Ejercicio 1. 2 puntos

1. [1 punto] Dada la siguiente especificación relacionada a un juego de mesa completar nombres adecuados para el problema a, el parámetro b, las etiquetas x, y, s, t, u, v y w y los predicados $p1$, $p2$, $p3$, $p4$ y $p5$. Justifique los nombres elegidos describiendo brevemente el problema.

```
problema a (in b: seq⟨ℤ⟩ ) : ℤ  {
      requiere x: {(∀i : ℤ)(0 ≤ i < |b| → 0 < b[i] < 7)}
      requiere y: { p1(b) }
      requiere z: { |b| = 5 }
      asegura s: { p2(b) ⟷ (resultado = 55) }
      asegura t: { p3(b) ⟷ (resultado = 45) }
      asegura u: { p4(b) ⟷ (resultado = 35) }
      asegura v: { p5(b) ⟷ (resultado = 25) }
      asegura w: { (¬p2(b) ∧ ¬p3(b) ∧ ¬p4(b) ∧ ¬p5(b)) ⟷ (resultado = 0)}
}
pred p1 (b:seq⟨ℤ⟩) {
    (∀i, j : ℤ)((0 ≤ i < |b| ∧ 0 ≤ j < |b| ∧ i ≤ j) ⟶ (b[i] ≤ b[j]))
}
pred p2 (b:seq⟨ℤ⟩) {
    (∀i, j : ℤ)((0 ≤ i < |b| ∧ 0 ≤ j < |b|) ⟶ (b[i] = b[j]))
}
pred p3 (b:seq⟨ℤ⟩) {
    (∃n, m : ℤ)(m ≠ n ∧ n ∈ b ∧ m ∈ b ∧ cantAp(b, n) = 1 ∧ cantAp(b, m) = 4)
}
pred p4 (b:seq⟨ℤ⟩) {
    (∃n, m : ℤ)(m ≠ n ∧ n ∈ b ∧ m ∈ b ∧ cantAp(b, n) = 2 ∧ cantAp(b, m) = 3)
}
pred p5 (b:seq⟨ℤ⟩) {
    (∃n, m, o, p, q : ℤ)(n ∈ b ∧ m ∈ b ∧ o ∈ b ∧ p ∈ b ∧ q ∈ b ∧ m = n + 1 ∧ o = m + 1 ∧ p = o + 1 ∧ q = p + 1)
}
```

Resolucion: Veamos algunos ejemplos de valores válidos para b:

- Sabemos que b es una lista de enteros.
- Por el predicado z sabemos que b tiene exactamente 5 elementos.
- Por el predicado x sabemos que todos los elementos de b toman valores en el conjunto $\{1, 2, 3, 4, 5, 6\}$.
- El predicado y nos dice que b satiface el predicado $p1$, que dice que los elementos de b estan ordenados de manera creciente.

Entonces podemos pensar en valores de $b =< 1, 1, 2, 3, 4 >$ o $b =< 5, 5, 5, 6, 6 >$. Ejemplos de valores inválidos serian $b =< 6, 5, 3, 2, 1 >$ (no está ordenado) y $b =< 4, 5, 8 >$ (no tiene 5 elementos y tiene valores fuera del rango válido).

Veamos los asegura del problema. Nos dicen que el resultado del problema va tomar valores diferentes de acuerdo a las características de los elementos de b:

- El predicado s nos dice que $resultado = 55$ sii se cumple $p2$: que todos los elementos de b sean iguales entre si
- El predicado t nos dice que $resultado = 45$ sii se cumple $p3$: que los elementos de b toman 2 valores distintos entre si y que uno de los valores aparece exactamente 1 vez y el otro 4 veces
- El predicado u es análogo y nos dice que $resultado = 35$ sii se cumple $p4$: que los elementos de b toman 2 valores distintos entre si y que uno de los valores aparece exactamente 2 veces y el otro 3 veces
- El predicado v nos dice que $resultado = 25$ sii los elementos de b son consecutivos
- El predicado w nos dice que $resultado = 0$ sii no se cumplen ninguna de las situaciones anteriores

De esta forma tenemos el juego que popularmente se conoce como "la generala" o "generala": se tiran 5 dados (de 6 caras, numerados del 1 al 6) y se asigna un puntaje a la combinación de valores obtenidos:

- 55 puntos si todos los dados son idénticos (generala)
- 45 puntos si hay 4 dados idénticos y 1 diferente (poker)
- 35 puntos si hay 2 dados idénticos entre si y 3 idénticos entre si, pero diferentes a los anteriores (full)
- 25 puntos si hay 5 dados consectivos (escalera)
- 0 puntos si no se observa ninguno de los patrones anteriores

Una propuesta de etiquetas (se podrían haber usado otras):

$a = puntajeJugada$
$b = dados$

$x = sonDados$
$y = estanOrdenados$
$z = tiene5Elementos$

$s = puntajeGenerala$
$t = puntajePoker$
$u = puntajeFull$
$v = puntajeEscalera$
$w = puntajeCero$

$p1 = estaOrdenado$
$p2 = esGenerala$
$p3 = esPoker$
$p4 = esFull$
$p5 = esEscalera$

2. [1 punto] Especificar el siguiente problema (se puede especificar de manera formal o semi-formal):

   Dados los inputs *jugadasJugador1:* $seq\langle seq\langle\mathbb{Z}\rangle\rangle$, *jugadasJugador2:*$seq\langle seq\langle\mathbb{Z}\rangle\rangle$, retornar 0, 1 o 2 según quién sea el jugador que obtiene más puntos. Los parámetros *jugadasJugador1* y *jugadasJugador2* representan jugadas. El puntaje de cada jugador será el resultado de aplicar el `problema a` (del punto 1.1) a cada elemento de la secuencias *jugadasJugador1* y *jugadasJugador2* y luego sumar todos estos valores. En caso de empate, el resultado será 0. En este juego, todos los jugadores tienen siempre la misma cantidad de jugadas.

   Resolucion: Nos piden especificar un nuevo problema. Como nos dicen los tipos de los parámetros de entrada, lo primero que podemos hacer es elegir un nombre de problema y escribir su signatura:

   `problema quienGano` (in jugadasJugador1: $seq\langle seq\langle\mathbb{Z}\rangle\rangle$, in jugadasJugador2: $seq\langle seq\langle\mathbb{Z}\rangle\rangle$) : $\mathbb{Z}$ {
   }

   En este punto podemos pensar en los aseguras del problema. De acuerdo al enuciado nos dicen que el valor del resultado depende de la suma de los puntajes de las jugadas:

   - $resultado = 1 \iff$ el puntaje de las jugadas del jugador 1 es mayor al del jugador 2
   - $resultado = 2 \iff$ el puntaje de las jugadas del jugador 1 es menor al del jugador 2
   - $resultado = 0 \iff$ el puntaje de las jugadas del jugador 1 es igual al del jugador 2

   Podemos pensar entonces que contamos con un problema auxiliar que, dado una secuencia de jugadas, nos devuelve el puntaje total:

   `problema totalJugadas` (in jugadas: $seq\langle seq\langle\mathbb{Z}\rangle\rangle$) : $\mathbb{Z}$ {
   }

   Este problema va a utilizar el problema del punto 1.1 (puntajeJugada), entonces necesitamos satisfacer sus requiere:

   `problema totalJugadas` (in jugadas: $seq\langle seq\langle\mathbb{Z}\rangle\rangle$) : $\mathbb{Z}$ {
   `requiere sonTodasJugadasValidas:` {para todo elemento j en $jugadas$ se cumple $esJugadaValida$ }
   `asegura:` {resultado = sumatoria de puntajeJugada() sobre todos los elementos de $jugadas$}
   }

   Donde usamos un predicado $esJugadaValida$ que lo armamos con los predicados del primer problema:
   $esJugadaValida(j) = \{sonDados(j) \wedge estanOrdenados(j) \wedge tiene5Elementos(j)\}$

Usando este problema auxiliar, podemos volver a escribir el problema que nos piden:

`problema quienGano` (in jugadasJugador1: $seq\langle seq\langle \mathbb{Z} \rangle\rangle$, in jugadasJugador2: $seq\langle seq\langle \mathbb{Z} \rangle\rangle$) : $\mathbb{Z}$ {

`asegura:` { $resultado = 1 \iff totalJugadas(jugadasJugador1) > totalJugadas(jugadasJugador2)$ }

`asegura:` { $resultado = 2 \iff totalJugadas(jugadasJugador1) < totalJugadas(jugadasJugador2)$ }

`asegura:` { $resultado = 0 \iff totalJugadas(jugadasJugador1) = totalJugadas(jugadasJugador2)$ }

}

Pero nuevamente, para poder usar $totalJugadas()$ tenemos que satisfacer sus requiere (que son a su vez los de $puntajeJugada()$). Además, nos dicen que para poder calcular $quienGano()$ las secuencias de jugadas tienen que tener la misma longitud, por lo que los agregamos como nuevos requiere:

`problema quienGano` (in jugadasJugador1: $seq\langle seq\langle \mathbb{Z} \rangle\rangle$, in jugadasJugador2: $seq\langle seq\langle \mathbb{Z} \rangle\rangle$) : $\mathbb{Z}$ {

`requiere mismaLongitud:` { $|jugadasJugador1| = |jugadasJugador2|$ }

`requiere sonJugadasValidas:` { $(\forall i : \mathbb{Z})(0 \leq i < |jugadasJugador1| \rightarrow esJugadaValida(jugadasJugador1[i]) \wedge esJugadaValida(jugadasJugador2[i])$ }

`asegura:` { $resultado = 1 \iff totalJugadas(jugadasJugador1) > totalJugadas(jugadasJugador2)$ }

`asegura:` { $resultado = 2 \iff totalJugadas(jugadasJugador1) < totalJugadas(jugadasJugador2)$ }

`asegura:` { $resultado = 0 \iff totalJugadas(jugadasJugador1) = totalJugadas(jugadasJugador2)$ }

}

Ejercicio 2. 4 puntos

1. [2 puntos] Programar en Haskell una función que satisfaga la especificación del `problema a` del Ejercicio 1. Recordá escribir los tipos de los parámetros.

   Pueden asumir como existentes las siguientes funciones sobre listas: *cantidadDeApariciones*, *esPermutacion*, *estaOrdenada* y *minimo* y *maximo*.

   Resolucion: comenzamos por escribir la función que calcula el puntaje, asumiendo que tenemos funciones auxiliares que saben identificar los puntajes de cada una de las jugadas:

```
puntajeJugada :: [Int] -> Int
puntajeJugada j | esGenerala j = 55
                | esPoker j = 45
                | esFull j = 35
                | esEscalera j = 25
                | otherwise = 0
```

   Ahora sólo nos quedaría implementar cada una de las funciones auxiliares. Dependiendo de la jugada a veces nos va a convenir usar pattern matching sobre los elementos de la lista y otras veces nos va a convenir identificar los valores mínimo y máximo de la jugada y contar sus apariciones en la jugada:

```
esGenerala :: [Int] -> Bool
-- sabemos que las listas son de 5 elementos (por especificacion)
-- asi que usamos pattern matching
esGenerala [a, b, c, d, e] = a == b && b == c && c == d && d == e

esPoker :: [Int] -> Bool
-- usamos cantidad de apariciones
-- tenemos que considerar los 2 casos posibles
esPoker j | cantAp(minimo, j) == 1 && cantAp(maximo, j) == 4 ||
            cantAp(minimo, j) == 4 && cantAp(maximo, j) == 1
  where minimo = min j
        maximo = max j

esFull :: [Int] -> Bool
-- nuevamente tenemos que considerar los 2 casos posibles
esFull j | cantAp(minimo, j) == 2 && cantAp(maximo, j) == 3 ||
           cantAp(minimo, j) == 3 && cantAp(maximo, j) == 2
where minimo = min j
```

```
        maximo = max j

esEscalera :: [Int] -> Bool
-- usamos pattern matching sobre los 5 elementos de la lista
esEscalera [a, b, c, d, e] = b == a + 1 && c == b + 1 && d == c + 1 && e = d + 1
```

2. [2 puntos] Programar en Python una función que implemente el enunciado del Ejercicio 1.2. Recordá escribir los tipos de los parḿetros y variables que uses en tu implementación. Dado que el Ejercicio 1.2 utiliza el Ejercicio 1.1, asumir que ya existe una implementación del Ejercicio 1.1.

   Resolucion: Nos dicen que podemos asumir que tenemos una implementación de *puntajeJugada*, entonces nos concentramos en hacer las sumas de las jugadas de ambos jugadores y compararlas para saber cual de ellos ganó. Para eso:

   a) vamos a usar una variable *total*1 para calcular el puntaje total de las jugadas del jugador 1

   b) vamos a usar otra variable *total*2 donde vamos a hacer lo mismo para el jugador 2

   c) vamos a armar un ciclo para calcular la suma del jugador1 y otro ciclo distinto para el jugador 2

   d) finalmente vamos a comparar los totales y vamos a devolver el resultado. para eso vamos a definir otra variable *res*

```
def totalJugadas(jugadasJugador1: list[list[int]], jugadasugador2: list[list[int]]) \
      -> int:

  # defino ls 3 variables que necesito
  total1: int = 0
  total2: int = 0
  res: int = 0

  # itero sobre las jugadas del jugador1 y sumo sus puntajes
  for j1 in jugadasJugador1:
    total1 = total1 + puntajeJugada(j1)

  # hago lo mismo para el jugador2
  for j2 in jugadasJugador2:
    total2 = total2 + puntajeJugada(j2)

  # me fijo quien gano
  if total1 > total2
    res = 1
  elif total1 < total2
    res = 2

  return res
```

   Notar que como inicializamos **res** en cero, no es necesario preguntar explícitamente sobre el caso del empate. Si de todas formas lo quisieramos hacer, deberíamos agregar un nuevo caso a la comparación:

```
...
# me fijo quien gano
if total1 > total2
  res = 1
elif total1 < total2
  res = 2
else:
  res = 0

return res
```

Ejercicio 3. 2 puntos

Sea la siguiente especificación del problema `sonIguales`, una posible implementación en lenguaje imperativo y el test suite:

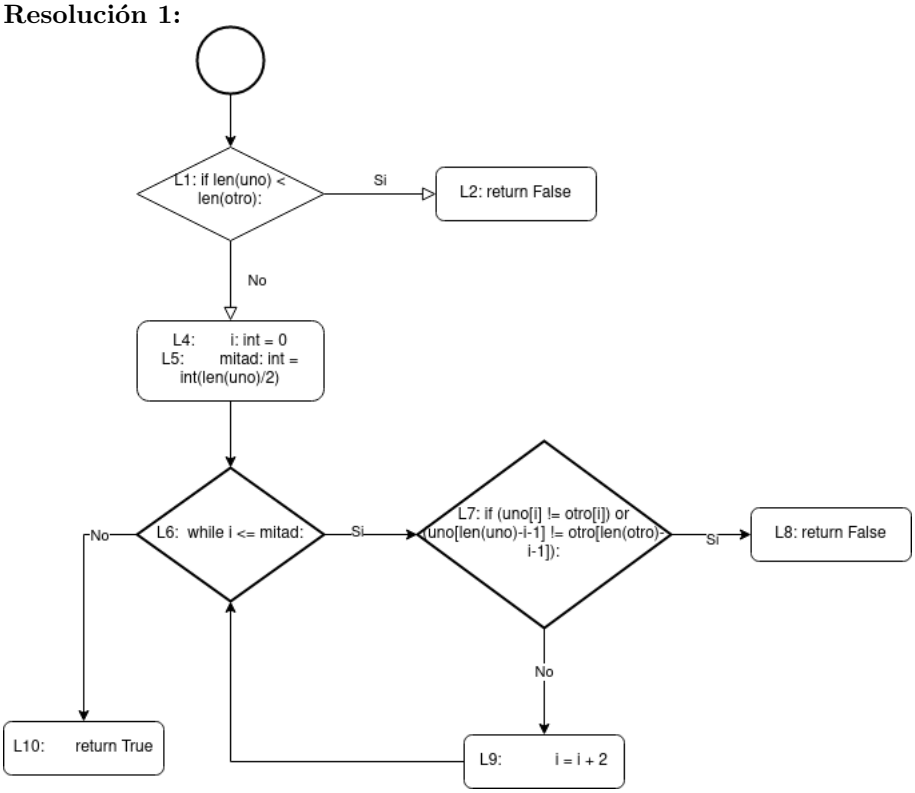
problema sonIguales (in uno: $seq\langle Char\rangle$,in otro: $seq\langle Char\rangle$) : Bool {
    requiere: $\{|uno| > 0 \land |otro| > 0\}$
    asegura: $\{result = true \leftrightarrow (|uno| = |otro| \land (\forall i : \mathbb{Z})(0 \leq i < |uno| \rightarrow uno[i] = otro[i]))\}$
}

```
def sonIguales(uno: str, otro: str) -> bool:
L1:    if len(uno) < len(otro):
L2:        return False
L3:    else:
L4:        i: int = 0
L5:        mitad: int = int(len(uno)/2) # ej: int(5/2) -> int(2.5) -> 2
L6:        while i <= mitad:
L7:            if (uno[i] != otro[i]) or (uno[len(uno)-i-1] != otro[len(otro)-i-1]):
L8:                return False
L9:            i = i + 2
L10:       return True
```

	Test1	Test2	Test3
Entrada	$uno =$ "abc" $otro =$ "abc"	$uno =$ "abc" $otro =$ "axy"	$uno =$ "ab" $otro =$ "abc"
Salida Esperada	Verdadero	Falso	Falso

1. [0.25 puntos] Dar el diagrama de control de flujo (control-flow graph) del programa `sonIguales`.
2. [0.5 puntos] ¿La ejecución del test suite resulta en la ejecución de todas las líneas del programa `sonIguales`? Justifique.
3. [0.5 puntos] ¿La ejecución del test suite resulta en la ejecución de todas las decisiones (branches) del programa? Justifique.
4. [0.75 puntos] Detalle **todos** los errores de la implementación. Agregar nuevos casos de tests y/o modificar casos de tests existentes para que el test suite detecte el/los defecto/s.

Resolución 1:



Resolución 2: Sí porque entre todos los casos se ejecutan todas las líneas:

1. Test1 ejecuta las líneas: 1, 4, 5, 6, 7, 9, 10 (no ejecuta: 2, 8)
2. Test2 ejecuta las líneas: 1, 4, 5, 6, 7, 8 (no ejecuta: 2, 9, 10)
3. Test3 ejecuta las líneas: 1, 2 (no ejecuta las otras)

Resolución 3: Sí porque entre los tres casos se ejecutan todas las decisiones:

1. Test1 ejecuta el No de la L1, el Sí de la L6, el No de la L7 y el No de la L6.
2. Test2 ejecuta el No de la L1, el Sí de la L6, el Sí de la L7
3. Test3 ejecuta el Sí de la L1

Resolución 4: En la línea 1 sólo se evalúa cuando la variable *uno* es de longitud menor a *otro*. Debería verificar si son distintos.

Se puede agregar el test entrada uno = "*abc*" otro = "*ab*", salida = *Falso* para detectar este caso.

En la línea 9 se incrementa de a dos, esto hace que sólo se revisen posiciones pares en palabras de longitud impar y se saltee verificaciones. Debería incrementar de a uno.

Se puede agregar el test entrada uno = "*axcxd*" otro = "*aycyd*", salida = *Falso* para detectar este caso.

Ejercicio 4. 2 puntos

1. [1 punto] Dada la siguiente especificación:

   `problema raizCuadrada` (in x: Float) : Float {

   `requiere:` $\{x \geq 0\}$

   `asegura:` $\{res^2 = x\}$

   }

   Indique y justifique cuáles de los siguientes algoritmos cumplen con la especificación:

   a) si $x \geq 0$ devuelve $\sqrt{x}$; si no devuelve 0
   b) si $x > 0$ devuelve $\sqrt{x}$; si no devuelve 0
   c) si $x > 1$ devuelve $\sqrt{x}$; si no devuelve 0
   d) si $x \geq 0$ devuelve $-\sqrt{x}$; si no se indefine

2. [1 punto] ¿Qué relación hay entre una especificación y un algoritmo? Dé un ejemplo de ambos conceptos.

4.1 Resolución: Los ítems que cumplen la especificación son el a, b y d. El c no cumple la especificación.

a Cumple porque para todos los valores que cumplen el requiere devuelve un resultado esperado.

b Cumple ídem a, en particular si x = 0, el resultado igual a cero es exactamente el asegura $0^2 = x$

c No cumple pues todos los valores entre 0 y 1 no devuelven $res^2 = x$, sino que devuelven 0.

d Cumple pues para todos los valores mayores o iguales a cero $res^2 = x$. Por ejemplo, x = 9, res = -3. $(-3)^2 = 9$

4.2 Resolución: La especificación es un contrato que establece **qué** debe resolver el problema, cuáles son los parámetros de entrada y salida, y cuáles son las condiciones que se deben cumplir para poder asegurar el resultado esperado.

Un algoritmo es una implementación en un lenguaje interpretable (o traducible, por ej. pseudocódigos) por una computadora, que resuelve de forma precisa una posible solución a un problema. Nos dice exactamente **cómo** se resuelve.

Podría haber más de un algoritmo para una misma especificación.

Un ejemplo de especificacion podría ser la del ejercicio 1.2. Un ejemplo de algoritmo el del ejercicio 2.2.