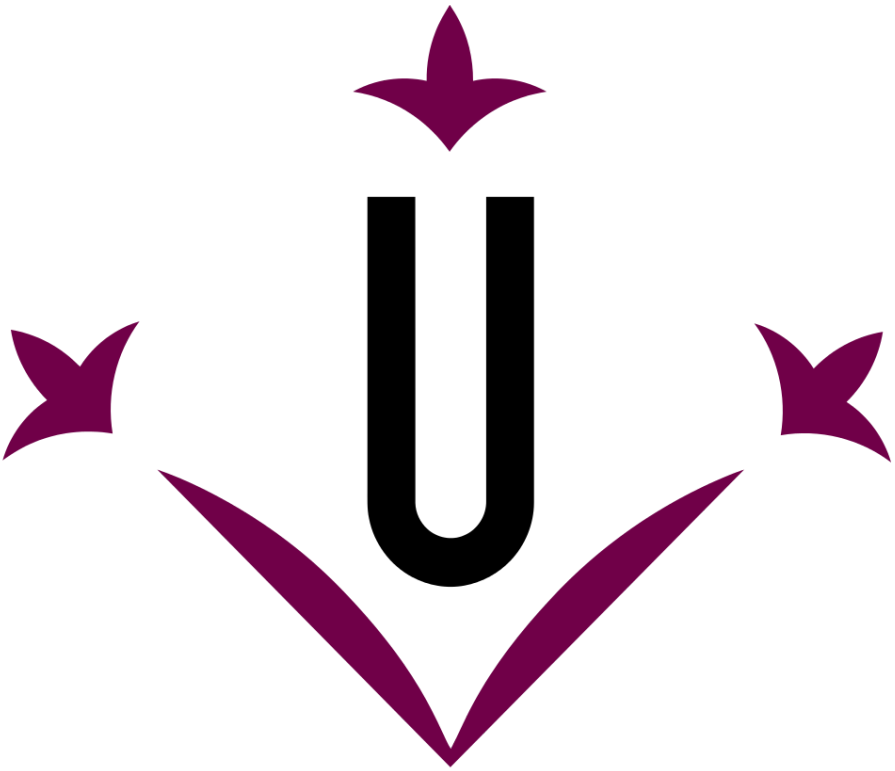
SCALA - ACTIVIDAD 1



Universitat de Lleida

Asignatura: Integración de Sistemas
Nombre: Sara Serrano Curdi
Fecha: 19/03/2023

Índice

1. Introducción

A lo largo de esta práctica veremos nuestra primera toma de contacto con la programación funcional. Nos centraremos en el desarrollo e implementación de diferentes funciones en el lenguaje Scala, aplicando distintos conceptos ya vistos en clase, como por ejemplo:

- Manejo de estructuras de datos, como listas y árboles
- Tipo *Option* como manejo de errores
- *foldLeft* y *foldRight*

Dividiremos el trabajo en 4 partes, cada una en sus respectivos ficheros y con sus funciones:

- List → partition, partitionMap, find, sort1, digitsToNum, mergeSorted, digitsToNumOption
- StdLib → countLengths, sumFirstPairsOf
- BST → insert, find, fold, fromList, inorder
- Hondt → hondt

Cabe destacar el uso de los ficheros *worksheet* para comprobar de forma dinámica y progresiva de los métodos implementados. Además, contamos también los respectivos ficheros de tests aplicados para comprobar cada una de las funciones mencionadas anteriormente.

2. List

En este apartado trataremos con la estructura de datos *List*. No la versión de la API de Scala, sino la que implementamos en clase en los ejercicios.

2.1 partition

```
def partition[A](l: List[A])(p: A => Boolean): (List[A], List[A]) =
    def decideList(elem: A, li: (List[A], List[A])): (List[A], List[A])
=
          if p(elem) then (Cons(elem, li._1), li._2) else (li._1,
Cons(elem, li._2))

     foldRight(l, (Nil:List[A], Nil: List[A]))( (h: A, acc: (List[A],
List[A])) => decideList(h, acc))
```

En partition, recibimos una lista de elementos y una función *p* con tal de, dependiendo de si cumplen los criterios de *p*, dividir la lista original en 2: la primera contendrá los elementos que sí cumplen y la segunda, aquellos que no. Para conseguirlo, hemos seguido la estrategia del *Type Tetris*, la cual veremos paso a paso cómo la hemos aplicado.

1. Sabemos que debemos llamar a *foldRight*
2. Por la implementación que tiene *foldRight*, sabemos que en la primera lista de parámetros debemos pasarle la lista sobre la que vamos a operar y el acumulador (el elemento base del cual partimos, e iremos modificando conforme avanzamos en la recursión). Al tratarse de listas, pasaremos una tupla de 2 listas vacías como acumulador.
3. En la segunda lista de parámetros, debemos pasar la función a aplicar a los elementos. En nuestro caso, nos interesa aplicar la separación de los elementos en función del resultado de *p*. Para ello, necesitamos una función de tipo (A, B) => B. Como no disponemos de ella inicialmente, la definimos dentro de *partition* y la llamaremos *decideList*.
4. Definimos *decideList* de la siguiente manera: sabemos que a cada vuelta de recursión tendremos disponibles el elemento de la lista y el acumulador. Por lo tanto, sabemos que (elemento: A, acumulador: (List[A], List[A]) => ???). Por la implementación de *foldRight*, sabemos que el tipo de retorno también debe ser el del acumulador, por lo que también devolveremos una tupla de 2 listas.
5. Una vez tenemos esto, solo hace falta aplicar la lógica que hemos mencionado anteriormente. Llamamos a la función *p*. Si el elemento ha cumplido los criterios, devolvemos la tupla con el elemento añadido a la lista izquierda y la lista derecha intacta. Para el caso en que no se cumplan los criterios de *p*, la tupla que retornaremos contendrá la lista izquierda intacta, y dicho elemento añadido en la lista derecha.

Como bien podemos observar, la función *decideList* podríamos haberla pasado de forma anónima a *foldRight*. Sin embargo, por ser el primer ejercicio y por falta de práctica, me ha sido más cómodo hacerlo por separado.

Una vez visto cómo se ha aplicado la técnica del *Type Tetris*, en los siguientes ejercicios obviaremos esta explicación y hablaremos sólo de la lógica que siguen las funciones.

2.2 partitionMap

```
def partitionMap[A, B, C](l: List[A])
      (p: A => Either[B, C]): (List[B], List[C]) =
     def decideListEither(elem: A, li: (List[B], List[C])):  (List[B],
List[C]) =
      p(elem) match
        case Left(value: B) => (Cons(value, li._1), li._2)
        case Right(value: C) => (li._1, Cons(value, li._2))

     foldRight(l, (Nil: List[B], Nil: List[C]))( (h: A, acc: (List[B],
List[C])) => decideListEither(h, acc))
```

En este caso, tenemos exactamente la misma lógica que el ejercicio anterior, pero usando el tipo *Either*. Sabemos que podemos encontrar 2 tipos de *Either*: un *Left* para indicar un error y un *Right* para indicar el éxito.

Sin embargo, tal y como podemos observar en la cabecera, el tipo *Either* nos indica que *Left* será de tipo B, y *Right* será de tipo C. Al mismo tiempo, el tipo de retorno nos indica que en la lista izquierda (donde deben estar los elementos “exitosos”) está el tipo B (el tipo de *Left*, cuando hay error), y viceversa para la lista derecha, y esto puede resultar contradictorio.
Por tanto, de cara a ajustarnos a los tipos, asumiremos que el contenido de *Left* serán los elementos exitosos, y en *Right* encontraremos los elementos erróneos.

Teniendo esto en mente, haremos lo mismo que en *partition*, con la siguiente diferencia al aplicar la función *p*:

- en caso de cumplir los criterios consideraremos que hay éxito, por lo que nos encontraremos ante el caso *Left* → añadiremos el elemento a la lista izquierda izquierda
- en caso de no cumplir los criterios, consideramos el elemento como erróneo, por lo que estamos ante el caso *Right* → añadiremos el elemento a la lista izquierda derecha

2.3 find

```
@annotation.tailrec
  def find[A](l: List[A])(p: A => Boolean): Option[A] =
    l match
      case Cons(h, t) => if p(h) then Some(h) else find(t)(p)
      case _ => None
```

Este ejercicio es más sencillo: recorremos la lista de forma recursiva hasta encontrar el primer elemento que cumpla una condición. Como se puede dar el caso de que ninguno la cumpla, retornamos un *Option*. Por tanto, si encontramos un elemento que satisfaga, debemos retornarlo (en forma de Some), y en caso contrario, un "error": None.

Además, agregamos la etiqueta `@annotation.tailrec` ya que hemos logrado la implementación *stack-safe* requerida (se retorna una llamada a *find*, sin elementos a "recordar" para la pila).

2.4 sort1

```
def sort1[A](as: List[A])(lte: (A, A) => Boolean): List[A] =
    def insertionSort(l: List[A], a: A): List[A] =
      l match
        case Cons(h, t) => if lte(a, h) then
            Cons(a, l) else
            Cons(h, insertionSort(t, a))
        case Nil => Cons(a, Nil)
    foldLeft(as, Nil)(insertionSort)
```

En este caso aplicamos el algoritmo de *Insertion sort* mediante *foldLeft*. La lógica del algoritmo es la siguiente:

- Iremos agregando números a una lista inicialmente vacía
- Al ser *foldLeft*, empezaremos a ordenar desde el final de la lista desordenada.

Supongamos el ejemplo List(3, 1, 2):

1. Aplicamos insertionSort con el 2 y el acumulador → Como el acumulador inicialmente está vacío, simplemente lo añadimos → Cons(2, Nil)
2. Aplicamos insertionSort con el 1 y el acumulador → El acumulador no está vacío, por lo que comprobamos dónde poner el 1.
    a. Como el 1 cumple los criterios de ordenación (lte), lo pondremos por delante → Cons(1, Cons(2, Nil))
3. Aplicamos insertionSort con el 3 y el acumulador → El acumulador no está vacío, por lo que comprobamos dónde poner el 3.
    a. Como el 3 cumple los criterios de ordenación (3 no es más pequeño que 1), seguimos comprobando con el resto del acumulador

b. 3 sigue sin ser más pequeño que el 2, por lo que continuamos comprobando el acumulador.
c. Finalmente llegamos al final del acumulador, por lo que simplemente lo añadimos → Cons(1, Cons(2, Cons(3, Nil)))

De esta manera, conseguimos ordenar la lista deseada.

2.5 digitsToNum

```
  def digitsToNum(l: List[Int]): Int =
        foldRight(l, "")((a: Int, acc: String) => (a.toString() + acc)).toInt
```

En este caso, debemos usar o bien *foldLeft* o *foldRight*. Si comparamos los dos podemos observar lo siguiente:

- *foldLeft* es *tail recursive*, tal y como habíamos visto en clase, podríamos pensar que debemos elegir esta opción sin dudar.
- *foldRight* es diferente a la versión vista en clase:
    - basa su implementación en *foldLeft*. Equivaldría al *foldRightViaFoldLeft.*
    - resulta ser es *tail recursive*
    - *foldLeft* y *foldRight* evalúan la lista desde izquierda y derecha respectivamente, por lo que para conseguir el mismo resultado, este *foldRight* necesita llamar a *reverse*.

Si analizamos esto, significa que ambas funciones son *tail recursive*, pero *foldRight* pasa 2 veces por la lista, mientras que *foldLeft* solo lo hace 1. De nuevo, *foldLeft* tiene las de ganar para ser la función elegida por parecer más eficiente.

No obstante, debemos recordar que *foldLeft* evalúa desde el final, por lo que al llamar a *digitsToNum* con la lista List(1, 2, 3, 4), obtendríamos el número 4321. Para conseguir el resultado esperado (1234), tendríamos que aplicar un *reverse*. De este modo, acabamos teniendo de todas formas 2 recorridos por la lista. Así que, puestos a recorrer 2 veces sí o sí, elegiremos *foldRight*. Obtenemos el resultado correcto sin tener que preocuparnos nosotros de darle la vuelta con *reverse*.

En cuanto a la función aplicada para pasar de dígito a número, es bastante sencilla. Partimos de una base donde no tenemos ningún número: lista vacía “” y los números que nos encontremos, los concatenamos en forma de *String.* Como debemos retornar un número, el resultado final lo pasamos a Int.

2.6 mergeSorted

```
def mergeSorted[A](l1: List[A], l2: List[A])(
      ord: (A, A) => Boolean
  ): List[A] =
    sort1(append(l1, l2))(ord)
```

En este caso, vamos a mezclar 2 listas ya ordenadas. Hemos optado por la versión sencilla: concatenamos las dos listas, y de la lista resultante, la ordenamos mediante *sort1*. Aplicando esta lógica, no nos afectan los casos en que alguna de las dos listas esté vacía, por lo que no necesitamos definirlos.

2.7 digitsToNumOption

```
  def digitsToNumOption(l: List[Int]): Option[Int] =
    def checkList(li: List[Int]): Option[Int] =
      li match
            case Cons(head, tail) => if head >= 0 && head <= 9 then
checkList(tail) else None
        case Nil => Some(digitsToNum(l))
    l match
      case Nil => None
      case Cons(head, tail) => checkList(l)
```

Aquí tenemos la versión de *digitsToNum* que considera la posibilidad de retornar un error mediante el tipo Option. Veamos cómo funciona:

- Si de entrada recibimos una lista vacía, no podemos pasarla a número, por lo que retornamos error (None).
- Si no está vacía, la pasaremos a números.

Ahora, empieza la evaluación de los números que recibamos. Los analizaremos con ayuda de la función auxiliar, ya que nos ayudará a distinguir entre recorrer la lista para analizar los dígitos y la lista original que mandaremos a *digitsToNum*.

La evaluación es sencilla: si todos los números se encuentran entre 0 y 9 se consideran dígitos válidos retornamos el resultado de *digitsToNum* dentro de un Some para satisfacer los tipos.

3. StdLib

En este apartado trataremos un par de ejercicios con estructuras originales de la API de Scala, tales como las mismas listas y Mapas.

3.1 countLengths

```
def countLengths(words: List[String]): Map[Int, Int] =
  def addPair(oldMap: Map[Int, Int], str: String): Map[Int, Int] =
    val len = str.length()
    if oldMap.exists(_._1 == len) then
          val currentValue = oldMap.apply(len)
          oldMap + (len -> (currentValue + 1))
    else oldMap + (len -> 1)

  words.foldLeft(Map[Int, Int]())(addPair)
```

Aquí recibiremos una lista de palabras, las cuales contaremos por cada una cuántas veces aparecen según su longitud, y lo retornaremos en forma de mapa. Para ello, seguiremos la siguiente lógica:

1. Aplicaremos nuestra función *addPair* mediante *foldLeft*. Como detalle a tener en cuenta, debemos llamar al *foldLeft* de las listas de *Scala*, no al que tenemos definido en el fichero *List*.
2. Para contar las veces que aparece cada palabra:
    a. Nos guardamos su longitud para hacer el mapeo en forma de (longitud → nº de apariciones)
    b. Si ya encontramos en el mapa (acumulador de *foldLeft*) dicha longitud, aumentamos en 1 en nº de apariciones
    c. En caso contrario, añadimos una nueva entrada al diccionario, indicando que lleva 1 aparición

3.2 sumFirstPairsOf

```
def sumFirstPowersOf(b: Int, n: Int): Int =
    def sumWithAcc(b: Int, n: Int, acc: Int): Int =
      if n - 1 >= 0 then
        val sum = acc + scala.math.pow(b, n-1).toInt
        sumWithAcc(b, n-1, sum)
        else acc
    sumWithAcc(b, n, 0)
```

Para sumar las primeras *n* potencias del número *b*, necesitamos un acumulador para ir guardando la suma de forma progresiva (ya que tenemos que hacer esta función *stack-safe*).

La suma se hará en forma de $b^{n-1} + b^{n-2} + b^{n-3} \ldots + b^0$. No sumaremos b^n ya que las n primeras potencias será desde 0 hasta n-1.

4. BST

En este fichero, trataremos funciones que manejan árboles BST.

4.1 insert

```
def insert[B >: A](a: B)(lt: (B, B) => Boolean): BST[B] =
    this match
      case Empty => Node(Empty, a, Empty)
      case Node(left, value, right) if lt(a, value) =>
            Node(left.insert(a)(lt), value, right)
      case Node(left, value, right) if lt(value, a) =>
            Node(left, value, right.insert(a)(lt))
      case _ => this
```

En esta función nos encargaremos de insertar elementos en un árbol de búsqueda binaria. Como dato a tener en cuenta (y que puede llevar a confusión), la definición de esta función se encuentra dentro del *companion object* de nuestro BST, por lo que para hacer referencia al objeto BST, podemos usar *this*.

Para saber cómo insertar un elemento, debemos recordar un poco de teoría sobre estos árboles: cuando nos encontramos en un nodo, los elementos de la rama izquierda son siempre más pequeños, y los de la rama derecha son siempre más grandes que el elemento actual. Por tanto, si queremos añadir un nuevo elemento, tendremos 4 posibles casos:

1. Nos encontramos en un nodo vacío → insertamos el elemento deseado en el nodo actual. Recordemos que en *Scala* los objetos son inmutables, por lo que no modificaremos el nodo, sino que retornamos uno nuevo siempre. Por tanto, retornamos una hoja.
2. El elemento a insertar es más pequeño que el actual → lo insertaremos por la izquierda
3. El valor actual es más pequeño que el elemento a insertar → lo insertaremos por la derecha
4. El único caso que queda es que el elemento actual sea igual que el que se desea insertar. Como en un BST no pueden haber elementos repetidos, retornamos el mismo árbol.

4.2 find

```
def find[B >: A](a: B)(lt: (B, B) => Boolean): Boolean =
    this match
      case Node(left, value, right) => if lt(a, value)
        then true
        else left.find(a)(lt) || right.find(a)(lt)
      case Empty => false
```

Para buscar si ya existe un elemento tendremos las siguientes posibilidades:

1. Si estamos en un nodo y el valor actual coincide con el que buscamos, lo hemos encontrado. En caso contrario, buscaremos por la rama izquierda y derecha.
    a. Al tratarse del operador binario OR, si la búsqueda de la izquierda resulta satisfactoria, no seguiremos buscando por la derecha, ya que la expresión ya evaluará a cierta.
2. Si llegamos a un nodo vacío, como es lógico, no lo hemos encontrado (de momento).

4.3 fold

```
def fold[B](b: B)(f: (B, A, B) => B): B =
    this match
      case Node(left, value, right) =>
            f(left.fold(b)(f), value, right.fold(b)(f))
      case Empty => b
```

Una vez entendemos cómo funcionan los *fold*s de las listas, veremos rápidamente cómo debería funcionar este.

- Igual que en las listas, tendremos el caso de haber llegado al final de un árbol (*Empty*), o de tener elementos por inspeccionar.
- Cuando nos encontramos en el final de la estructura a recorrer, sabemos que retornamos el acumulador
- Si estamos en un nodo, buscamos aplicar la función que nos pasen al resto del árbol.
    - Esto significa tener algo del estilo f(ramaIzquierda, valorActual, ramaDerecha).
        - Como la función la aplicaremos de forma recursiva, llamaremos a fold desde las ramas izquierda y derecha
        - Observamos que los tipos se cumplen: el valor actual es de tipo A, y las ramas son árboles (tipo B). Retornamos un tipo B, que será el árbol resultante de haber aplicado la función.

4.4 fromList

```
def fromList[A](l: List[A])(lt: (A, A) => Boolean): BST[A] =
    def toTree(l: List[A], tree: BST[A])(lt: (A, A) => Boolean): BST[A]
=
      l match
        case head :: next =>
          val newTree = tree.insert(head)(lt)
          toTree(next, newTree)(lt)
        case Nil => tree
    toTree(l, Empty)(lt)
```

Aquí nos encargaremos de pasar de una lista de elementos, a un BST que contenga dichos elementos, obviamente ordenados según el criterio de la función *lt*.

Aquí hay un detalle en el que me gustaría hacer hincapié. Lo primero que podríamos pensar es que para implementar esta función, vamos recorriendo la lista y a medida que nos encontramos elementos, los vamos insertando con *insert*, y tendremos un árbol ordenado como resultado.

Si nos pasaran la lista List(1, 2, 3, 4), obtendríamos un árbol como el de la figura 1.

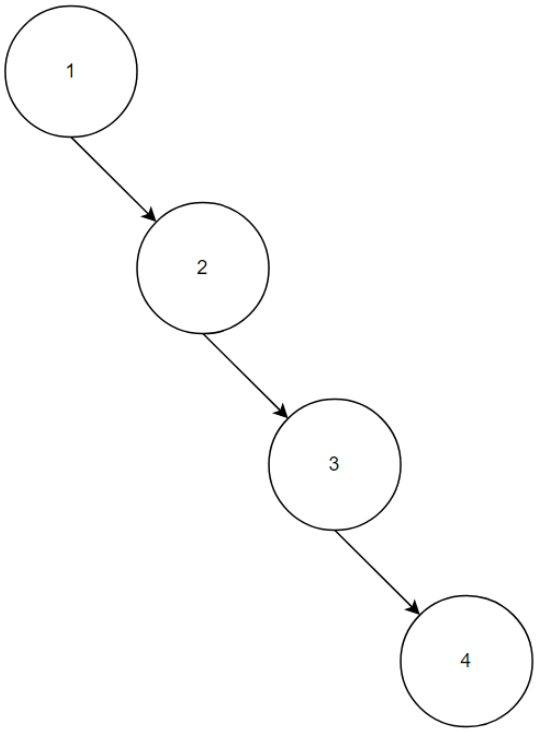


Figura 1: Árbol resultante (no completo)

Si recordamos la teoría vista previamente en la asignatura *Estructuras de datos*, este tipo de árbol se llama *no completo*, y tiene una limitación importante: dependiendo de la estructura usada para guardar el árbol, causaríamos un gran desperdicio del espacio.

No obstante, si consultamos la documentación de Scala, veremos que las listas son secuencias lineales, es decir, son *LinkedLists*. De modo que el tema del espacio no nos supone un problema ya que usamos listas. Si en su lugar tuviésemos un vector, sí que sería un problema a tener en cuenta.

Hecha esta aclaración, la lógica de la función será la siguiente: recorreremos la lista proporcionada, y a medida que encontremos elementos, se los pasaremos a la función *insert* e iremos guardando el árbol resultante en un acumulador, el cual retornaremos al llegar al final de la lista.

4.5 inorder

```
def inorderViaFold[A](t: BST[A]): List[A] =
    t.fold(Nil){
      (l: List[A], v: A, r: List[A]) => l ++ (v :: r)
    }
```

Antes de nada, debemos recordar cómo funciona el recorrido en inorden. Se trata de ir evaluando los valores del árbol primero por la rama izquierda, luego el valor actual, y finalmente por la rama derecha. Esto, aplicado a un BST, nos permite obtener los elementos de forma ordenada.

En cuanto a la implementación, tal y como se nos pide, llamaremos a *fold* con la siguiente función: concatenaremos **la lista** de los valores de la rama izquierda con la concatenación (mediante ++ en lugar de *Cons*) del valor actual y la lista derecha. De esta forma, respetamos el funcionamiento de inorden.

4.6 Tests

En los tests BSTSuite que podemos encontrar en el proyecto, hay uno que ha sido ligeramente modificado:

```
test("singleton") {
    val t = Node(Empty, 1, Empty)
    assertEquals(t.find(1)(_ < _), false)
  }
```

El valor original del assertEquals era `true`. Sin embargo, esto no es posible, ya que si solo tenemos el número 1, es imposible encontrar un valor que sea estrictamente más pequeño.

5. Hondt

Llegamos a la parte final de la práctica: implementar el sistema de Hondt para la repartición de escaños. Para ello, veamos un ejemplo: supongamos que tenemos los siguientes 3 partidos con sus respectivos votos: ((“A” → 50.000), (“B” → 30.000), (“C” → 10.000)) y 5 escaños. Realizaremos los siguientes pasos:

1. Mapeamos cada partido con las divisiones de sus votos

- “A” → (50.000, 25.000, 16.667, 12.500, 10.000)
- “B” → (30.000, 15.000, 10.000, 12.500, 10.000)
- “C” → (10.000, 5.000, 3.333, 2.500, 2.000)

2. Juntamos en una sola lista todas las divisiones de todos los votos

(50.000, 25.000, 16.667, 12.500, 10.000,
30.000, 15.000, 10.000, 12.500, 10.000,
10.000, 5.000, 3.333, 2.500, 2.000)

3. Ordenamos dichos votos en orden descendente

(50000, 30000, 25000, 16666, 15000, 12500, 10000, 10000, 10000, 7500, 6000, 5000, 3333, 2500, 2000)

4. Nos quedamos los primeros n valores (sea n el número de escaños)

50000, 30000, 25000, 16666, 15000

5. Buscamos a qué partidos pertenecen esos votos máximos

A, B, A, A, B, A

6. Mapeamos a cada partido con la suma de los escaños conseguidos

((“A” → 3), (“B” → 2))

```
def hondt(votes: Map[String, Int], n: Int): Map[String, Int] =
    val mapa_divisiones = votes.map(
      (partido: String, n_votos: Int) =>
      (partido, dividir(n_votos, n).reverse)
      )

    val todos_los_votos = mapa_divisiones.flatMap(
      (partido: String, l: List[Int]) => l
      ).toList

      val todos_los_votos_ordenados2 =
todos_los_votos.sorted(Ordering.Int.reverse)

    val max_votos2 = todos_los_votos_ordenados2.take(n)

    val lista_partidos_con_escaños: List[String] = buscar_partido(
List[String](), max_votos2, mapa_divisiones
      )

    agrupar_escaños(lista_partidos_con_escaños, Map[String, Int]())
```

Para comprobar de forma progresiva su correcto funcionamiento, se ha usado el *worksheet* correspondiente.

6. Conclusiones

Después de haber realizado esta práctica, las conclusiones más decisivas que puedo destacar son las siguientes:

- La programación funcional es realmente difícil si no estás acostumbrado a trabajar con funciones ni recursividad. Incluso partiendo de la base de que entiendas cómo funciona, pensar en diseñar todo en base a recursividad no siempre sale a la primera.
- Scala tiene una sintaxis realmente amigable. Cuando te sientes atascado es muy fácil orientarte gracias a los tipos tan marcados que hay y al lenguaje tan sencillo de leer.
- En consecuencia del punto anterior, la técnica del *Type Tetris* me ha salvado en muchas ocasiones cuando no sabía cómo continuar.
- El código resultante es realmente agradable en comparación a cómo quedaría con la programación imperativa.

Dicho esto, ha resultado una práctica bastante agradable, en la que se nota la curva de dificultad del lenguaje, pero una vez superada, se puede seguir de forma progresiva sin muchos problemas. Es más, una vez te deshaces de los pensamientos de “cómo lo resolverías de forma imperativa”, es más sencillo diseñar funciones más complejas como la de Hondt.