ ys))
     , x `notElem` xs || x `notElem` ys]
```

Ejercicio 5

Números libres de cuadrados

Ejercicio propuesto el 15–11–18

Un número entero positivo es libre de cuadrados si no es divisible por el cuadrado de ningún entero mayor que 1. Por ejemplo, 70 es libre de cuadrado porque sólo es divisible por 1, 2, 5, 7 y 70; en cambio, 40 no es libre de cuadrados porque es divisible por 2^2.

Definir la función

```
libreDeCuadrados :: Integer -> Bool
```

tal que (libreDeCuadrados x) se verifica si x es libre de cuadrados. Por ejemplo,

```
libreDeCuadrados 70                  ==  True
libreDeCuadrados 40                  ==  False
libreDeCuadrados 510510              ==  True
libreDeCuadrados (((10^10)^10)^10)   ==  False
```

Soluciones

```
import Data.List (nub)

-- 1ª definición
```

```haskell
-- =============

libreDeCuadrados :: Integer -> Bool
libreDeCuadrados x = x == product (divisoresPrimos x)

-- (divisoresPrimos x) es la lista de los divisores primos de x. Por
-- ejemplo,
--    divisoresPrimos 40  ==  [2,5]
--    divisoresPrimos 70  ==  [2,5,7]
divisoresPrimos :: Integer -> [Integer]
divisoresPrimos x = [n | n <- divisores x, primo n]

-- (divisores n) es la lista de los divisores del número n. Por ejemplo,
--    divisores 30  ==  [1,2,3,5,6,10,15,30]
divisores :: Integer -> [Integer]
divisores n = [x | x <- [1..n], n `mod` x == 0]

-- (primo n) se verifica si n es primo. Por ejemplo,
--    primo 30  == False
--    primo 31  == True
primo :: Integer -> Bool
primo n = divisores n == [1, n]

-- 2ª definición
-- =============

libreDeCuadrados2 :: Integer -> Bool
libreDeCuadrados2 n =
  null [x | x <- [2..n], rem n (x^2) == 0]

-- 3ª definición
-- =============

libreDeCuadrados3 :: Integer -> Bool
libreDeCuadrados3 n =
  null [x | x <- [2..floor (sqrt (fromIntegral n))]
          , rem n (x^2) == 0]

-- 4ª definición
-- =============
```

```haskell
libreDeCuadrados4 :: Integer -> Bool
libreDeCuadrados4 x =
  factorizacion x == nub (factorizacion x)

-- (factorizacion n) es la lista de factores primos de n. Por ejemplo,
--    factorizacion 180  ==  [2,2,3,3,5]
factorizacion :: Integer -> [Integer]
factorizacion n | n == 1    = []
                | otherwise = x : factorizacion (div n x)
  where x = menorFactor n

-- (menorFactor n) es el menor divisor de n. Por ejemplo,
--    menorFactor 15  ==  3
menorFactor :: Integer -> Integer
menorFactor n = head [x | x <- [2..], rem n x == 0]

-- Comparación de eficiencia
-- =========================

--    λ> libreDeCuadrados 510510
--    True
--    (0.76 secs, 89,522,360 bytes)
--    λ> libreDeCuadrados2 510510
--    True
--    (1.78 secs, 371,826,320 bytes)
--    λ> libreDeCuadrados3 510510
--    True
--    (0.01 secs, 0 bytes)
--    λ> libreDeCuadrados4 510510
--    True
--    (0.00 secs, 153,216 bytes)
```

Ejercicio 6

Capicúas productos de dos números de dos dígitos

Ejercicio propuesto el 16–11–18

El número 9009 es capicúa y es producto de dos números de dos dígitos, pues 9009 = 91x99.

Definir la lista

```
capicuasP2N2D :: [Int]
```

cuyos elementos son los números capicúas que son producto de 2 números de dos dígitos. Por ejemplo,

```
take 5  capicuasP2N2D  ==  [121,242,252,272,323]
length  capicuasP2N2D  ==  74
drop 70 capicuasP2N2D  ==  [8008,8118,8448,9009]
```

Soluciones

```
import Data.List (nub, sort)

capicuasP2N2D :: [Int]
capicuasP2N2D = [x | x <- productos, esCapicua x]
```

```haskell
-- productos es la lista de números que son productos de 2 números de
-- dos dígitos.
productos :: [Int]
productos = sort (nub [x*y | x <- [10..99], y <- [x..99]])

-- (esCapicua x) se verifica si x es capicúa.
esCapicua :: Int -> Bool
esCapicua x = xs == reverse xs
  where xs = show x
```

Ejercicio 7

Números autodescriptivos

Ejercicio propuesto el 19–11–18

Un número n es autodescriptivo cuando para cada posición k de n (empezando a contar las posiciones a partir de 0), el dígito en la posición k es igual al número de veces que ocurre k en n. Por ejemplo, 1210 es autodescriptivo porque tiene 1 dígito igual a ”0”, 2 dígitos iguales a ”1”, 1 dígito igual a ”2ƨ ningún dígito igual a ”3”.

Definir la función

```
autodescriptivo :: Integer -> Bool
```

tal que (autodescriptivo n) se verifica si n es autodescriptivo. Por ejemplo,

```
λ> autodescriptivo 1210
True
λ> [x | x <- [1..100000], autodescriptivo x]
[1210,2020,21200]
λ> autodescriptivo 9210000001000
True
```

Nota: Se puede usar la función genericLength.

Soluciones

```haskell
import Data.List (genericLength)

autodescriptivo :: Integer -> Bool
autodescriptivo n = autodescriptiva (digitos n)

digitos :: Integer -> [Integer]
digitos n = [read [d] | d <- show n]

autodescriptiva :: [Integer] -> Bool
autodescriptiva ns =
  and [x == ocurrencias k ns | (k,x) <- zip [0..] ns]

ocurrencias :: Integer -> [Integer] -> Integer
ocurrencias x ys = genericLength (filter (==x) ys)
```

Ejercicio 8

Número de parejas

Ejercicio propuesto el 20–11–18

Definir la función

```
nParejas :: Ord a => [a] -> Int
```

tal que (nParejas xs) es el número de parejas de elementos iguales en xs. Por ejemplo,

```
nParejas [1,2,2,1,1,3,5,1,2]        ==  3
nParejas [1,2,1,2,1,3,2]            ==  2
nParejas [1..2*10^6]                ==  0
nParejas2 ([1..10^6] ++ [1..10^6])  ==  1000000
```

En el primer ejemplos las parejas son (1,1), (1,1) y (2,2). En el segundo ejemplo, las parejas son (1,1) y (2,2).

Comprobar con QuickCheck que para toda lista de enteros xs, el número de parejas de xs es igual que el número de parejas de la inversa de xs.

Soluciones

```
import Test.QuickCheck
import Data.List ((\\), group, sort)
```

```haskell
-- 1ª solución
nParejas :: Ord a => [a] -> Int
nParejas []     = 0
nParejas (x:xs) | x `elem` xs = 1 + nParejas (xs \\ [x])
                | otherwise   = nParejas xs

-- 2ª solución
nParejas2 :: Ord a => [a] -> Int
nParejas2 xs =
  sum [length ys `div` 2 | ys <- group (sort xs)]

-- 3ª solución
nParejas3 :: Ord a => [a] -> Int
nParejas3 = sum . map (`div` 2). map length . group . sort

-- 4ª solución
nParejas4 :: Ord a => [a] -> Int
nParejas4 = sum . map ((`div` 2) . length) . group . sort

-- Equivalencia
prop_equiv :: [Int] -> Bool
prop_equiv xs =
  nParejas xs == nParejas2 xs &&
  nParejas xs == nParejas3 xs &&
  nParejas xs == nParejas4 xs

-- Comprobación:
--    λ> quickCheck prop_equiv
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia
--    λ> nParejas [1..20000]
--    0
--    (2.54 secs, 4,442,808 bytes)
--    λ> nParejas2 [1..20000]
--    0
--    (0.03 secs, 12,942,232 bytes)
--    λ> nParejas3 [1..20000]
--    0
```

```
--    (0.02 secs, 13,099,904 bytes)
--    λ> nParejas4 [1..20000]
--    0
--    (0.01 secs, 11,951,992 bytes)

-- Propiedad:
prop_nParejas :: [Int] -> Bool
prop_nParejas xs =
  nParejas xs == nParejas (reverse xs)

-- Compropación
comprueba :: IO ()
comprueba = quickCheck prop_nParejas

-- Comprobación:
--    λ> comprueba
--    +++ OK, passed 100 tests.
```

Ejercicio 9

Reconocimiento de particiones

Ejercicio propuesto el 21–11–18

Una partición de un conjunto es una división del mismo en subconjuntos disjuntos no vacíos.

Definir la función

```
esParticion :: Eq a => [[a]] -> Bool
```

tal que (esParticion xss) se verifica si xss es una partición; es decir sus elementos son listas no vacías disjuntas. Por ejemplo.

```
esParticion [[1,3],[2],[9,5,7]]  ==  True
esParticion [[1,3],[2],[9,5,1]]  ==  False
esParticion [[1,3],[],[9,5,7]]   ==  False
esParticion [[2,3,2],[4]]        ==  True
```

Soluciones

```
import Data.List ((\\), intersect)

-- 1ª definición
-- =============
```

```haskell
esParticion :: Eq a => [[a]] -> Bool
esParticion xss =
  [] `notElem` xss &&
  and [disjuntos xs ys | xs <- xss, ys <- xss \\ [xs]]

disjuntos :: Eq a => [a] -> [a] -> Bool
disjuntos xs ys = null (xs `intersect` ys)

-- 2ª definición
-- =============

esParticion2 :: Eq a => [[a]] -> Bool
esParticion2 []       = True
esParticion2 (xs:xss) =
  not (null xs) &&
  and [disjuntos xs ys | ys <- xss] &&
  esParticion2 xss

-- 3ª definición
-- =============

esParticion3 :: Eq a => [[a]] -> Bool
esParticion3 []       = True
esParticion3 (xs:xss) =
  not (null xs) &&
  all (disjuntos xs) xss &&
  esParticion3 xss

-- Equivalencia
prop_equiv :: [[Int]] -> Bool
prop_equiv xss =
  and [esParticion xss == f xss | f <- [ esParticion2
                                       , esParticion3]]

-- Comprobación
--    λ> quickCheck prop_equiv
--    +++ OK, passed 100 tests.

-- Comparación de eficiencia:
--    λ> esParticion [[x] | x <- [1..3000]]
```

```
--    True
--    (4.37 secs, 3,527,956,400 bytes)
--    λ> esParticion2 [[x] | x <- [1..3000]]
--    True
--    (1.26 secs, 1,045,792,552 bytes)
--    λ> esParticion3 [[x] | x <- [1..3000]]
--    True
--    (1.30 secs, 1,045,795,272 bytes)
--    λ> esParticion3 [[x] | x <- [1..3000]]
--    True
--    (1.30 secs, 1,045,795,272 bytes)
```

Ejercicio 10

Relación definida por una partición

Ejercicio propuesto el 22–11–18

Dos elementos están relacionados por una partición xss si pertenecen al mismo elemento de xss.

Definir la función

```
relacionados :: Eq a => [[a]] -> a -> a -> Bool
```

tal que (relacionados xss y z) se verifica si los elementos y y z están relacionados por la partición xss. Por ejemplo,

```
relacionados [[1,3],[2],[9,5,7]] 7 9  ==  True
relacionados [[1,3],[2],[9,5,7]] 3 9  ==  False
relacionados [[1,3],[2],[9,5,7]] 4 9  ==  False
```

Soluciones

```
-- 1ª definición
-- =============

relacionados :: Eq a => [[a]] -> a -> a -> Bool
```

```haskell
relacionados [] _ _ = False
relacionados (xs:xss) y z
  | y `elem` xs = z `elem` xs
  | otherwise   = relacionados xss y z

-- 2ª definición
-- =============

relacionados2 :: Eq a => [[a]] -> a -> a -> Bool
relacionados2 xss y z =
  or [elem y xs && elem z xs | xs <- xss]

-- 3ª definición
-- =============

relacionados3 :: Eq a => [[a]] -> a -> a -> Bool
relacionados3 xss y z =
  or [[y,z] `subconjunto` xs | xs <- xss]

-- (subconjunto xs ys) se verifica si xs es un subconjunto de ys; es
-- decir, si todos los elementos de xs pertenecen a ys. Por ejemplo,
--    subconjunto [3,2,3] [2,5,3,5]  ==  True
--    subconjunto [3,2,3] [2,5,6,5]  ==  False
subconjunto :: Eq a => [a] -> [a] -> Bool
subconjunto xs ys = and [elem x ys | x <- xs]

-- 4ª definición
-- =============

relacionados4 :: Eq a => [[a]] -> a -> a -> Bool
relacionados4 xss y z =
  any ([y,z] `subconjunto`) xss
```

Ejercicio 11

Ceros finales del factorial

Ejercicio propuesto el 23–11–18

Definir la función

```
cerosDelFactorial :: Integer -> Integer
```

tal que (cerosDelFactorial n) es el número de ceros en que termina el factorial de n. Por ejemplo,

```
cerosDelFactorial 24                          ==  4
cerosDelFactorial 25                          ==  6
length (show (cerosDelFactorial (1234^5678)))  ==  17552
```

Soluciones

```
import Data.List (genericLength)

-- 1ª definición
-- =============

cerosDelFactorial :: Integer -> Integer
cerosDelFactorial n = ceros (factorial n)

-- (factorial n) es el factorial n. Por ejemplo,
```

```haskell
--    factorial 3  ==  6
factorial :: Integer -> Integer
factorial n = product [1..n]

-- (ceros n) es el número de ceros en los que termina el número n. Por
-- ejemplo,
--    ceros 320000  ==  4
ceros :: Integer -> Integer
ceros n | rem n 10 /= 0 = 0
        | otherwise     = 1 + ceros (div n 10)

-- 2ª definición
-- =============

cerosDelFactorial2 :: Integer -> Integer
cerosDelFactorial2 n = ceros2 (factorial n)

-- (ceros n) es el número de ceros en los que termina el número n. Por
-- ejemplo,
--    ceros 320000  ==  4
ceros2 :: Integer -> Integer
ceros2 n = genericLength (takeWhile (=='0') (reverse (show n)))

-- 3ª definición
-- =============

cerosDelFactorial3 :: Integer -> Integer
cerosDelFactorial3 n
  | n < 5     = 0
  | otherwise = m + cerosDelFactorial3 m
  where m = n `div` 5

-- Comparación de la eficiencia
--    λ> cerosDelFactorial1 (3*10^4)
--    7498
--    (3.96 secs, 1,252,876,376 bytes)
--    λ> cerosDelFactorial2 (3*10^4)
--    7498
--    (3.07 secs, 887,706,864 bytes)
--    λ> cerosDelFactorial3 (3*10^4)
```

```
--    7498
--    (0.03 secs, 9,198,896 bytes)
```