



Ejercicio 1. Definir las siguientes funciones sobre listas:

1. `longitud :: [t] -> Integer`, que dada una lista devuelve su cantidad de elementos.
2. `ultimo :: [t] -> t` según la siguiente especificación:

```
problema ultimo (s: seq⟨T⟩) : T {  
    requiere: { |s| > 0 }  
    asegura: { resultado = s[|s| - 1] }  
}
```

3. `principio :: [t] -> [t]` según la siguiente especificación:

```
problema principio (s: seq⟨T⟩) : seq⟨T⟩ {  
    requiere: { |s| > 0 }  
    asegura: { resultado = subseq(s, 0, |s| - 1) }  
}
```

4. `reverso :: [t] -> [t]` según la siguiente especificación:

```
problema reverso (s: seq⟨T⟩) : seq⟨T⟩ {  
    requiere: { True }  
    asegura: { resultado tiene los mismos elementos que s pero en orden inverso. }  
}
```

Ejercicio 2. Definir las siguientes funciones sobre listas:

1. `pertenece :: (Eq t) => t -> [t] -> Bool` según la siguiente especificación:

```
problema pertenece (e: T, s: seq⟨T⟩) : ℬ {  
    requiere: { True }  
    asegura: { resultado = true ↔ e ∈ s }  
}
```

2. `todosIguales :: (Eq t) => [t] -> Bool`, que dada una lista devuelve verdadero sí y solamente sí todos sus elementos son iguales.

3. `todosDistintos :: (Eq t) => [t] -> Bool` según la siguiente especificación:

```
problema todosDistintos (s: seq⟨T⟩) : ℬ {  
    requiere: { True }  
    asegura: { resultado = false ↔ existen dos posiciones distintas de s con igual valor }  
}
```

4. `hayRepetidos :: (Eq t) => [t] -> Bool` según la siguiente especificación:

```
problema hayRepetidos (s: seq⟨T⟩) : ℬ {  
    requiere: { True }  
    asegura: { resultado = true ↔ existen dos posiciones distintas de s con igual valor }  
}
```

5. `quitar :: (Eq t) => t -> [t] -> [t]`, que dados un entero x y una lista xs , elimina la primera aparición de x en la lista xs (de haberla).
6. `quitarTodos :: (Eq t) => t -> [t] -> [t]`, que dados un entero x y una lista xs , elimina todas las apariciones de x en la lista xs (de haberlas). Es decir:

```
problema quitarTodos (e: T, s: seq⟨T⟩) : seq⟨T⟩ {
  requiere: { True }
  asegura: { resultado es igual a s pero sin el elemento e. }
}
```

7. `eliminarRepetidos :: (Eq t) => [t] -> [t]` que deja en la lista una única aparición de cada elemento, eliminando las repeticiones adicionales.
8. `mismosElementos :: (Eq t) => [t] -> [t] -> Bool`, que dadas dos listas devuelve verdadero sí y solamente sí ambas listas contienen los mismos elementos, sin tener en cuenta repeticiones, es decir:

```
problema mismosElementos (s: seq⟨T⟩, r: seq⟨T⟩) : ℤ {
  requiere: { True }
  asegura: { resultado = true ↔ todo elemento de s pertenece r y viceversa }
}
```

9. `capicua :: (Eq t) => [t] -> Bool` según la siguiente especificación:

```
problema capicua (s: seq⟨T⟩) : ℤ {
  requiere: { True }
  asegura: { (resultado = true) ↔ (s = reverso(s)) }
}
```

Por ejemplo `capicua [á', 'c', 'b', 'b', 'c', á']` es `true`, `capicua [á', 'c', 'b', 'd', á']` es `false`.

Ejercicio 3. Definir las siguientes funciones sobre listas de enteros:

1. `sumatoria :: [Integer] -> Integer` según la siguiente especificación:

```
problema sumatoria (s: seq⟨ℤ⟩) : ℤ {
  requiere: { True }
  asegura: { resultado =  $\sum_{i=0}^{|s|-1} s[i]$  }
}
```

2. `productoria :: [Integer] -> Integer` según la siguiente especificación:

```
problema productoria (s: seq⟨ℤ⟩) : ℤ {
  requiere: { True }
  asegura: { resultado =  $\prod_{i=0}^{|s|-1} s[i]$  }
}
```

3. `maximo :: [Integer] -> Integer` según la siguiente especificación:

```
problema maximo (s: seq⟨ℤ⟩) : ℤ {
  requiere: { |s| > 0 }
  asegura: { resultado ∈ s ∧ todo elemento de s es menor o igual a resultado }
}
```

4. `sumarN :: Integer -> [Integer] -> [Integer]` según la siguiente especificación:

```
problema sumarN (n: ℤ, s: seq⟨ℤ⟩) : seq⟨ℤ⟩ {
  requiere: { True }
  asegura: { |resultado| = |s| ∧ cada posición de resultado contiene el valor que hay en esa posición en s sumado n }
}
```

5. `sumarElPrimero :: [Integer] -> [Integer]` según la siguiente especificación:

```
problema sumarElPrimero (s: seq⟨ℤ⟩) : seq⟨ℤ⟩ {  
    requiere: { |s| > 0 }  
    asegura: { resultado = sumarN(s[0], s) }  
}
```

Por ejemplo `sumarElPrimero [1,2,3]` da `[2,3,4]`

6. `sumarElUltimo :: [Integer] -> [Integer]` según la siguiente especificación:

```
problema sumarElUltimo (s: seq⟨ℤ⟩) : seq⟨ℤ⟩ {  
    requiere: { |s| > 0 }  
    asegura: { resultado = sumarN(s[|s| - 1], s) }  
}
```

Por ejemplo `sumarElUltimo [1,2,3]` da `[4,5,6]`

7. `pares :: [Integer] -> [Integer]` según la siguiente especificación:

```
problema pares (s: seq⟨ℤ⟩) : seq⟨ℤ⟩ {  
    requiere: { True }  
    asegura: { resultado sólo tiene los elementos pares de s en el orden dado, respetando las repeticiones }  
}
```

Por ejemplo `pares [1,2,3,5,8,2]` da `[2,8,2]`

8. `multiplosDeN :: Integer -> [Integer] -> [Integer]` que dado un número n y una lista xs , devuelve una lista con los elementos de xs múltiplos de n .

9. `ordenar :: [Integer] -> [Integer]` que ordena los elementos de la lista en forma creciente. *Sugerencia:* Pensar cómo pueden usar la función máximo para que ayude a generar la lista ordenada necesaria.

Ejercicio 4. a) Definir las siguientes funciones sobre listas de caracteres, interpretando una palabra como una secuencia de caracteres sin blancos:

- a) `sacarBlancosRepetidos :: [Char] -> [Char]`, que reemplaza cada subsecuencia de blancos contiguos de la primera lista por un solo blanco en la lista resultado.
- b) `contarPalabras :: [Char] -> Integer`, que dada una lista de caracteres devuelve la cantidad de palabras que tiene.
- c) `palabras :: [Char] -> [[Char]]`, que dada una lista arma una nueva lista con las palabras de la lista original.
- d) `palabraMasLarga :: [Char] -> [Char]`, que dada una lista de caracteres devuelve su palabra más larga.
- e) `aplanar :: [[Char]] -> [Char]`, que a partir de una lista de palabras arma una lista de caracteres concatenándolas.
- f) `aplanarConBlancos :: [[Char]] -> [Char]`, que a partir de una lista de palabras, arma una lista de caracteres concatenándolas e insertando un blanco entre cada palabra.
- g) `aplanarConNBlancos :: [[Char]] -> Integer -> [Char]`, que a partir de una lista de palabras y un entero n , arma una lista de caracteres concatenándolas e insertando n blancos entre cada palabra (n debe ser no negativo).

b) ¿Cómo cambian los ejercicios si agregamos el renombre de tipos: `type Texto = [Char]`?

Ejercicio 5. Definir las siguientes funciones sobre listas:

1. `sumaAcumulada :: (Num t) => [t] -> [t]` según la siguiente especificación:

```
problema sumaAcumulada (s: seq⟨T⟩) : seq⟨T⟩ {  
    requiere: { T es un tipo numérico }  
    requiere: { cada elemento de s es mayor estricto que cero }  
    asegura: { |s| = |resultado| ∧ el valor en la posición i de resultado es  $\sum_{k=0}^i s[k]$  }  
}
```

Por ejemplo `sumaAcumulada [1, 2, 3, 4, 5]` es `[1, 3, 6, 10, 15]`.

2. `descomponerEnPrimos :: [Integer] -> [[Integer]]` según la siguiente especificación:

```
problema descomponerEnPrimos (s: seq(Z)) : seq(seq(Z)) {  
    requiere: { Todos los elementos de s son mayores a 2 }  
    asegura: { |resultado| = |s| }  
    asegura: { todos los valores en las listas de resultado son números primos }  
    asegura: { multiplicar todos los elementos en la lista en la posición i de resultado es igual al valor en la posición i de s }  
}
```

Por ejemplo `descomponerEnPrimos [2, 10, 6]` es `[[2], [2, 5], [2, 3]]`.

Ejercicio 6. En este ejercicio trabajaremos con la lista de contactos del teléfono.

- Implementar una función que me diga si una persona aparece en mi lista de contactos del teléfono: `enLosContactos :: Nombre -> ContactosTel -> Bool`
- Implementar una función que agregue una nueva persona a mis contactos, si esa persona está ya en mis contactos entonces actualiza el teléfono. `agregarContacto :: Contacto -> ContactosTel -> ContactosTel`
- Implementar una función que dado un nombre, elimine un contacto de mis contactos. Si esa persona no está no hace nada. `eliminarContacto :: Nombre -> ContactosTel -> ContactosTel`

Para esto definiremos los siguientes tipos:

- `type Texto = [Char]`
- `type Nombre = Texto`
- `type Telefono = Texto`
- `type Contacto = (Nombre, Telefono)`
- `type ContactosTel = [Contacto]`

Sugerencia: Implementar las funciones auxiliares `elNombre` y `elTelefono` para que dado un `Contacto` devuelva el dato del nombre y el teléfono respectivamente.

Ejercicio 7. En este ejercicio trabajaremos con lockers de una facultad.

Para resolverlo usaremos un tipo `MapaDeLockers` que será una secuencia de `locker`.

Cada `locker` es una tupla con la primera componente correspondiente al número de identificación, y la segunda componente el estado.

El estado es a su vez una tupla cuya primera componente dice si esta ocupado (`False`) o libre (`True`), y la segunda componente es un texto con el código de ubicación del locker.

```
type Identificacion = Integer  
type Ubicacion = Texto  
type Estado = (Disponibilidad, Ubicacion)  
type Locker = (Identificacion, Estado)  
type MapaDeLockers = [Locker]  
type Disponibilidad = Bool
```

- Implementar `existeElLocker :: Identificacion -> MapaDeLockers -> Bool`, una función para saber si un locker existe en la facultad.
- Implementar `ubicacionDelLocker :: Identificacion -> MapaDeLockers -> Ubicacion`, una función que dado un locker que existe en la facultad, me dice la ubicación del mismo.
- Implementar `estaDisponibleElLocker :: Identificacion -> MapaDeLockers -> Bool`, una función que dado un locker que existe en la facultad, me devuelve Verdadero si esta libre.
- Implementar `ocuparLocker :: Identificacion -> MapaDeLockers -> MapaDeLockers`, una función que dado un locker que existe en la facultad, y está libre, lo ocupa.

Por ejemplo, un posible mapa de lockers puede ser:

```
lockers =  
[  
  (100, (False, "ZD39I")),  
  (101, (True, "JAH3I")),  
  (103, (True, "IQSA9")),  
  (105, (True, "QOTSA")),  
  (109, (False, "893JJ")),  
  (110, (False, "99292"))  
]
```