



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⟩) : seq⟨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⟩) : ℤ {  
  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⟩) : ℤ {  
  requiere: { True }  
  asegura: { resultado = false ↔ (∃ i, j : ℤ) (0 ≤ i < |s| ∧ 0 ≤ j < |s| ∧ i ≠ j ∧ s[i] = s[j]) }  
}
```

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

```
problema hayRepetidos (s: seq⟨T⟩) : ℤ {  
  requiere: { True }  
  asegura: { resultado = true ↔ (∃ i, j : ℤ) (0 ≤ i < |s| ∧ 0 ≤ j < |s| ∧ i ≠ j ∧ s[i] = s[j]) }  
}
```

5. `quitar :: (Eq t) => t -> [t] -> [t]`, que dada una lista xs y un elemento x , elimina la primera aparición de x en la lista xs (de haberla).
6. `quitarTodos :: (Eq t) => t -> [t] -> [t]`, que dada una lista xs y un elemento x , 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 ↔ (∀ e : T)(e ∈ s ↔ e ∈ r) }
}
```

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

```
problema capicua (s: seq⟨T⟩) : ℤ {
  requiere: { True }
  asegura: { resultado = true ↔ (∀ i : ℤ)(0 ≤ i < ⌊|s|/2⌋ → s[i] = s[|s| - 1 - i]) }
}
```

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 = ∑i=0|s|-1 s[i] }
}
```

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

```
problema productoria (s: seq⟨ℤ⟩) : ℤ {
  requiere: { True }
  asegura: { resultado = ∏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 ∧ (∀ i : ℤ)(0 ≤ i < |s| → resultado ≥ s[i] ) }
}
```

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

```
problema sumarN (n: ℤ, s: seq⟨ℤ⟩) : seq⟨ℤ⟩ {
  requiere: { True }
  asegura: { |resultado| = |s| ∧ (∀ i : ℤ)(0 ≤ i < |s| → resultado[i] = s[i] + n ) }
}
```

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

```
problema sumarElPrimero (s: seq⟨ℤ⟩) : seq⟨ℤ⟩ {  
    requiere: { |s| > 0 }  
    asegura: { |resultado| = |s| ∧ (∀ i : ℤ)(0 ≤ i < |s| → resultado[i] = s[i] + s[0]) }  
}
```

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| = |s| ∧ (∀ i : ℤ)(0 ≤ i < |s| → resultado[i] = s[i] + s[|s| - 1]) }  
}
```

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.

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

1. `sacarBlancosRepetidos :: [Char] -> [Char]`, que reemplaza cada subsecuencia de blancos contiguos de la primera lista por un solo blanco en la segunda lista.
2. `contarPalabras :: [Char] -> Integer`, que dada una lista de caracteres devuelve la cantidad de palabras que tiene.
3. `palabraMasLarga :: [Char] -> [Char]`, que dada una lista de caracteres devuelve su palabra más larga.
4. `palabras :: [Char] -> [[Char]]`, que dada una lista arma una nueva lista con las palabras de la lista original.
5. `aplanar :: [[Char]] -> [Char]`, que a partir de una lista de palabras arma una lista de caracteres concatenándolas.
6. `aplanarConBlancos :: [[Char]] -> [Char]`, que a partir de una lista de palabras, arma una lista de caracteres concatenándolas e insertando un blanco entre cada palabra.
7. `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).

Ejercicio 5. Definir las siguientes funciones sobre listas:

1. `nat2bin :: Integer -> [Integer]`, que recibe un número no negativo y lo transforma en una lista de bits correspondiente a su representación binaria. Por ejemplo `nat2bin 8` devuelve `[1, 0, 0, 0]`.
2. `bin2nat :: [Integer] -> Integer` según la siguiente especificación:

```
problema bin2nat (s: seq⟨ℤ⟩) : ℤ {  
    requiere: { s es una lista de 0's y 1's }  
    asegura: { resultado es el número entero no negativo cuya representación binaria es s. }  
}
```

Por ejemplo `bin2nat [1, 0, 0, 0, 1]` devuelve 17.

3. `nat2hex :: Integer -> [Char]`, que recibe un número no negativo y lo transforma en una lista de caracteres correspondiente a su representación hexadecimal. Por ejemplo `nat2hex 45` devuelve `['2', 'D']`.

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

```
problema sumaAcumulada (s: seq⟨T⟩) : seq⟨T⟩ {
  requiere: { T ∈ [N, Z, R] }
  asegura: { (∀ i : Z) (0 ≤ i < |s| → resultado[i] = ∑k=0i s[k]) }
}
```

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

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

```
problema descomponerEnPrimos (s: seq⟨Z⟩) : seq⟨seq⟨Z⟩⟩ {
  requiere: { True }
  asegura: { resultado es lista de listas de enteros, que resulta de descomponer en números primos cada uno de los números de s, manteniendo el orden. }
}
```

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

Ejercicio 6. Definir las siguientes funciones sobre conjuntos:

1. `agregarATodos :: Integer -> Set (Set Integer) -> Set (Set Integer)` que dado un número n y un conjunto de conjuntos de enteros cls agrega a n en cada conjunto de cls .
2. `partes :: Integer -> Set (Set Integer)` que genere todos los subconjuntos del conjunto $\{1, 2, \dots, n\}$. Por ejemplo `partes 2` es `[[], [1], [2], [1, 2]]`.
3. `productoCartesiano :: Set Integer -> Set Integer -> Set (Integer, Integer)` según la siguiente especificación:

```
problema productoCartesiano (s: seq⟨Z⟩, r: seq⟨Z⟩) : seq⟨< Z, Z >⟩ {
  requiere: { sinRepetidos(s) ∧ sinRepetidos(r) }
  asegura: { resultado es el conjunto de todas las duplas posibles (como pares de dos elementos) tomando el primer elemento de s y el segundo elemento de r. }
}

pred sinRepetidos(s : seq⟨Z⟩) {
  (∀ i, j : Z) (0 ≤ i < |s| ∧ 0 ≤ j < |s| ∧ i ≠ j → s[i] ≠ s[j])
}
```

Por ejemplo `productoCartesiano [1, 2, 3] [3, 4]` es `[(1, 3), (2, 3), (3, 3), (1, 4), (2, 4), (3, 4)]`.