# Ejercicios extra para la práctica 2

**Ejercicio** [`empareja`] Define una función recursiva `empareja :: [a] -> [b] -> [(a,b)]` que dadas dos listas empareje sus elementos uno a uno. Por ejemplo:

```
> empareja [1,2,3] "abc"
[(1,'a'),(2,'b'),(3,'c')]

> empareja [1,2,3,4] "abc"
[(1,'a'),(2,'b'),(3,'c')]

> empareja [1,2,3] "abcd"
[(1,'a'),(2,'b'),(3,'c')]

> empareja "abcd" [1..]
[('a',1),('b',2),('c',3),('d',4)]
```

La función `empareja` está predefinida como `zip`. Define una propiedad QuickCheck para comprobar tu definición de `empareja` es equivalente a `zip`.

**Ejercicio** [`emparejaCon`] Define una función recursiva `emparejaCon :: (a -> b -> c) -> [a] -> [b] -> [c]` que dadas una función y dos listas empareje sus elementos uno a uno y aplique a cada pareja la función. Por ejemplo:

```
> emparejaCon (+) [1,2,3] [4,5,6]
[5,7,9]

> emparejaCon max [7, 11, 1] [5, 15, 0, 6]
[7,15,1]

> emparejaCon (\ x y -> chr (x + ord y)) [1..] "hola"
"iqoe"
```

La función `emparejaCon` está predefinida como `zipWith`. Define una propiedad QuickCheck para comprobar tu definición de `emparejaCon` es equivalente a `zipWith`.

**Ejercicio** [`separa`] Define una función recursiva `separaRec :: (a -> Bool) -> [a] -> ([a], [a])` que dados un predicado y una lista, separe la lista en dos listas; la primera contiene los valores que satisfacen el predicado y la segunda los que no. En ambas listas los elementos deben aparecer en el mismo orden que aparecían en la lista original. Por ejemplo:

```
> separaRec even [1..10]
([2,4,6,8,10],[1,3,5,7,9])

> separaRec (`elem` "aeiouAEIOU") "haskell"
("ae","hskll")

> separaRec (<5) [8.75, 6, 10, 4.5, 2, 1.25, 7.5, 3.5]
([4.5,2.0,1.25,3.5],[8.75,6.0,10.0,7.5])
```

Vuelve a definir la función mediante comprensión de listas (`separaC`) y un plegado a la derecha (`separaP`) y utiliza QuickCheck para comprobar que las tres versiones son equivalentes.

**Ejercicio** [`lista de pares`] En Haskell podemos representar una aplicación (`Map<K,V>` en Java) como una lista de

pares `[(k, v)]` donde el primer dato de cada par es la clave y el segundo el valor que corresponde a tal clave. Para simplificar, supondremos que en una lista no existen dos pares que tengan la misma clave.

Por ejemplo, la siguiente es una lista que representa la cotización de algunos valores en bolsa:

```
cotizacion :: [(String, Double)]
cotizacion = [("apple", 116), ("intel", 35), ("google", 824), ("nvidia", 67)]
```

Define una función recursiva `buscarRec :: Eq a => a -> [(a,b)] -> [b]` que dados una clave y una lista de pares (clave, valor) devuelva una lista con el único valor que tiene asociado esa clave. Si la clave no aparece en la lista, el resultado será la lista vacía. Por ejemplo:

```
> buscarRec "google" cotizacion
[824.0]

> buscarRec "ibm" cotizacion
[]
```

Vuelve a definir la función mediante comprensión de listas (`buscarC`) y un plegado a la derecha (`buscarP`).

Utiliza QuickCheck para comprobar que las tres versiones son equivalentes. Es probable que la propiedad falle y que las tres versiones no sean equivalentes. ¿Por qué falla la propiedad? Realiza los cambios necesarios para que las 3 funciones sean equivalentes.

Usando composición `(.)`, `sum`, `map` y `filter` define una función `valorCartera :: [(String, Double)] -> [(String, Double)] -> Double` que dada una cartera de valores (representada como una lista de pares con el nombre de los títulos y su cantidad) calcule su valor para una cotización dada (representada como una lista de pares con el nombre de títulos y su valor). Por ejemplo:

```
> valorCartera [("intel", 1), ("nvidia", 1)] cotizacion
102.0

> valorCartera [("intel", 3), ("nvidia", 2)] cotizacion
239.0

> valorCartera [("intel", 3), ("facebook", 4), ("nvidia", 2)] cotizacion
239.0
```

**Ejercicio** [`mezcla`] Define una función recursiva `mezcla :: Ord a => [a] -> [a] -> [a]` que dadas dos listas ordenadas devuelva su mezcla ordenada. Por ejemplo:

```
> mezcla [0,2,4,6,8] [1,3,5,7,9]
[0,1,2,3,4,5,6,7,8,9]

> mezcla [1,1,1] [2,2]
[1,1,1,2,2]

> mezcla [1] [2,3,4]
[1,2,3,4]

> mezcla "aeiou" "bcdfgh"
"abcdefghiou"
```

**Ejercicio** [`takeUntil`] Define una función recursiva `takeUntil :: (a -> Bool) -> [a] -> [a]` que dados un predicado y una lista tome elementos de la lista hasta encontrar el primero que verifique el predicado. Por ejemplo:

```
> takeUntil even [1,3,5,7,9,10,11,13]
[1,3,5,7,9]

> takeUntil odd [1,3,5,7,9,10,11,13]
[]
```

```
> takeUntil isUpper "saldoCuenta"
"saldo"
```

La función predefinida `takeWhile :: (a -> Bool) -> [a] -> [a]` toma elementos de una lista mientras verifiquen un predicado. Utiliza QuickCheck y `takeWhile` para comprobar tu implementación de `takeUntil`.

**Ejercicio** [**números felices**] Un número feliz es un entero positivo $n$ que verifica la siguiente propiedad: se reemplaza $n$ por la suma de los cuadrados de sus dígitos y se repite el proceso hasta obtener un 1. Por ejemplo, el 7 es feliz porque:

- $7 \to 7^2 = 49$
- $49 \to 4^2 + 9^2 = 16 + 81 = 97$
- $97 \to 9^2 + 7^2 = 81 + 49 = 130$
- $130 \to 1^2 + 3^2 + 0^2 = 1 + 9 + 0 = 10$
- $10 \to 1^2 + 0^2 = 1 + 0 = 1$
- 1

Un número que no es feliz es infeliz. En tal caso, el proceso anterior entra en bucle. Por ejemplo, el 4 es infeliz porque el proceso entra en bucle (aunque el bucle no tiene por qué presentarse en el primer elemento de la secuencia):

- $4 \to 4^2 = 16$
- $16 \to 1^2 + 6^2 = 1 + 36 = 37$
- $37 \to 3^2 + 7^2 = 9 + 49 = 58$
- $58 \to 5^2 + 8^2 = 25 + 64 = 89$
- $89 \to 8^2 + 9^2 = 64 + 81 = 145$
- $145 \to 1^2 + 4^2 + 5^2 = 1 + 16 + 25 = 42$
- $42 \to 4^2 + 2^2 = 16 + 4 = 20$
- $20 \to 2^2 + 0^2 = 4 + 0 = 4$
- 4

Define una función `digitosDe :: Integer -> [Integer]` que dado un natural devuelva un lista con sus dígitos:

```
> digitosDe 1234
[1,2,3,4]

> digitosDe 7
[7]

> digitosDe 1000
[1,0,0,0]

> digitosDe 0
[0]
```

Sugerencia: define una función local auxiliar `digitosDeAc` que utilice recursión con acumulador.

Usando composición `(.)` define una función `sumaCuadradosDigitos :: Integer -> Integer` que dado un natural devuelva la suma de los cuadrados de sus dígitos:

```
> sumaCuadradosDigitos 9
81

> sumaCuadradosDigitos 29
85

> sumaCuadradosDigitos 1234
30
```

Define una función `esFeliz :: Integer -> Bool` que determine si un entero positivo es feliz:

```
> esFeliz 7
True
```

```
> esFeliz 4
False
```

Sugerencia: define una función local auxiliar `esFelizAc` que utilice recursión con acumulador y emplea éste para almacenar los números que se van obteniendo en la sucesivas iteraciones.

Usando listas por comprensión, define la función `felicesHasta :: Integer -> [Integer]` que dado un natural $n$ devuelva todos los números felices menores o iguales que $n$:

```
> felicesHasta 20
[1,7,10,13,19]
```

¿Cuántos números felices hay menores o iguales que 1000?

**Ejercicio** [**borrar**] Define una función recursiva `borrarRec :: Eq a => a -> [a] -> [a]` que dados un valor y una lista borre todas las apariciones del valor de la lista:

```
> borrarRec 'a' "abracadabra"
"brcdbr"

> borrarRec 2 [1,2,2,3,2,2,1]
[1,3,1]

> borrarRec 2 [1,3,1]
[1,3,1]
```

Vuelve a definir la función mediante comprensión de listas (`borrarC`) y un plegado a la derecha (`borrarP`) y utiliza QuickCheck para comprobar que las tres versiones son equivalentes.

**Ejercicio** [**agrupar**] Usando un plegado a la derecha, define una función `agrupar :: Eq a => [a] -> [[a]]` que dada una lista agrupe en sublistas los elementos consecutivos iguales:

```
> agrupar "mississippi"
["m","i","ss","i","ss","i","pp","i"]

> agrupar [1,2,2,3,3,3,4,4,4,4,3,3,3,2,2,1]
[[1],[2,2],[3,3,3],[4,4,4,4],[3,3,3],[2,2],[1]]

> agrupar [1,2,3,2,1]
[[1],[2],[3],[2],[1]]
```

**Ejercicio** [**aplica**] Define una función recursiva `aplicaRec :: a -> [(a->b)] -> [b]` que dados un valor y una lista de funciones devuelva una lista con los resultados de aplicar cada una de las funciones al valor:

```
> aplicaRec 5 [(*2), (+3), (^2), (2^)]
[10,8,25,32]

> aplicaRec pi [cos, sin, (^2)]
[-1.0,1.2246063538223773e-16,9.869604401089358]
```

Vuelve a definir la función mediante comprensión de listas (`aplicaC`), un `map` (`aplicaM`), y un plegado a la derecha (`aplicaP`). Utiliza QuickCheck para comprobar que las cuatro versiones son equivalentes.