Taller 2: Funciones de alto orden

Juan Francisco Díaz Frias Julián Puyo

Marzo 2024

1. Ejercicio de programación: Comparador de algoritmos de ordenamiento

Un proceso recurrente en programación consiste en ordenar un conjunto de datos de acuerdo a una relación de orden entre los elementos de ese conjunto.

Hay muchos algoritmos de ordenamiento, cuya base es la comparación de un elemento del conjunto con otro. Esta es también la operación más costosa dentro de esos algoritmos. ¿Cómo saber cuál es mejor? Se dice que un algoritmo a_1 es mejor que otro algoritmo a_2 para ordenar un conjunto, si el número de comparaciones que hace el algoritmo a_1 es menor que el número de comparaciones que hace el algoritmo a_2.

Para efectos de este taller, nuestros conjuntos a ordenar estarán organizados en listas y los datos de los conjuntos serán parte de un conjunto ordenado (pueden ser enteros, cadenas, flotantes, o cualquier tipo de datos que tenga un orden asociado). Lo que pretendemos es construir una función que permita comparar dos algoritmos diferentes de ordenamiento sobre una entrada específica, y saber cuál es mejor en términos del número de comparaciones realizadas.

Dada una secuencia de valores (puede haber repetidos) del mismo tipo T, $< x_1, x_2, \ldots, x_n >$ no necesariamente ordenada, queremos:

- Ordenarla, es decir construir la secuencia $< y_1, y_2, \ldots, y_n >$ tal que $\forall i \in 1..(n-1)$: $y_i \leq y_{i+1}$ y $\{x_1, x_2, \ldots, x_n\} = \{y_1, y_2, \ldots, y_n\}$.

Para ello vamos a representar una secuencia de elementos con una lista de Scala.

Una lista en Scala es el resultado de invocar al constructor *List* con sus elementos. Por ejemplo $List(10, 8, 5, 9, 7)$ representa la secuencia de 5 enteros $< 10, 8, 5, 9, 7 >$.

Para este ejercicio utilizaremos solamente las siguientes funciones (métodos) para manipular listas de elementos de un tipo T ($List[T]$):

- l.isEmpty: Boolean (devuelve si una lista l está vacía)
- l.head: Int (devuelve el primer elemento de la lista l)
- l.tail: List[Int] (devuelve la lista sin el primer elemento l)
- $x :: l$ devuelve la lista que representa la secuencia $< x, x_1, x_2, \ldots, x_n >$ si l es la lista que representa la secuencia $< x_1, x_2, \ldots, x_n >$
- $l1 + +l2$ devuelve la lista que representa la concatenación de las secuencias representadas por $l1$ y $l2$

El objetivo de este taller es volverse competentes en desarrollar funciones de alto orden. Vamos a usarlas para representar los *Algoritmos de Ordenamiento* y construir los generadores de algoritmos conocidos como el *insertionSort* y una versión sencilla del *quickSort*, y un comparador de algoritmos.

Comenzaremos por representar cada *Algoritmo de Ordenamiento* como un valor del tipo $List[T] => (List[T], Int)$ es decir como una función que recibe una lista de elementos de tipo T de entrada y produce una pareja de salida compuesta por la lista de elementos de salida (la lista ordenada según un orden por definir) y el número de comparaciones realizadas para encontrar la respuesta. Y también representaremos un comparador de elementos del tipo T como un valor del tipo $(T, T) => Boolean$.

```
type algoritmoOrd[T] = List[T] => (List[T], Int)
type Comparador[T] = (T,T)=>Boolean
```

1.1. El algoritmo de ordenamiento por inserciones: *insertionSort*

El algoritmo de ordenamiento por inserciones se basa en una idea muy sencilla que da lugar a un algoritmo recursivo. Si la lista es de la forma $< x_1, x_2, \ldots, x_n >$:

- Ordenar por el mismo método $< x_2, \ldots, x_n >$. LLamemos $< y_2, \ldots, y_n >$ la lista ordenada resultante.
- Insertar x en su orden, según el comparador que se esté utilizando, en la lista ordenada $< y_2, \ldots, y_n >$ y devolver esta nueva lista.

1.1.1. Implementar la función de inserción

Implemente la función `insert` que reciba un elemento e de tipo T, una lista ordenada l de elementos de tipo T y un comparador de elementos del tipo T, *comp*, y devuelva una pareja de tipo $(List[T], Int)$ tal que el primer elemento de la pareja corresponde a la lista ordenada, según *comp*, que contiene los elementos de l y e.

```
def insert(e:T, l:List[T], comp:Comparador[T]): (List[T], Int) = {
      // Recibe un elemento e de tipo T a insertar en una lista ordenada l de elementos de tipo T
      // Y un comparador comp de elementos del tipo T
      // y devuelve, en una pareja la lista ordenada incluyendo el elemento e
      // y cuantas comparaciones se hicieron para lograrlo
...
    }
```

1.1.2. Implementar la función generadora de instancias del *insertionSort*

Implemente la función `insertionSort` que recibe un comparador de elementos del tipo T, *comp*, y devuelve el algoritmo de ordenamiento correspondiente al *insertionSort*.

```
def insertionSort[T](comp:Comparador[T]):AlgoritmoOrd[T] = {
    // Recibe una lista de elementos de tipo T y un comparador de esos elementos
    // y devuelve una funcion que corres'ponde al  InsertionSort
...
  }
```

La idea es poder usar esta función de la siguiente manera:

```
def menorQue(a:Int, b:Int): Boolean = a < b

val iSort_Asc = insertionSort[Int](menorQue)

iSort_Asc(List(4,5,6,1,2,3))
```

La cual dará como respuesta:

```
val res0: (List[Int], Int) = (List(1, 2, 3, 4, 5, 6),13)
```

indicando que ordenar la lista $List(4, 5, 6, 1, 2, 3)$ usando *insertionSort* con el comparador *menorQue* da como resultado la lista $List(1, 2, 3, 4, 5, 6)$ realizando 13 comparaciones.

1.2. El algoritmo de ordenamiento rápido: *quickSort*

El algoritmo de ordenamiento rápido se basa en otra idea sencilla: tomar la lista de entrada l y partirla en dos sublistas disyuntas l_1 y l_2 de l tales que:

- los elementos de l_1 unidos a los de l_2 son los elementos de la lista l, y
- todos los elementos de la lista l_1 sean menores, según el comparador utilizado, que todos los elementos de la lista l_2.

Típicamente, esta partición de l en l_1 y l_2 se logra escogiendo un valor de l como pivote, y dejando en l_1 los menores que el pivote según el comparador, y en l_2 los que no son menores que el pivote. En el *quickSort* tradicional, ese pivote se toma al azar. En este ejercicio, se tomará el primer elemento de la lista l.

Luego de tener l_1 y l_2, se ordenan con el mismo algoritmo esas dos sublistas (llamados recursivos) para luego concatenar las listas resultantes y obtener la lista deseada ordenada.

1.2.1. Implementar la función de partición de una lista en dos

Implemente la función `menoresQue_noMenoresQue` que recibe una lista de elementos de tipo T, l, un pivote v de tipo T, y un comparador de elementos de T, y devuelve una tripleta cuyo primer elemento es la lista l_1, cuyo segundo elemento es la lista l_2 (l_1, l_2 son la partición de l según el pivote v) y cuyo tercer elemento es el número de comparaciones realizadas para hacer esa partición.

```
def menoresQue_noMenoresQue(l:List[T], v:T, comp:Comparador[T]): (List[T],List[T],Int) = {
      // Recibe la una lista de elementos de tipo T  y un valor v de tipo T y un comparador de elementos de T
      // y devuelve la lista de elementos de l que son menores  que v,
      // la lista de los que no son menores que v
      // y cuantas comparaciones se hicieron para llegar a ella
...
    }
```

1.2.2. Implementar la función generadora de instancias del *quickSort*

Implemente la función `quickSort` que recibe un comparador de elementos del tipo T, *comp*, y devuelve el algoritmo de ordenamiento correspondiente al *quickSort*.

```
def quickSort[T](comp:Comparador[T]):AlgoritmoOrd[T] = {
    // Recibe una lista de elementos de tipo T y un comparador de esos elementos
    // y devuelve una funcion que corresponde al  quickSort
...
  }
```

La idea es poder usar esta función de la siguiente manera:

```
def menorQue(a:Int, b:Int): Boolean = a < b

val qSort_Asc = quickSort[Int](menorQue)

qSort_Asc(List(4,5,6,1,2,3))
```

La cual dará como respuesta:

```
val res1: (List[Int], Int) = (List(1, 2, 3, 4, 5, 6),16)
```

indicando que ordenar la lista $List(4, 5, 6, 1, 2, 3)$ usando *insertionSort* con el comparador *menorQue* da como resultado la lista $List(1, 2, 3, 4, 5, 6)$ realizando 16 comparaciones.

1.3. Implementando el comparador de algoritmos de ordenamiento por comparaciones

Implemente la función `comparar` que reciba dos algoritmos de ordenamiento de elementos de tipo T, a_1 y a_2, y una lista de elementos de T, l, y devuelva una pareja (c_1, c_2) tal que c_1 es el número de comparaciones realizadas por a_1 para ordenar l y c_2 es el número de comparaciones realizadas por a_2 para ordenar l.

En el caso en que el resultado de ordenar l con a_1 sea diferente al de ordenarla con a_2 (evidencia de que los dos algoritmos que estamos comparando, no hacen lo mismo) se debe devolver la pareja $(-1, -1)$.

```
  def comparar[T](a1:AlgoritmoOrd[T], a2:AlgoritmoOrd[T], l:List[T]):(Int,Int) = {
    // Recibe dos algoritmos de ordenamiento y una lista para ordenar
    // y devuelve una pareja con el numero de comparaciones hechas por a1,
    // y el numero de comparaciones hechas por a2 para esa instancia de l en particular
    // si los dos algoritmos dan el mismo resultado
    // sino, devuelve (-1,-1)
...
  }
```

Un ejemplo del uso de esta función es:

```
comparar(iSort_Asc,qSort_Asc,List(4,5,6,1,2,3))
comparar(iSort_Asc,qSort_Desc,List(4,5,6,1,2,3))
```

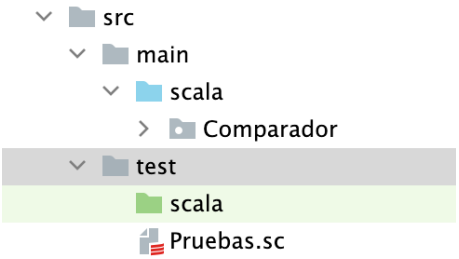
La cual dará como respuesta:

```
val res2: (Int, Int) = (13,16)
val res3: (Int, Int) = (-1,-1)
```

2. Entrega

2.1. Paquete *Comparador* 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, `insert`, `insertionSort`, `menoresQue_noMenoresQue`, `quickSort`, y `comparar`, deben ser implementadas en un paquete de Scala denominado `Comparador`. En ese paquete debe venir un archivo denominado `package.scala` que debe tener la forma siguiente:

```
package object Comparador {
  type AlgoritmoOrd[T] = List[T] => (List[T], Int)
  type Comparador[T] = (T,T)=>Boolean

  def insert[T](e:T, l:List[T], comp:Comparador[T]): (List[T], Int) = {
    // Recibe un elemento e de tipo T a insertar en una lista ordenada l de elementos de tipo T
    // y devuelve, en una pareja la lista ordenada incluyendo el elemento e
    // y cuantas comparaciones se hicieron para lograrlo
...
  }

  def insertionSort[T](comp:Comparador[T]):AlgoritmoOrd[T] = {
    // Recibe una lista de elementos de tipo T y un comparador de esos elementos
    // y devuelve la lista ordenada y el numero de comparaciones realizadas en una pareja
    // usando el InsertionSort
```

```
...
  }

  def menoresQue_noMenoresQue[T](l:List[T], v:T, comp:Comparador[T]): (List[T],List[T],Int) = {
    // Recibe la una lista de elementos de tipo T  y un valor v de tipo T
    // devuelve la lista de elementos de l que son menores  que v,
    // la lista de los que no son menores que v
    // y cuantas comparaciones se hicieron para llegar a ella
...
  }


  def quickSort[T](comp:Comparador[T]):AlgoritmoOrd[T] = {
    // Recibe una lista de elementos de tipo T y un comparador de esos elementos
    // y devuelve la lista ordenada y el numero de comparaciones realizadas en una pareja
    // Usando el quickSort
...
  }

  def comparar[T](a1:AlgoritmoOrd[T], a2:AlgoritmoOrd[T], l:List[T]):(Int,Int) = {
    // Recibe dos algoritmos de ordenamiento y una lista para ordenar
    // y devuelve una pareja con el numero de comparaciones hechas por a1,
    // y el numero de comparaciones hechas por a2 para esa instancia de l en particular
    // si los dos algoritmos dan el mismo resultado
    // sino, devuelve (-1,-1)
    val (l1,c1) = a1(l)
    val (l2,c2) = a2(l)
    if (l1==l2) (c1,c2) else (-1,-1)
  }
}
```

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 Comparador._
import scala.util.Random

val random = new Random()

def listaAlAzar(long:Int): List[Int] = {
  //Crea una lista de long enteros,
  // con valores aleatorios entre 1 y long*2
  val v = Vector.fill(long){
    random.nextInt(long*2)+1
    }
  v.toList
}

def menorQue(a:Int, b:Int): Boolean = a < b
def mayorQue(a:Int, b:Int): Boolean = a > b

val iSort_Asc = insertionSort[Int](menorQue)
val iSort_Desc = insertionSort[Int](mayorQue)
iSort_Asc(List(4,5,6,1,2,3))


val qSort_Asc = quickSort[Int](menorQue)
val qSort_Desc = quickSort[Int](mayorQue)
qSort_Asc(List(4,5,6,1,2,3))

comparar(iSort_Asc,qSort_Asc,List(4,5,6,1,2,3))
comparar(iSort_Asc,qSort_Desc,List(4,5,6,1,2,3))

val lAsc100 = (1 to 100).toList
val lAsc1000 = (1 to 1000).toList
val lDsc100 = (1 to 100).toList.reverse
val lDsc1000 = (1 to 1000).toList.reverse
comparar(iSort_Asc,qSort_Asc,lAsc100)
comparar(iSort_Asc,qSort_Asc,lAsc1000)
comparar(iSort_Asc,qSort_Asc,lDsc100)
comparar(iSort_Asc,qSort_Asc,lDsc100)




val l5=listaAlAzar(5)
val l10=listaAlAzar(10)
val l20=listaAlAzar(20)
val l50=listaAlAzar(50)
```

```
iSort_Asc(l5)
iSort_Desc(l5)
iSort_Asc(l10)
iSort_Desc(l10)
iSort_Asc(l20)
iSort_Desc(l20)
iSort_Asc(l50)
iSort_Desc(l50)

qSort_Asc(l5)
qSort_Desc(l5)
qSort_Asc(l10)
qSort_Desc(l10)
qSort_Asc(l20)
qSort_Desc(l20)
qSort_Asc(l50)
qSort_Desc(l50)

comparar(iSort_Asc,qSort_Asc,l5)
comparar(iSort_Asc,qSort_Asc,l10)
comparar(iSort_Asc,qSort_Asc,l20)
comparar(iSort_Asc,qSort_Asc,l50)
```

Nótese que se ha usado un paquete para generar listas al azar de diferentes longitudes, para poder hacer pruebas con listas de tamaños grandes.

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 funciones de alto orden, informe de corrección y conclusiones.

2.2.1. Informe de funciones de alto orden

Tal como se ha visto en clase, usar funciones de alto orden significa usarlas como parámetro, de forma anónima o como respuesta. Indique en una tabla, para cada función realizada, cuáles de estas tres formas de uso se utilizaron.

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 los programas entregados, 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,`insert`, `insertionSort`, `menoresQue_noMenoresQue`, `quickSort`, y `comparar`, 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 21 de marzo de 2024, 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.