

TEMA A

Ejercicio 1: Programar la función

```
estaEnDNI :: Int -> Bool
```

que dado un número, devuelve `True` si es una de las cifras de tu DNI. Por ejemplo, si tu número de DNI es `40.123.321`:

```
estaEnDNI 4 debe dar True
estaEnDNI 7 debe dar False
estaEnDNI 0 debe dar True
estaEnDNI 1 debe dar True
estaEnDNI 40 debe dar False
```

Ejercicio 2: Programar la función recursiva (utilizando *pattern matching*)

```
sumaDNI :: [Int] -> Int
```

que dada una lista de enteros `xs` suma solo los elementos que son cifras de tu DNI. Por ejemplo, si tu número de DNI es `40.123.321`:

```
sumaDNI [3, 5, -1, 2, 123, 50, 4] = 3 + 2 + 4 = 9
```

Ejercicio 3: Programa mediante composición sin recursión, usando `sumatoria'` definida en el apartado **4c del Proyecto I**, la función

```
sumaDNI' :: [Int] -> Int
```

que al igual que `sumaDNI` suma solo los elementos que son cifras de tu DNI.

TIP: Pueden definir una función auxiliar con tipo `Int -> Int` que utilice a `estaEnDNI`

Ejercicio 4 (*): Programar mediante recursión la función

```
reducir :: [a] -> (a -> a -> a) -> a
```

que dada una lista `xs` y un operador `op` realiza dicha operación entre todos los elementos de `xs`, por ejemplo:

```
reducir [3, 4, 5, 6] (+) = 3 + 4 + 5 + 6 = 18
reducir [3, 4, 5, 6] (*) = 3 * 4 * 5 * 6 = 360
reducir [3, 4, 5, 6] f = f 3 (f 4 (f 5 6))
```

para ello usar un caso base con listas de un solo elemento y un caso inductivo con listas de al menos dos elementos. La función `reducir` no funciona para las listas vacías.

TEMA B

Ejercicio 1: Programar la función

```
estaEnDNI :: Int -> Bool
```

que dado un número, devuelve `True` si es una de las cifras de tu DNI. Por ejemplo, si tu número de DNI es `40.123.321`:

```
estaEnDNI 4 debe dar True
estaEnDNI 7 debe dar False
estaEnDNI 0 debe dar True
estaEnDNI 1 debe dar True
estaEnDNI 40 debe dar False
```

Ejercicio 2: Programar la función recursiva (utilizando *pattern matching*)

```
cuentaDNI :: [Int] -> Int
```

que dada una lista de enteros `xs` cuenta la cantidad de elementos que son cifras de tu DNI. Por ejemplo, si tu número de DNI es `40.123.321`:

```
cuentaDNI [3, 5, 21, 2, 50, 4, 40] = 3
```

Ejercicio 3: Programa mediante composición sin recursión, usando `sumatoria'` definida en el apartado **4c del Proyecto I**, la función

```
cuentaDNI' :: [Int] -> Int
```

que al igual que `cuentaDNI` cuenta cuantos elementos son cifras de tu DNI.

TIP: Pueden definir una función auxiliar con tipo `Int -> Int` que utilice a `estaEnDNI`

Ejercicio 4 (*): Programar mediante recursión la función

```
separar :: [a] -> (a -> Bool) -> ([a],[a])
```

que dada una lista `xs` y un predicado `(a -> Bool)` devuelve un par de listas en la primera los elementos que al aplicar el predicado dan `True` y en la segunda los elementos que al aplicar el predicado dan `False`.

```
separar [3, 4, 5, 6] odd debe devolver ([3,5],[4,6])
```

```
separar ["hola","no","si","chau"] tieneA
debe devolver
```

```
(["hola", "chau"],["no", "si"])
```

donde la función `tieneA :: Char -> Bool` devuelve `True` si la palabra contiene el carácter `'a'`.

TEMA C

Ejercicio 1: Programar la función

```
estaEnDNI :: Int -> Bool
```

que dado un número, devuelve `True` si es una de las cifras de tu DNI. Por ejemplo, si tu número de DNI es `40.123.321`:

```
estaEnDNI 4 debe dar True
estaEnDNI 7 debe dar False
estaEnDNI 0 debe dar True
estaEnDNI 1 debe dar True
estaEnDNI 40 debe dar False
```

Ejercicio 2: Programar la función recursiva (utilizando *pattern matching*)

```
cuentaNoDNI :: [Int] -> Int
```

que dada una lista de enteros `xs` cuenta la cantidad de elementos que no son cifras de tu DNI. Por ejemplo, si tu número de DNI es `40.123.321`:

```
cuentaNoDNI [6, 5, 1, 40, 2, 4, 7] = 4
```

Ejercicio 3: Programa mediante composición sin recursión, usando `sumatoria'` definida en el apartado **4c del Proyecto I**, la función

```
cuentaNoDNI' :: [Int] -> Int
```

que al igual que `cuentaNoDNI` cuenta los elementos de `xs` que no son cifras de tu DNI.

TIP: Pueden definir una función auxiliar con tipo `Int -> Int` que utilice a `estaEnDNI`

Ejercicio 4 (*): Programar mediante recursión la función

```
aplicaSegun :: [Int] -> Int -> (Int -> a) -> (Int -> a) -> [a]
```

que dada una lista `xs`, un entero `n` y dos funciones `f` y `g`, devuelve una lista en la que a cada elemento de la lista `xs` si es mayor o igual que `n` le aplica la función `f` y si es menor que `n` le aplica la función `g`.

```
aplicaSegun [1, 3, 4, 5, 6, 2] 4 (+2) (*2) = [2, 6, 6, 7, 8, 4]
```

```
aplicaSegun [3, 1, 5, 7] 4 even odd = [True, True, False, False]
```

TEMA D

Ejercicio 1: Programar la función

```
letraEnApellido :: Char -> Bool
```

que dada una caracter devuelve `True` si y solo si es una de las letras de tu apellido (en minúscula). **Deben aclarar en la solución con un comentario cuál de sus apellidos usan como referencia.** Por ejemplo, si tu apellido es **Argento**:

```
letraEnApellido 'r' debe dar True
letraEnApellido 'u' debe dar False
letraEnApellido 'a' debe dar True
letraEnApellido 'R' debe dar False (porque es mayúscula)
letraEnApellido 't' debe dar True
letraEnApellido 'p' debe dar False
```

Ejercicio 2: Programar la función recursiva (utilizando *pattern matching*)

```
cuentaEnApellido :: [Char] -> Int
```

que dada una lista de caracteres `xs` devuelve la cantidad de letras (en minúsculas) en `xs` que están en tu apellido. **Deben aclarar en la solución con un comentario cuál de sus apellidos usan como referencia.** Por ejemplo, si tu apellido es **Argento**:

```
cuentaEnApellido "Suerte en el parcialito!" = 12
cuentaEnApellido "trrG" = 3
cuentaEnApellido "aRGENTO" = 1
```

Ejercicio 3: Programa mediante composición sin recursión, usando `sumatoria` definida en el apartado **4c del Proyecto I**, la función

```
cuentaEnApellido' :: [Char] -> Int
```

que al igual que `cuentaEnApellido` cuenta los caracteres de `xs` que son letras minúsculas de tu apellido.

TIP: Pueden definir una función auxiliar con tipo `Char -> Int` que utilice a la función `letraEnApellido`

Ejercicio 4 (*): Programar mediante recursión la función

```
alterna :: [a] -> (a -> b) -> (a -> b) -> [b]
```

que dada una lista `xs` y dos funciones `f` y `g`, devuelve una lista en la que a cada elemento de la lista `xs` se le aplica de manera alternada las funciones `f` y `g`. Por ejemplo:

```
alterna [1, 2, 3, 4] (+1) (*2) = [2, 4, 4, 8]
alterna [1, 1, 1, 1] odd even = [True, False, True, False]
```