INFORME DEL TALLER 3

MAYRA ALEJANDRA SANCHEZ SALINAS (2040506)

FUNDAMENTOS DE PROGRAMACIÓN FUNCIONAL Y CONCURRENTE

JUAN FRANCISCO DIAZ FRIAS

UNIVERSIDAD DEL VALLE
SANTIAGO DE CALI, VALLE DEL CAUCA
2022

CONTENIDO

EL ALGORITMO HUFFMAN

El algoritmo de Huffman es un algoritmo para la construcción de códigos de Huffman, desarrollado por David A. Huffman en 1952 y descrito en A Method for the Construction of Minimum-Redundancy Codes.

Este algoritmo toma un alfabeto de n símbolos, junto con sus frecuencias de aparición asociadas, y produce un código de Huffman para ese alfabeto y esas frecuencias.

IMPLEMENTACIÓN

- Representación del árbol de Huffman

```
/**
 * Representación de árbol
 */
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 carácter 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.

```
/**
 * Dado un árbol de Huffman devuelven su peso y la lista de caracteres que codifica
 */
 def peso (arbol: ArbolH): Int = arbol match{
   case Nodo(_, _, _, w)  => w
   case Hoja(_, w)        => w
 }
def cars (arbol: ArbolH): List[Char] = arbol match {
  case Nodo(_, _, cs, _) => cs
  case Hoja(c, _)        => List(c)
}
```

Dados dos subárboles de un árbol de Huffman, permite construir el árbol de Huffman correspondiente.

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

Y esta es la función que se usará cuando se necesite.

CONSTRUYENDO ÁRBOLES DE HUFFMAN

- La función ***cadenaAlista*** crea una lista de caracteres correspondiente a una cadena dada. (Está en la parte de pruebas)

```
def cadenaAlista(cad: String): List[Char] = cad.toList
```

- La función ***ocurrencias*** recibe un texto en forma de lista de caracteres y devuelve la lista con la frecuencia en que cada carácter aparece en el texto.

```
def ocurrencias (cars: List[Char]): List[(Char,Int)] = {
  (Map[Char,Int]() /:/ cars){
    (m,c) => m + (c -> (m.get(c) map {_+1} getOrElse 1))
  } toList
}
```

- La función ***listaDeHojasOrdenadas*** recibe una lista de frecuencias como la producida por la anterior, y devuelve la lista de hojas del árbol de Huffman correspondiente, ordenada ascendentemente por la frecuencia de cada carácter

```
def listaDeHojasOrdenadas ( frecs : List [ ( Char , Int ) ] ) : List [ Hoja ] = {
  frecs map { a => Hoja(a._1, a._2) } sortBy { peso _ }
}
```

- La función ***listaUnitaria***, recibe una lista de árboles de Huffman (sean hojas o nodos), y devuelve true si solo hay un árbol en la lista, y false en caso contrario

```
def listaUnitaria ( arboles : List [ ArbolH ] ) : Boolean = {
  arboles.size == 1
}
```

- La función ***combinar***, recibe una lista de árboles de Huffman ordenada ascendentemente por el peso de cada árbol, toma los dos primeros (los de menor peso) si los hay, y devuelve 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.

```
def combinar (arboles: List[ArbolH]): List[ArbolH] = arboles match{
  case Nil |_ :: Nil => arboles
  case x :: y :: ts => hacerNodoArbolH(x, y) :: ts sortBy { peso _ }
}
```

- La función currificada ***hastaQue***, recibe una pareja condición y acción (cond : List[ArbolH] => Boolean, mezclar : List[ArbolH] => List[ArbolH]), y luego una lista ordenada de árboles de Huffman, y devuelve 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.

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

- La función ***crearArbolDeHuffman***, mencionada antes, recibe un texto en forma de lista de caracteres y devuelve el árbol de Huffman asociado a ese texto.

```
def crearArbolDeHuffman ( cars : List [ Char ] ) : ArbolH = {
  hastaQue(listaUnitaria, combinar)(((listaDeHojasOrdenadas _) compose ocurrencias)(cars)).head
}
```

DECODIFICANDO

- La función ***decodificar***, recibe un árbol de Huffman y una lista de bits correspondiente a la codificación de un mensaje con ese árbol, y devuelve la lista de caracteres correspondiente al mensaje decodificado.

```
def decodificar(arbol: ArbolH, bits: List[Bit]): List[Char] = {
  def decodificarOtro(arbol: ArbolH, bits: List[Bit]): (Char, List[Bit]) =
    (bits, arbol) match {
      case (xs, Hoja(c, _)) => (c, xs)
      case (Nil, _) => sys.error("Esta secuencia termina en medio del arbol")
      case (x :: xs, Nodo(l, r, _, _)) => decodificarOtro(if (x == 0) l else r, xs)
      case _ => sys.error("Esto puede suceder")
    }
  if (bits.isEmpty) Nil else {
    val (c, remainder) = decodificarOtro(arbol, bits)
    c :: decodificar(arbol, remainder)
  }
}
```

CODIFICANDO

Usando el árbol de huffman para codificar hay que recorrer el árbol desde la raiz hasta una hoja donde se encuentre el carácter a codificar, para saber los bits que se necesitan para modificarlo

Forma ineficiente:

- La función currificada ***codificar***, recibe un árbol de Huffman y luego una lista de caracteres correspondiente al mensaje a codificar con ese árbol, y devuelve la lista de bits correspondiente al mensaje codificado.

```
def codificar(arbol: ArbolH)(texto: List[Char]): List[Bit] = {
  def codificarChar(arbol: ArbolH, c: Char): List[Bit] = {
    require(cars(arbol).contains(c))
    arbol match {
      case Hoja(x, _)       => Nil
      case Nodo(l, r, _, _) =>
        if (cars(l).contains(c))
          0 :: codificarChar(l, c)
        else 1 :: codificarChar(r, c)
    }
  }
  def _codificar(arbol: ArbolH)(text: List[Char]): List[List[Bit]] =
    text match {
      case Nil     => Nil
      case c :: cs => codificarChar(arbol, c) :: _codificar(arbol)(cs)
    }
  _codificar(arbol)(texto).flatten
}
```

Usando una tabla de códigos para codificar y 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.

Forma eficiente:

- La función currificada ***codigoEnBits***, recibe una tabla de códigos, y luego un carácter, y devuelve la lista de Bits correspondiente a ese carácter según la tabla.

```
def codigoEnBits (tabla: TablaCodigos) (car: Char): List [Bit] = {
  tabla find { case (c, bs) => c == car } map { _._2 } getOrElse Nil
}
```

- La función ***mezclarTablasDeCodigos***, recibe dos tablas de códigos correspondientes a dos subárboles (izquierdo y derecho) de un árbol de Huffman, y devuelve la tabla de códigos correspondiente al árbol del que hacen parte.

```
def mezclarTablasDeCodigos(a: TablaCodigos, b: TablaCodigos): TablaCodigos = a ~~union~~ b
```

- La función ***convertir***, recibe un árbol de Huffman, y devuelve la tabla de códigos correspondiente a ese árbol.

```
def convertir (arbol: ArbolH): TablaCodigos = arbol match {
  case Hoja(car, peso) => List((car, Nil))
  case Nodo(izq, der, cars, peso) => mezclarTablasDeCodigos(izquierda(convertir(izq)), derecha(convertir(der)))
}
```

- La función currificada ***codificarRapido***, recibe un árbol de Huffman y luego una lista de caracteres correspondiente al mensaje a codificar con ese árbol,

y devuelve la lista de bits correspondiente al mensaje codificado, haciéndolo a través de una tabla de códigos para ser más eficientes.

```
def codificarRapido(arbol: ArbolH)(texto: List[Char]): List[Bit] = texto map codigoEnBits(convertir(arbol)) flatten
```

INFORME DE USO DEL RECONOCIMIENTO DE PATRONES

FUNCION	USO TÉCNICA
peso	✔
cars	✔
ocurrencias	✘
listaDeHojasOrdenadas	✘
listaUnitaria	✘
combinar	✔
decodificar	✔
codificar	✔
codigoEnBits	✘
mazclarTablasDeCodigos	✘
convertir	✔
codificarRapido	✘
hastaQue	✘
crearArbolDeHuffman	✘

La técnica del reconocimiento de patrones me parece interesante porque almacena los datos y gracias a eso se pueden crear grandes cosas como por ejemplo: reconocimiento de voz, reconocimiento de huellas dactilares, reconocimiento de música, reconocimiento de caras y entre otros.

Esto a mi consideración es un gran avance ya que con reconocimiento de patrones podemos crear más programas como los mencionados anteriormente.

Las funciones que son marcadas con que no se usó la técnica es debido a que, según entendí, el reconocimiento almacena los datos y de la única forma que lo hace es haciendo match y usando la clase case.

INFORME DE CORRECCIÓN

ARGUMENTACIÓN SOBRE LA CORRECCIÓN

FUNCIÓN	ARGUMENTACIÓN
peso y cars	Se debe ingresar un árbol y estas funciones simplemente devuelve el peso y la lista de caracteres que codifica
ocurrencias	Esta función genera las frecuencias con las que se repiten letras en una lista de caracteres
listaDeHojasOrdenadas	Esta función ordena la secuencia de los árboles de acuerdo a la frecuencia de sus caracteres
listaUnitaria	Esta función simplemente evalúa la lista de la cadena dada y verifica cuantos arboles contiene y si tiene solo 1 árbol el resultado será true
combinar	Esta función combina (así como la función de mezclar códigos) dos o más árboles
decodificar	Esta función decodifica un árbol, es decir que se debe ingresar el listado de bits y lo que hace la función es identificar el listado y entregar la cadena
codificar y codificarRapido	Estas dos funciones hacen lo mismo pero la forma en que está hecha la función es diferente, lo que hace la función de codificar se resume en el map de codificarRapido y lo que hace es entregar la lista en bits de la cadena
codigoEnBits	Esta función entrega el listado de bits de un carácter de cualquier árbol
mezclarTablasDeCodigos	Lo que hace esta función es unir los códigos generados de las listas
convertir	Esta función es la más importante ya que es muy necesaria para pasar la lista de caracteres a una lista de bits correspondiente a cada carácter
hastaQue	Esta función llama dos funciones anteriormente creadas y se evalúa en una lista de hojas ordenadas
crearArbolDeHuffman	Esta función contiene la función hasta que dentro de ella y es debido a que gracias a que se debe tener un tope para crear los árboles, es decir, hasta llegar a un solo árbol

CASOS DE PRUEBA

1. Estas son las cadenas de string que se usarán en todos los casos de prueba

```
val lc= cadenaALista ( cad = "La_vida_es_dura")
val lc2= cadenaALista ( cad = "Me_gusta_programar")
val lc3= cadenaALista ( cad = "Soy_mayra")
val lc4= cadenaALista ( cad = "Me_duele_la_espalda")
val lc5= cadenaALista ( cad = "H")
```

   Lo que hace esto es pasar de cadena a lita por lo que queda así:

```
val lc: List[Char] = List(L, a, _, v, i, d, a, _, e, s, _, d, u, r, a)
val lc2: List[Char] = List(M, e, _, g, u, s, t, a, _, p, r, o, g, r, a, m, a, r)
val lc3: List[Char] = List(S, o, y, _, m, a, y, r, a)
val lc4: List[Char] = List(M, e, _, d, u, e, l, e, _, l, a, _, e, s, p, a, l, d, a)
val lc5: List[Char] = List(H)
```

2. Lista de hojas ordenadas va de la mano con la función ocurrencias que es la que nos da la frecuencia en la que cada carácter aparece en el texto

```
val lho= listaDeHojasOrdenadas ( ocurrencias( lc ) )
val lho2=listaDeHojasOrdenadas ( ocurrencias( lc2 ) )
val lho3=listaDeHojasOrdenadas ( ocurrencias( lc3 ) )
val lho4=listaDeHojasOrdenadas ( ocurrencias( lc4 ) )
val lho5=listaDeHojasOrdenadas ( ocurrencias( lc5 ) )
```

   Lo que devuelve es la lista de las hojas ordenadas con la frecuencia mencionada anteriormente.

```
val lho: List[Huffman.Hoja] = List(Hoja(e,1), Hoja(s,1), Hoja(u,1), Hoja(i,1), Hoja(v,1), Hoja(L,1), Hoja(r,1), Hoja(d,2), Hoja(a,3), Hoja(_,3))
val lho2: List[Huffman.Hoja] = List(Hoja(e,1), Hoja(s,1), Hoja(t,1), Hoja(u,1), Hoja(p,1), Hoja(o,1), Hoja(m,1), Hoja(M,1), Hoja(g,2), Hoja(_,2), Hoja(a,3), Hoja(r,3))
val lho3: List[Huffman.Hoja] = List(Hoja(m,1), Hoja(_,1), Hoja(r,1), Hoja(o,1), Hoja(S,1), Hoja(y,2), Hoja(a,2))
val lho4: List[Huffman.Hoja] = List(Hoja(s,1), Hoja(u,1), Hoja(M,1), Hoja(p,1), Hoja(d,2), Hoja(a,3), Hoja(l,3), Hoja(_,3), Hoja(e,4))
val lho5: List[Huffman.Hoja] = List(Hoja(H,1))
```

3. Lo que hace la lista unitaria es verificar si solo hay un árbol en la lista

```
// Verificar si hay un solo arbol en la lista
listaUnitaria ( lho )
listaUnitaria ( lho2 )
listaUnitaria ( lho3 )
listaUnitaria ( lho4 )
listaUnitaria ( lho5 )
```

```
val res0: Boolean = false
val res1: Boolean = false
val res2: Boolean = false
val res3: Boolean = false
val res4: Boolean = true
```

Ya que según los resultados arrojados anteriormente podemos observar que la única con una letra es lho5 y por eso su respuesta es true

4. Esta función recibe la lista de los caracteres que se instalaron al principio de la prueba

```
// Crear arbol
val arbol1  = crearArbolDeHuffman ( lc )
val arbol2  = crearArbolDeHuffman ( lc2 )
val arbol3  = crearArbolDeHuffman ( lc3 )
val arbol4  = crearArbolDeHuffman ( lc4 )
val arbol5  = crearArbolDeHuffman ( lc5 )
```

val arbol1: Huffman.ArbolH = Nodo(Nodo(Hoja(a,3),Hoja(_,3),List(a, _),6),Nodo(Nodo(Nodo(Hoja(u,1),Hoja(i,1),List(u, i),2),Nodo(Hoja(e,1),Hoja(s,1),List(e, s),2),List(u, i, e, s),4),Nodo(Hoja(d,2),Nodo(Hoja(r,1),Nodo(Hoja(v,1),Hoja(L,1),List(v, L),2),List(r, v, L),3),List(d, r, v, L),5),List(u, i, e, s, d, r, v, L),9),List(a, _, u, i, e, s, d, r, v, L),15)

val arbol2: Huffman.ArbolH = Nodo(Nodo(Nodo(Hoja(g,2),Hoja(_,2),List(g, _),4),Nodo(Nodo(Hoja(t,1),Hoja(u,1),List(t, u),2),Nodo(Hoja(e,1),Hoja(s,1),List(e, s),2),List(t, u, e, s),4),List(g, _, t, u, e, s),8),Nodo(Nodo(Nodo(Hoja(m,1),Hoja(M,1),List(m, M),2),Nodo(Hoja(p,1),Hoja(o,1),List(p, o),2),List(m, M, p, o),4),Nodo(Hoja(a,3),Hoja(r,3),List(a, r),6),List(m, M, p, o, a, r),10),List(g, _, t, u, e, s, m, M, p, o, a, r),18)

val arbol3: Huffman.ArbolH = Nodo(Nodo(Nodo(Hoja(m,1),Hoja(_,1),List(m, _),2),Hoja(y,2),List(m, _, y),4),Nodo(Hoja(a,2),Nodo(Hoja(S,1),Nodo(Hoja(r,1),Hoja(o,1),List(r, o),2),List(S, r, o),3),List(a, S, r, o),5),List(m, _, y, a, S, r, o),9)

val arbol4: Huffman.ArbolH = Nodo(Nodo(Nodo(Nodo(Hoja(M,1),Hoja(p,1),List(M, p),2),Nodo(Hoja(s,1),Hoja(u,1),List(s, u),2),List(M, p, s, u),4),Hoja(e,4),List(M, p, s, u, e),8),Nodo(Nodo(Hoja(d,2),Hoja(a,3),List(d, a),5),Nodo(Hoja(l,3),Hoja(_,3),List(l, _),6),List(d, a, l, _),11),List(M, p, s, u, e, d, a, l, _),19)

val arbol5: Huffman.ArbolH = Hoja(H,1)

5. Ahora con esta función se obtiene la tabla de códigos de cada árbol

```
// Convertir
val codigo1 = convertir ( arbol1 )
val codigo2 = convertir ( arbol2 )
val codigo3 = convertir ( arbol3 )
val codigo4 = convertir ( arbol4 )
val codigo5 = convertir ( arbol5 )
```

val codigo1: Huffman.TablaCodigos = List((a,List(0, 0)), (_,List(0, 1)), (u,List(1, 0, 0, 0)), (i,List(1, 0, 0, 1)), (e,List(1, 0, 1, 0)), (s,List(1, 0, 1, 1)), (d,List(1, 1, 0)), (r,List(1, 1, 1, 0)), (v,List(1, 1, 1, 1, 0)), (L,List(1, 1, 1, 1, 1)))

val codigo2: Huffman.TablaCodigos = List((g,List(0, 0, 0)), (_,List(0, 0, 1)), (t,List(0, 1, 0, 0)), (u,List(0, 1, 0, 1)), (e,List(0, 1, 1, 0)), (s,List(0, 1, 1, 1)), (m,List(1, 0, 0, 0)), (M,List(1, 0, 0, 1)), (p,List(1, 0, 1, 0)), (o,List(1, 0, 1, 1)), (a,List(1, 1, 0)), (r,List(1, 1, 1)))

val codigo3: Huffman.TablaCodigos = List((m,List(0, 0, 0)), (_,List(0, 0, 1)), (y,List(0, 1)), (a,List(1, 0)), (S,List(1, 1, 0)), (r,List(1, 1, 1, 0)), (o,List(1, 1, 1, 1)))

val codigo4: Huffman.TablaCodigos = List((M,List(0, 0, 0, 0)), (p,List(0, 0, 0, 1)), (s,List(0, 0, 1, 0)), (u,List(0, 0, 1, 1)), (e,List(0, 1)), (d,List(1, 0, 0)), (a,List(1, 0, 1)), (l,List(1, 1, 0)), (_,List(1, 1, 1)))

val codigo5: Huffman.TablaCodigos = List((H,List()))

Por ejemplo el código de u en val codigo1 es: (1, 0, 0, 0) y así se puede ver que se le asigna bits a cada carácter de la lista de texto

6. Ahora usando los códigos obtenidos en la función anterior, lo que se hace en esta es mezclar los códigos.

```
// Mezclar tablas de codigos
mezclarTablasDeCodigos ( codigo1 , codigo2 )
mezclarTablasDeCodigos ( codigo2 , codigo3 )
mezclarTablasDeCodigos ( codigo3 , codigo4 )
mezclarTablasDeCodigos ( codigo4 , codigo5 )
mezclarTablasDeCodigos ( codigo5 , codigo1 )
```

Los resultados a las pruebas anteriores son las siguientes:

val res5: Huffman.TablaCodigos = List((a,List(0, 0)), (_,List(0, 1)), (u,List(1, 0, 0, 0)), (i,List(1, 0, 0, 1)), (e,List(1, 0, 1, 0)), (s,List(1, 0, 1, 1)), (d,List(1, 1, 0)), (r,List(1, 1, 1, 0)), (v,List(1, 1, 1, 1, 0)), (L,List(1, 1, 1, 1, 1)), (g,List(0, 0, 0)), (_,List(0, 0, 1)), (t,List(0, 1, 0, 0)), (u,List(0, 1, 0, 1)), (e,List(0, 1, 1, 0)), (s,List(0, 1, 1, 1)), (m,List(1, 0, 0, 0)), (M,List(1, 0, 0, 1)), (p,List(1, 0, 1, 0)), (o,List(1, 0, 1, 1)), (a,List(1, 1, 0)), (r,List(1, 1, 1)))

val res6: Huffman.TablaCodigos = List((g,List(0, 0, 0)), (_,List(0, 0, 1)), (t,List(0, 1, 0, 0)), (u,List(0, 1, 0, 1)), (e,List(0, 1, 1, 0)), (s,List(0, 1, 1, 1)), (m,List(1, 0, 0, 0)), (M,List(1, 0, 0, 1)),

(p,List(1, 0, 1, 0)), (o,List(1, 0, 1, 1)), (a,List(1, 1, 0)), (r,List(1, 1, 1)), (m,List(0, 0, 0)), (_,List(0, 0, 1)), (y,List(0, 1)), (a,List(1, 0)), (S,List(1, 1, 0)), (r,List(1, 1, 1, 0)), (o,List(1, 1, 1, 1)))

val res7: Huffman.TablaCodigos = List((m,List(0, 0, 0)), (_,List(0, 0, 1)), (y,List(0, 1)), (a,List(1, 0)), (S,List(1, 1, 0)), (r,List(1, 1, 1, 0)), (o,List(1, 1, 1, 1)), (M,List(0, 0, 0, 0)), (p,List(0, 0, 0, 1)), (s,List(0, 0, 1, 0)), (u,List(0, 0, 1, 1)), (e,List(0, 1)), (d,List(1, 0, 0)), (a,List(1, 0, 1)), (l,List(1, 1, 0)), (_,List(1, 1, 1)))

val res8: Huffman.TablaCodigos = List((M,List(0, 0, 0, 0)), (p,List(0, 0, 0, 1)), (s,List(0, 0, 1, 0)), (u,List(0, 0, 1, 1)), (e,List(0, 1)), (d,List(1, 0, 0)), (a,List(1, 0, 1)), (l,List(1, 1, 0)), (_,List(1, 1, 1)), (H,List()))

val res9: Huffman.TablaCodigos = List((H,List()), (a,List(0, 0)), (_,List(0, 1)), (u,List(1, 0, 0, 0)), (i,List(1, 0, 0, 1)), (e,List(1, 0, 1, 0)), (s,List(1, 0, 1, 1)), (d,List(1, 1, 0)), (r,List(1, 1, 1, 0)), (v,List(1, 1, 1, 1, 0)), (L,List(1, 1, 1, 1, 1)))

7. Lo que hace la función codificar y codificar rápido es evaluar el árbol dado y pasarlo a una lista de 0 y 1

```
// Codificar rapido
codificarRapido ( arbol1 ) ( lc )
codificarRapido ( arbol2 ) ( lc2 )
codificarRapido ( arbol3 ) ( lc3 )
codificarRapido ( arbol4 ) ( lc4 )
codificarRapido ( arbol5 ) ( lc5 )
```

La única diferencia entre estas dos funciones es que codificar rápido recibe el árbol entero, mientras que para la función codificar debemos ingresar la lista de caracteres del árbol

```
/*
 Codificar (Arroja el mismo resultado de la funcion codificarRapido
 */
val codificar1 = codificar(arbol1)(List('L','a','_','v','i','d','a','_','e','s','_','d','u','r','a'))
val codificar2 = codificar(arbol2)(List('M','e','_','g','u','s','t','a','_','p','r','o','g','r','a','m','a','r'))
val codificar3 = codificar(arbol3)(List('S','o','y','_','m','a','y','r','a'))
val codificar4 = codificar(arbol4)(List('M','e','_','d','u','e','l','e','_','l','a','_','e','s','p','a','l','d','a'))
val codificar5 = codificar(arbol5)(List('H'))
```

Y la función codificar también arroja el mismo resultado que codificar rápido:

val codificar1: List[Huffman.Bit] = List(1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0)

val codificar2: List[Huffman.Bit] = List(1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1)

val codificar3: List[Huffman.Bit] = List(1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0)

val codificar4: List[Huffman.Bit] = List(0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1)

val codificar5: List[Huffman.Bit] = List()

8. Con la función codificar bits se puede obtener la lista de bits que tiene 1 carácter y para eso recibe una tabla de códigos que se generan de los árboles.

```
/*
 Codigo en bits
 */
// Codigo 1 "La vida es dura"
codigoEnBits (codigo1)( car = 'd')

// Codigo 2 "Me gusta programar"
codigoEnBits (codigo2)( car = 'e')

// Codigo 3 "Soy mayra"
codigoEnBits (codigo3)( car = 'o')

// Codigo 4 "Me duele la espalda"
codigoEnBits (codigo4)( car = 'd')

// Codigo 5 "H"
codigoEnBits (codigo5)( car = 'H')
```

```
val res15: List[Huffman.Bit] = List(1, 1, 0)

val res16: List[Huffman.Bit] = List(0, 1, 1, 0)

val res17: List[Huffman.Bit] = List(1, 1, 1, 1)

val res18: List[Huffman.Bit] = List(1, 0, 0)

val res19: List[Huffman.Bit] = List()
```

Por ejemplo para la letra d los bits son: (1, 1, 0)

Para la letra e los bits son: (0, 1, 1, 0)

Para la letra o los bits son: (1, 1, 1, 1)

Para la letra d los bits son: (1, 0, 0)

Y para la letra H no se genero ningún bit y como se puede notar la letra d en el código 1 y en el código 4 son diferentes

9. En esta función sólo combine dos árboles, sin embargo, como la función reciben listas entonces en esa lista puedo poner de 1 a más árboles

```
// Combinar
combinar ( List(arbol1,arbol2))
combinar ( List(arbol2,arbol3))
combinar ( List(arbol3,arbol4))
combinar ( List(arbol4,arbol5))
combinar ( List(arbol5,arbol1))
```

La idea de la función combinar es unir los nodos y las hojas de cada árbol y es por eso que las respuestas a cada combinación es extremadamente larga.

val res20: List[Huffman.ArbolH] = List(Nodo(Nodo(Nodo(Hoja(a,3),Hoja(_,3),List(a,_),6),Nodo(Nodo(Nodo(Hoja(u,1),Hoja(i,1),List(u, i),2),Nodo(Hoja(e,1),Hoja(s,1),List(e, s),2),List(u, i, e, s),4),Nodo(Hoja(d,2),Nodo(Hoja(r,1),Nodo(Hoja(v,1),Hoja(L,1),List(v,L),2),List(r, v, L),3),List(d, r, v, L),5),List(u,i,e,s,d,r,v,L),9),List(a,_,u,i,e,s,d,r,v,L),15),Nodo(Nodo(Nodo(Hoja(g,2),Hoja(_,2),List(g,_),4),Nodo(Nodo(Hoja(t,1),Hoja(u,1),List(t, u),2),Nodo(Hoja(e,1),Hoja(s,1),List(e, s),2),List(t, u, e, s),4),List(g, _, t, u, e, s),8),Nodo(Nodo(Nodo(Hoja(m,1),Hoja(M,1),List(m, M),2),Nodo(Hoja(p,1),Hoja(o,1),List(p, o),2),List(m, M, p, o),4),Nodo(Hoja(a,3),Hoja(r,3),List(a, r),6),List(m, M, p, o, a, r),10),List(g, _, t, u, e, s, m, M, p, o, a, r),18),List(a, _, u, i, e, s, d, r...

valres21:List[Huffman.ArbolH]=List(Nodo(Nodo(Nodo(Nodo(Hoja(g,2),Hoja(_,2),List(g,_),4),Nodo(Nodo(Hoja(t,1),Hoja(u,1),List(t,u),2),Nodo(Hoja(e,1),Hoja(s,1),List(e,s),2),List(t,u,e,s),4),List(g,_,t,u,e,s),8),Nodo(Nodo(Nodo(Hoja(m,1),Hoja(M,1),List(m,M),2),Nodo(Hoja(p,1),Hoja(o,1),List(p,o),2),List(m,M,p,o),4),Nodo(Hoja(a,3),Hoja(r,3),List(a,r),6),List(m,M,p,o,a,r),10),List(g,_,t,u,e,s,m,M,p,o,a,r),18),Nodo(Nodo(Nodo(Hoja(m,1),Hoja(_,1),List(m,_),2),Hoja(y,2),List(m,_,y),4),Nodo(Hoja(a,2),Nodo(Hoja(S,1),Nodo(Hoja(r,1),Hoja(o,1),List(r,o),2),List(S, r, o),3),List(a, S, r, o),5),List(m, _, y, a, S, r, o),9),List(g, _, t, u, e, s, m, M, p, o, a, r, m, _, y, a, S, r, o),27))

val res22: List[Huffman.ArbolH] = List(Nodo(Nodo(Nodo(Nodo(Hoja(m,1),Hoja(_,1),List(m, _),2),Hoja(y,2),List(m, _, y),4),Nodo(Hoja(a,2),Nodo(Hoja(S,1),Nodo(Hoja(r,1),Hoja(o,1),List(r, o),2),List(S, r, o),3),List(a, S, r, o),5),List(m, _, y, a, S, r, o),9),Nodo(Nodo(Nodo(Nodo(Hoja(M,1),Hoja(p,1),List(M, p),2),Nodo(Hoja(s,1),Hoja(u,1),List(s, u),2),List(M, p, s, u),4),Hoja(e,4),List(M, p, s, u, e),8),Nodo(Nodo(Hoja(d,2),Hoja(a,3),List(d, a),5),Nodo(Hoja(l,3),Hoja(_,3),List(l, _),6),List(d, a, l, _),11),List(M, p, s, u, e, d, a, l, _),19),List(m, _, y, a, S, r, o, M, p, s, u, e, d, a, l, _),28))
val res23: List[Huffman.ArbolH] = List(Nodo(Nodo(Nodo(Nodo(Nodo(Hoja(M,1),Hoja(p,1),List(M, p),2),Nodo(Hoja(s,1),Hoja(u,1),List(s,u),2),List(M,p,s,u),4),Hoja(e,4),List(M,p,s,u,e),8),Nodo(Nodo(Hoja(d,2),Hoja(a,3),List(d, a),5),Nodo(Hoja(l,3),Hoja(_,3),List(l, _),6),List(d, a, l, _),11),List(M, p, s, u, e, d, a, l, _),19),Hoja(H,1),List(M, p, s, u, e, d, a, l, _, H),20))

val res24: List[Huffman.ArbolH] = List(Nodo(Hoja(H,1),Nodo(Nodo(Hoja(a,3),Hoja(_,3),List(a, _),6),Nodo(Nodo(Nodo(Hoja(u,1),Hoja(i,1),List(u, i),2),Nodo(Hoja(e,1),Hoja(s,1),List(e, s),2),List(u, i, e, s),4),Nodo(Hoja(d,2),Nodo(Hoja(r,1),Nodo(Hoja(v,1),Hoja(L,1),List(v, L),2),List(r, v, L),3),List(d, r, v, L),5),List(u, i, e, s, d, r, v, L),9),List(a, _, u, i, e, s, d, r, v, L),15),List(H, a, _, u, i, e, s, d, r, v, L),16))

10. Para la función decodificar se necesita ingresar el árbol creado al inicio de la prueba y la lista de bits que se obtuvieron de la función codificar

```
// Decodificar
decodificar( arbol1 , List(1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0))
```

```
decodificar( arbol2 , List(1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1,
1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1) )

decodificar( arbol3 , List(1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0) )

decodificar( arbol4 , List(0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1,
1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1) )

decodificar( arbol5 , List() )
```

El resultado de decodificar la lista de bits nos arrojará la cadena que dan los bits

```
val res25: List[Char] = List(L, a, _, v, i, d, a, _, e, s, _, d, u, r, a)
val res26: List[Char] = List(M, e, _, g, u, s, t, a, _, p, r, o, g, r, a, m, a, r)
val res27: List[Char] = List(S, o, y, _, m, a, y, r, a)
val res28: List[Char] = List(M, e, _, d, u, e, l, e, _, l, a, _, e, s, p, a, l, d, a)
val res29: List[Char] = List()
```

11. La función hasta que llama las funciones lista unitaria y combinar hasta que la lista contenga un solo árbol

```
// Hasta que
hastaQue(listaUnitaria, combinar)(lho)
hastaQue(listaUnitaria, combinar)(lho2)
hastaQue(listaUnitaria, combinar)(lho3)
hastaQue(listaUnitaria, combinar)(lho4)
hastaQue(listaUnitaria, combinar)(lho5)
```

val res30: List[Huffman.ArbolH] = List(Nodo(Nodo(Hoja(a,3),Hoja(_,3),List(a, _),6),Nodo(Nodo(Nodo(Hoja(u,1),Hoja(i,1),List(u, i),2),Nodo(Hoja(e,1),Hoja(s,1),List(e, s),2),List(u, i, e, s),4),Nodo(Hoja(d,2),Nodo(Hoja(r,1),Nodo(Hoja(v,1),Hoja(L,1),List(v, L),2),List(r, v, L),3),List(d, r, v, L),5),List(u, i, e, s, d, r, v, L),9),List(a, _, u, i, e, s, d, r, v, L),15))

val res31: List[Huffman.ArbolH] = List(Nodo(Nodo(Nodo(Hoja(g,2),Hoja(_,2),List(g, _),4),Nodo(Nodo(Hoja(t,1),Hoja(u,1),List(t, u),2),Nodo(Hoja(e,1),Hoja(s,1),List(e, s),2),List(t, u, e, s),4),List(g, _, t, u, e, s),8),Nodo(Nodo(Nodo(Hoja(m,1),Hoja(M,1),List(m, M),2),Nodo(Hoja(p,1),Hoja(o,1),List(p, o),2),List(m, M, p, o),4),Nodo(Hoja(a,3),Hoja(r,3),List(a, r),6),List(m, M, p, o, a, r),10),List(g, _, t, u, e, s, m, M, p, o, a, r),18))

val res32: List[Huffman.ArbolH] = List(Nodo(Nodo(Nodo(Hoja(m,1),Hoja(_,1),List(m, _),2),Hoja(y,2),List(m, _, y),4),Nodo(Hoja(a,2),Nodo(Hoja(S,1),Nodo(Hoja(r,1),Hoja(o,1),List(r, o),2),List(S, r, o),3),List(a, S, r, o),5),List(m, _, y, a, S, r, o),9))

val res33: List[Huffman.ArbolH] = List(Nodo(Nodo(Nodo(Nodo(Hoja(M,1),Hoja(p,1),List(M, p),2),Nodo(Hoja(s,1),Hoja(u,1),List(s, u),2),List(M, p, s, u),4),Hoja(e,4),List(M, p, s, u, e),8),Nodo(Nodo(Hoja(d,2),Hoja(a,3),List(d, a),5),Nodo(Hoja(l,3),Hoja(_,3),List(l, _),6),List(d, a, l, _),11),List(M, p, s, u, e, d, a, l, _),19))

```
val res34: List[Huffman.ArbolH] = List(Hoja(H,1))
```