Taller 3: Reconocimiento de patrones

Juan Francisco Díaz Frias Emily Núñez

Septiembre 2025

1. Ejercicios de programación

En este taller se trabajará el algoritmo de codificación de Huffman para compresión de archivos. **El objetivo es adquirir destreza para escribir código funcional usando la técnica de reconocimiento de patrones.**

Esta aplicación es un método para representar datos como secuencias de unos y ceros (bits). Por ejemplo, el código estandar ASCII se usa para representar texto en el computador codificando cada caracter como una secuencia de 8 bits, lo cual permite usar 256 diferentes caracteres. En general podemos distinguir N símbolos diferentes usando $\log(N)$ bits por símbolo. Si todos nuestros mensajes son hechos con 8 símbolos: A, B, C, D, E, F, G y H podemos escoger un código con 3 bits por caracter, por ejemplo: A 000, B 001, C 010, D 011, E 100, F 101, G 110, H 111.

Con este código, el mensaje $BACADAEFAEA$ es codificado como una cadena de 33 bits: 00100001000001100010010101000100000

Los códigos como el ASCII y el anterior son conocidos como códigos de longitud fija, porque cada símbolo es representado con el mismo número de bits. Esto tiene algunas desventajas respecto a usar códigos de longitud variable, en el cual diferentes símbolos pueden ser representados con diferentes números de bits. En general, si nuestros mensajes son tal que algunos símbolos aparecen muy frecuentemente y algunos muy rara vez, podemos codificar datos más eficientemente. Una de las dificultades de usar códigos de longitud variable es conocer cuándo se ha alcanzado el fin de un símbolo en una lectura de una secuencia de unos y ceros. Una solución es diseñar el código de tal forma que el código asociado a un símbolo completo no sea el inicio (prefijo) del código asociado a otro símbolo. Tal código es llamado código prefijo. En general podemos usar código prefijo con longitud variable. Un esquema particular para hacer esto es llamado el método

de codificación Huffman, propuesto por David Huffman. Un código Huffman puede ser representado por un árbol binario donde las hojas son los símbolos a codificar.

1.1. El algoritmo de Huffman

Huffman propone que a partir de una lista de símbolos con una determinada frecuencia $SF = \{(S_1, F_1), \ldots (S_n, F_n)\}$ (F_i es el número esperado de veces que aparece el símbolo S_i en un cadena de 100 símbolos) puede obtener un árbol binario donde los símbolos están en las hojas, de tal forma que el código asociado a cada símbolo corresponde a la etiqueta del camino de ir desde la raíz hasta la hoja del símbolo, etiquetando 0 si escoge la rama izquierda o 1 si escoge la rama derecha.

Se construirá un árbol binario de abajo hacia arriba (desde las hojas hasta la raíz) tal que cada nodo contendrá una pareja (LS, F) donde $LS \subseteq \{S1, S2, ..., Sn\}$, es una lista de símbolos y F una frecuencia acumulada. Inicialmente hay tantas hojas como símbolos, cada una conteniendo la pareja $(\{S_i\}, F_i)$, y no hay ningún nodo interno creado. El procedimiento que se describe a continuación, considera como entrada una lista de nodos internos que hay que conectar en un árbol binario y conecta dos de ellos a través de un nodo interno. Estos dos nodos desaparecen de la lista de entrada, en la cual se incluye el nuevo nodo interno. La aplicación repetida de este procedimiento, termina construyendo el árbol requerido.

Considere una lista de nodos internos $LN = \{(LS_1, FA_1), (LS_2, FA_2), \ldots, (LS_k, FA_k)\}$, donde LS_i es una lista de símbolos, y FA_i su respectiva frecuencia acumulada.

Sean FA_i y FA_j las dos menores frecuencias en LN, con $i \neq j$. Se crea un nodo interno, como el padre de los nodos (LS_i, FA_i) y (LS_j, FA_j) y que contiene como pareja a $(LS_i \bigcup LS_j, FA_i + FA_j)$.

Por último, se remplazan en LN los nodos(LS_i, FA_i) y (LS_j, FA_j) por el nodo $(LS_i \bigcup LS_j, FA_i + FA_j)$, y se reitera el procedimiento, hasta que en LN quede un solo nodo interno, que de hecho, será el nodo raíz del árbol construido.

Ejemplo: considere inicialmente

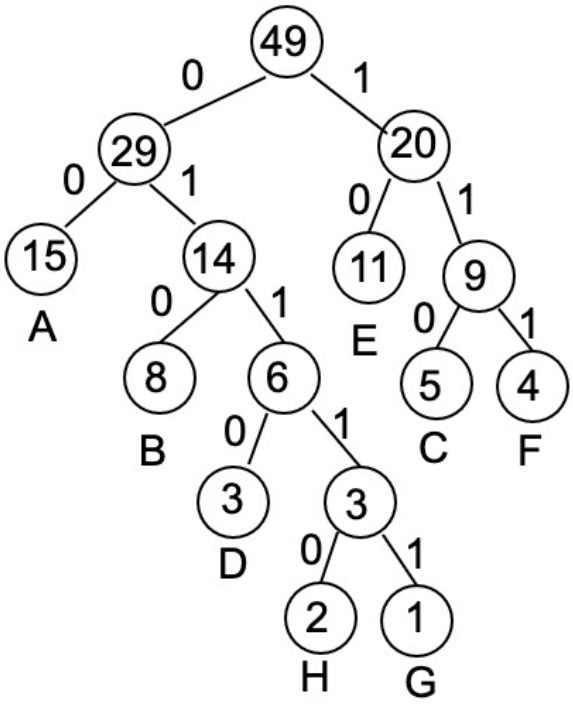
$$LN = \{(A, 15), (B, 8), (C, 5), (D, 3), (E, 11), (F, 4), (G, 1), (H, 2)\}.$$

La reiteración del procedimiento descrito da como resultados intermedios las siguientes listas de nodos internos:

$LN = \{(A, 15), (B, 8), (C, 5), (D, 3), (E, 11), (F, 4), (\{G, H\}, 3)\}$
$LN = \{(A, 15), (B, 8), (C, 5), (\{D, G, H\}, 6), (E, 11), (F, 4)\}$
$LN = \{(A, 15), (B, 8), (\{C, F\}, 9), (\{D, G, H\}, 6), (E, 11)\}$
$LN = \{(A, 15), (\{B, D, G, H\}14), (\{C, F\}, 9), (E, 11)\}$
$LN = \{(A, 15), (\{B, D, G, H\}, 14), (\{C, F, E\}, 20)\}$
$LN = \{(\{A, B, D, G, H\}, 29), (\{C, F, E\}, 20)\}$

$LN = \{(\{A, B, C, D, E, F, G, H\}, 49)\}$

Luego los códigos asignados serían (ver figura):



A 00, B 010, C 110, D 0110, E10, F 111, G 01111, H 01110

Así el código: 01000011000100000111011100110 corresponde al mensaje $BADAEADFAC$ (Separándolo por prefijos: 010 00 0110 00 10 00 0110 111 00 110 cuando se lee el código).

- ¿Por qué el código no es prefijo de ningún otro?.
- ¿Por qué tiene este árbol esa propiedad?.
- ¿A qué símbolos se le asignan los códigos más cortos?.

Definición del problema: El problema a resolver se puede enunciar de esta manera:

Entrada: Lista de símbolos y frecuencias L, un mensaje a codificar M o un código a decodificar C.

Salida: El código del mensaje M y su porcentaje de ahorro con respecto al código ASCII de 7 bits, o el mensaje del código C.

1.2. Implementación

En Scala vamos a representar un árbol de Huffman, así:

```
abstract class ArbolH
case class Nodo (izq: ArbolH, der: ArbolH,
                 cars: List[Char], peso: Int) extends ArbolH
case class Hoja(car: Char, peso: Int) extends ArbolH
```

Las hojas de árbol corresponden a un caracter y su frecuencia (peso). Los nodos internos además de contener sus hijos izquierdo y derecho, contienen la lista de caracteres que hay en sus hojas, y la suma de las frecuencias (peso) de esas hojas.

Para empezar, implemente las funciones `peso` y `cars` que dado un árbol de Huffman devuelven su peso y la lista de caracteres que codifica, respectivamente.

```
  def peso(arbol: ArbolH): Int = arbol match {
...
  }
  def cars(arbol: ArbolH): List[Char] = arbol match {
...
  }
```

Con estas funciones ya se puede escribir la función que dados dos subárboles de un árbol de Huffman, permite construir el árbol de Huffman correspondiente:

```
  def hacerNodoArbolH(izq: ArbolH, der: ArbolH) =
    Nodo(izq, der, cars(izq) ::: cars(der), peso(izq) + peso(der))
```

Esta función la puede usar donde la necesite.

1.2.1. Construyendo árboles de Huffman

Dado un texto, es posible calcular y construir un árbol de Huffman óptimo, en el sentido en que la codificación de ese texto será de longitud mínima, y sin pérdida de información (es decir, se puede decodificar y recuperar el texto original sin errores). El objetivo de esta sección es que usted escriba una función `crearArbolDeHuffman` que reciba un texto en forma de lista de caracteres y devuelva el árbol de Huffman asociado a ese texto.

Para ayudarlo con esta tarea, siga los siguientes pasos:

1. Escriba una función `ocurrencias`, que reciba un texto en forma de lista de caracteres y devuelva la lista con la frecuencia en que cada caracter aparece en el texto. Su función debe lucir así:

   ```
   def ocurrencias(cars: List[Char]): List[(Char, Int)] = {
     ...
   }
   ```

   Utilice las funciones auxiliares que considere necesarias.

2. Escriba una función `listaDeHojasOrdenadas`, que reciba una lista de frecuencias como la producida por la función anterior, y devuelva la lista de hojas del árbol de Huffman correspondiente, ordenada ascendentemente por la frecuencia de cada caracter. Su función debe lucir así:

   ```
   def listaDeHojasOrdenadas(frecs: List[(Char, Int)]): List[Hoja] = {
   ...
   }
   ```

   Utilice las funciones auxiliares que considere necesarias.

3. Escriba una función `listaUnitaria`, que reciba una lista de árboles de Huffman (sean hojas o nodos), y devuelva *true* si sólo hay un árbol en la lista, y $false$ en caso contrario. Su función debe lucir así:

```
def listaUnitaria(arboles: List[ArbolH]): Boolean = {
...
}
```

4. Escriba una función `combinar`, que reciba una lista de árboles de Huffman ordenada ascendentemente por el peso de cada árbol, tome los dos primeros (los de menor peso) si los hay, y devuelva una lista de árboles de Huffman ordenada ascendentemente con los mismos árboles originales, salvo los dos primeros. En lugar de ellos, en la lista de árboles de salida de la función debe aparecer el árbol de Huffman correspondiente a combinar esos dos árboles en uno solo, además del resto de árboles que no fueron usados en la combinación, y obviamente, la lista debe seguir estando ordenada. Su función debe lucir así:

```
def combinar(arboles: List[ArbolH]): List[ArbolH] = {
...
}
```

   Utilice las funciones auxiliares que considere necesarias.

5. Escriba una función *currificada* `hastaQue`, que reciba una pareja condición y acción ($cond : List[ArbolH] => Boolean, mezclar : List[ArbolH] => List[ArbolH]$), y luego una lista ordenada de árboles de Huffman, y devuelva una lista de árboles de Huffman correspondiente a aplicar la acción *mezclar* repetidamente sobre la lista original y sus resultados, hasta que la lista resultante cumpla la condición *cond*. Su función devuelve esta última lista. Su función debe lucir así:

```
  def hastaQue(cond: List[ArbolH]=>Boolean, mezclar:List[ArbolH]=>List[ArbolH] )
              (listaOrdenadaArboles: List[ArbolH]): List[ArbolH] = {

  ...
  }
```

6. Escriba la función `crearArbolDeHuffman`, mencionada antes, que reciba un texto en forma de lista de caracteres y devuelva el árbol de Huffman asociado a ese texto. Su función debe lucir así.

```
def crearArbolDeHuffman(cars: List[Char]): ArbolH = {
    ...
  }
```

   Esta función sólo tiene una línea de código consistente en invocar y utilizar como argumentos las funciones anteriores de la manera correcta.

1.2.2. Decodificando

Para efectos de este ejercicio, vamos a suponer que los Bits los representamos con un entero:

```
type Bit = Int
```

Decodificar consiste en recibir un mensaje codificado como una lista de Bits, y, usar el árbol de Huffman para encontrar el texto codificado.

Escriba una función `decodificar`, que reciba un árbol de Huffman y una lista de bits correspondiente a la codificación de un mensaje con ese árbol, y devuelva la lista de caracteres correspondiente al mensaje decodificado. Su función debe lucir así:

```
def decodificar(arbol: ArbolH, bits: List[Bit]): List[Char] = {
...
}
```

Utilice las funciones auxiliares que considere necesarias.

1.2.3. Codificando

Codificar consiste en recibir un mensaje (es decir una lista de caracteres), y, usar el árbol de Huffman para codificar el mensaje como una lista de Bits.

Usando el árbol de Huffman para codificar .

Escriba una función *currificada* `codificar`, que reciba un árbol de Huffman y luego una lista de caracteres correspondiente al mensaje a codificar con ese árbol, y devuelva la lista de bits correspondiente al mensaje codificado. Su función debe lucir así:

```
def codificar(arbol: ArbolH)(texto: List[Char]): List[Bit] = {
...
}
```

Utilice las funciones auxiliares que considere necesarias.

Al implementar este algoritmo notará que por cada caracter a codificar hay que recorrer el árbol desde la raíz hasta una hoja donde se encuentre el caracter a codificar, para saber los bits que se necesitan para codificarlo.

Esto es muy ineficiente.

Usando una tabla de códigos para codificar Para resolver esa ineficiencia, una alternativa es tener el código correspondiente a cada caracter en una tabla de códigos, y cada que se necesite codificar ese caracter, extraer su código de la tabla y no recorriendo el árbol de Huffman. Una manera sencilla de implementar esa tabla de códigos (no necesariamente la más eficiente) es con una lista de parejas $(car, lBits)$, donde car es un caracter y $lBits$ es la lista de Bits que lo codifica según el árbol.

```
type TablaCodigos = List[(Char, List[Bit])]
```

Para ayudarlo con esta tarea, siga los siguientes pasos:

1. Escriba una función *currificada* `codigoEnBits`, que reciba una tabla de códigos, y luego un caracter, y devuelva la lista de Bits correspondiente a ese caracter según la tabla. Su función debe lucir así:

```
def codigoEnBits(tabla: TablaCodigos)(car: Char): List[Bit] = {
...
}
```

2. Escriba una función `mezclarTablasDeCodigos`, que reciba dos tablas de códigos correspondientes a dos subárboles (izquierdo y derecho) de un árbol de Huffman, y devuelva la tabla de códigos correspondiente al árbol del que hacen parte. Su función debe lucir así:

```
def mezclarTablasDeCodigos(a: TablaCodigos, b: TablaCodigos): TablaCodigos ={
  ...
}
```

   Utilice las funciones auxiliares que considere necesarias.

3. Escriba una función `convertir`, que reciba un árbol de Huffman, y devuelva la tabla de códigos correspondiente a ese árbol. Use la función `mezclarTablasDeCodigos` para programar esta función. Su función debe lucir así:

```
def convertir(arbol: ArbolH): TablaCodigos = {
  ...
}
```

4. Escriba una función *currificada* `codificarRapido`, que reciba un árbol de Huffman y luego una lista de caracteres correspondiente al mensaje a codificar con ese árbol, y devuelva la lista de bits correspondiente al mensaje codificado, haciéndolo a través de una tabla de códigos para ser más eficientes. Su función debe lucir así:

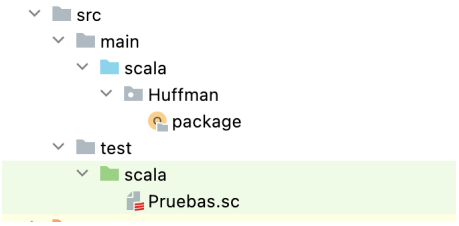
```
def codificarRapido(arbol: ArbolH)(texto: List[Char]): List[Bit] = {
    ...
}
```

   Utilice las funciones auxiliares que considere necesarias.

2. Entrega

2.1. Paquete *Huffman* y *worksheet* de pruebas

Usted deberá entregar dos archivos `package.scala` y `pruebas.sc` los cuales harán parte de su estructura de proyecto `IntellijIdea`, como se muestra en la figura a continuación:



Las funciones correspondientes a cada ejercicio, `peso`, `cars`, `ocurrencias`, `listaDeHojasOrdenadas`, `listaUnitaria`, `combinar`, `decodificar`, `codificar`, `codigoEnBits`, `mezclarTablasDeCodigos`, `convertir`, `codificarRapido`, `hastaQue` y `crearArbolDeHuffman` deben ser implementadas en un paquete de Scala denominado `Huffman`. **Todas, salvo las últimas dos deben usar reconocimiento de patrones esencialmente.** En ese paquete debe venir un archivo denominado `package.scala` que debe tener la forma siguiente:

```
package object Huffman {
  abstract class ArbolH
  case class Nodo (izq: ArbolH, der: ArbolH,
                   cars: List[Char], peso: Int) extends ArbolH
  case class Hoja(car: Char, peso: Int) extends ArbolH

  // Parte 1: Funciones esenciales y sencillas
  def peso(arbol: ArbolH): Int = arbol match { ...
  }

  def cars(arbol: ArbolH): List[Char] = arbol match { ...
  }

  def hacerNodoArbolH(izq: ArbolH, der: ArbolH) =
    Nodo(izq, der, cars(izq) ::: cars(der), peso(izq) + peso(der))

  // Parte 2: Construyendo arboles de Huffman

  /**
   * En este taller estamos trabajando con listas de caracteres.
   * La funcion cadenaAListA crea una lista de caracteres correspondiente  a una cadena dada
   */
  def cadenaAListA(cad: String): List[Char] = cad.toList

  def ocurrencias(cars: List[Char]): List[(Char, Int)] = { ...
  }

  def listaDeHojasOrdenadas(frecs: List[(Char, Int)]): List[Hoja] = { ...
  }

  def listaUnitaria(arboles: List[ArbolH]): Boolean = { ...
  }

  def combinar(arboles: List[ArbolH]): List[ArbolH] = { ...
  }

   def hastaQue(cond: List[ArbolH]=>Boolean, mezclar:List[ArbolH]=>List[ArbolH] )
              (listaOrdenadaArboles: List[ArbolH]): List[ArbolH] = { ...
  }

  def crearArbolDeHuffman(cars: List[Char]): ArbolH = { ...
  }

  // Part3 3: Decodificar

  type Bit = Int

  def decodificar(arbol: ArbolH, bits: List[Bit]): List[Char] = { ...
  }

  // Parte 4a: Codificando usando arboles de Huffman

  def codificar(arbol: ArbolH)(texto: List[Char]): List[Bit] = { ...
  }

  // Parte 4b: Codificando  usando tablas de codigos

  type TablaCodigos = List[(Char, List[Bit])]
```

```
  def codigoEnBits(tabla: TablaCodigos)(car: Char): List[Bit] = { ...
  }

  def mezclarTablasDeCodigos(a: TablaCodigos, b: TablaCodigos): TablaCodigos ={ ...
  }

  def convertir(arbol: ArbolH): TablaCodigos = { ...
  }

  def codificarRapido(arbol: ArbolH)(texto: List[Char]): List[Bit] = { ...
  }

}
```

Dicho paquete será usado en un *worksheet* de Scala con casos de prueba. Estos casos de prueba deben venir en un archivo denominado `pruebas.sc` . Un ejemplo de un tal archivo es el siguiente:

```
import Huffman._

val arbolEjemplo = hacerNodoArbolH(
  hacerNodoArbolH(Hoja('x', 1), Hoja('e', 1)),
  Hoja('t', 2)
)

val lc=cadenaALista("La vida es dura")
val lho=listaDeHojasOrdenadas(ocurrencias(lc))
listaUnitaria(lho)
crearArbolDeHuffman(lc)
```

Al ejecutarlo da lo siguiente:

```
import Huffman._

val arbolEjemplo: Huffman.Nodo = Nodo(Nodo(Hoja(x,1),Hoja(e,1),List(x, e),2),Hoja(t,2),List(x, e, t),4)




val lc: List[Char] = List(L, a,  , v, i, d, a,  , e, s,  , d, u, r, a)
val lho: List[Huffman.Hoja] = List(Hoja(r,1), Hoja(u,1), Hoja(s,1), Hoja(e,1), Hoja(i,1), Hoja(v,1),
                                   Hoja(L,1), Hoja(d,2), Hoja( ,3), Hoja(a,3))
val res0: Boolean = false
val res1: Huffman.ArbolH = Nodo(Nodo(Hoja(a,3),Nodo(Hoja(L,1),Hoja(d,2),List(L, d),3),List(a, L, d),6),
                                Nodo(Nodo(Nodo(Hoja(r,1),Hoja(u,1),List(r, u),2),
                                          Nodo(Hoja(s,1),Hoja(e,1),List(s, e),2),
                                     List(r, u, s, e),4),
                                     Nodo(Nodo(Hoja(i,1),Hoja(v,1),List(i, v),2),
                                          Hoja( ,3),
                                     List(i, v,  ),5),
                                List(r, u, s, e, i, v,  ),9),
                           List(a, L, d, r, u, s, e, i, v,  ),15)
```

Otro ejemplo de prueba es el siguiente:

```
val a='a'
val b='b'
val c='c'
val d='d'
val e= 'e'
val f='f'
val g='g'
val h='h'
val ah= crearArbolDeHuffman(List(a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,
  b,b,b,b,b,b,b,b,
  c,c,c,c,c,c,
  d,d,d,
  e,e,e,e,e,e,e,e,e,e,e,
  f,f,f,f,
  g,
  h,h
))
val cod_a = codificar(ah)(List(a))
val cod_b = codificar(ah)(List(b))
val cod_c = codificar(ah)(List(c))
```

```
val cod_d = codificar(ah)(List(d))
val cod_e = codificar(ah)(List(e))
val cod_f = codificar(ah)(List(f))
val cod_g = codificar(ah)(List(g))
val cod_h = codificar(ah)(List(h))

decodificar(ah, cod_a)
decodificar(ah, cod_b)
```

Al ejecutarlo da lo siguiente:

```
val a: Char = a
val b: Char = b
val c: Char = c
val d: Char = d
val e: Char = e
val f: Char = f
val g: Char = g
val h: Char = h
val ah: Huffman.ArbolH = Nodo(Nodo(Nodo(Hoja(f,4),Hoja(c,5),List(f, c),9),
                                   Hoja(e,11),
                              List(f, c, e),20),
                              Nodo(Nodo(Nodo(Hoja(d,3),Nodo(Hoja(g,1),Hoja(h,2),List(g, h),3),
                                        List(d, g, h),6),
                                        Hoja(b,8),
                                   List(d, g, h, b),14),
                              Hoja(a,15),
                              List(d, g, h, b, a),29),
                         List(f, c, e, d, g, h, b, a),49)

val cod_a: List[Huffman.Bit] = List(1, 1)
val cod_b: List[Huffman.Bit] = List(1, 0, 1)
val cod_c: List[Huffman.Bit] = List(0, 0, 1)
val cod_d: List[Huffman.Bit] = List(1, 0, 0, 0)
val cod_e: List[Huffman.Bit] = List(0, 1)
val cod_f: List[Huffman.Bit] = List(0, 0, 0)
val cod_g: List[Huffman.Bit] = List(1, 0, 0, 1, 0)
val cod_h: List[Huffman.Bit] = List(1, 0, 0, 1, 1)

val res5: List[Char] = List(a)
val res6: List[Char] = List(b)
```

2.2. Informe del taller - secciones

Todo taller debe venir acompañado de un informe en formato pdf. El informe debe contener la información explícita solicitada en el enunciado.

Para este caso, el informe del taller debe contener al menos tres secciones: informe de uso del reconocimiento de patrones, informe de corrección y conclusiones.

2.2.1. Informe de uso del reconocimiento de patrones

Tal como se ha visto en clase, el reconocimiento de patrones es una herramienta my poderosa y expresiva para programar. En esta sección usted debe hacer una tabla indicando en cuáles funciones usó la técnica y en cuáles no. Y en las que no, indicar por qué no lo hizo.

También se espera una apreciación corta de su parte, sobre el uso del reconocimiento de patrones como técnica de programación.

2.2.2. Informe de corrección

Es muy importante reflexionar sobre la corrección del código entregado. Para ello se deberá argumentar sobre la corrección de las funciones entregadas, y también deberá entregar un conjunto de pruebas. Todo esto lo consigna en esta sección del informe, dividida de la siguiente manera:

Argumentación sobre la corrección Para cada función, `peso`, `cars`, `ocurrencias`, `listaDeHojasOrdenadas`, `listaUnitaria`, `combinar`, `decodificar`, `codificar`, `codigoEnBits`, `mezclarTablasDeCodigos`, `convertir`, `codificarRapido`, `hastaQue` y `crearArbolDeHuffman`, argumente lo más formalmente posible por qué es correcta. Utilice inducción o inducción estructural donde lo vea pertinente. Estas argumentaciones las consigna en esta sección del informe.

Casos de prueba Para cada función se requieren mínimo 5 casos de prueba donde se conozca claramente el valor esperado, y se pueda evidenciar que el valor calculado por la función corresponde con el valor esperado en toda ocasión.

Una descripción de los casos de prueba diseñados, sus resultados y una argumentación de si cree o no que los casos de prueba son suficientes para confiar en la corrección de cada uno de sus programas, los registra en esta sección del informe. Obviamente, esta parte del informe debe ser coherente con el archivo de pruebas entregado, `pruebas.sc`.

2.3. Fecha y medio de entrega

Todo lo anterior, es decir los archivos `package.scala`, `pruebas.sc`, e `Informe del taller`, debe ser entregado vía el campus virtual, a más tardar a las 10 a.m. del jueves 9 de octubre de 2025, en un archivo comprimido que traiga estos tres elementos.

Cualquier indicación adicional será informada vía el foro del campus asociado a *programación funcional*.

3. Evaluación

Cada taller será evaluado tanto por el profesor del curso con ayuda del monitor, como por 2 compañeros que hayan entregado el taller. A este tipo de evaluación se le conoce como *Coevaluación*.

El objetivo de la coevaluación es lograr aprendizajes a través de:

- La lectura de lo que otros compañeros hicieron ante el mismo reto. Esto permite contrastar las soluciones propias con las de otros, y aprender de ellas, o compartir mejores maneras de hacer algo con otros.

- Retroalimentar a los compañeros que fueron asignados para evaluar. Al escribir la percepción que tenemos sobre el trabajo del otro, podemos aprender de cómo lo hicieron, y dar indicaciones al otro sobre otra formas de hacer lo mismo o, incluso, felicitarlo por la solución que presenta.
- La lectura de las retroalimentaciones de mis compañeros o del profesor/monitor.

La calificación de cada taller corresponderá entonces:

- en un 80 % a la calificación ponderada que reciba del profesor/monitor, vía una rúbrica de evaluación (pesa 5 veces lo que pesa la de otro estudiante) y de los tres compañeros asignados para evaluarlo.
- en un 20 % a la calificación que el sistema hará del trabajo de evaluación asignado. El Sistema tiene un método inteligente para estimar esa calificación, a partir de las evaluaciones realizadas por cada estudiante y por el profesor/monitor.