Taller 4: Colecciones y Expresiones For

Juan Francisco Díaz Frias Julián Puyo

Abril 2024

1. Ejercicios de programación

1.1. Las canicas y los frascos

Suponga que tenemos m canicas y n frascos para almacenarlas, cada uno con capacidad para un máximo de c canicas.

Su tarea es desarrollar un programa que genere todas las formas de distribuir las m canicas en los n frascos respetando la capacidad de cada frasco. Por ejemplo si $m = 10$, $n = 3$ y $c = 5$, las distintas soluciones se ven en la siguiente tabla:

0	5	5	3	2	5	4	3	3	5	3	2
1	4	5	3	3	4	4	4	2	5	4	1
1	5	4	3	4	3	4	5	1	5	5	0
2	3	5	3	5	2	5	0	5			
2	4	4	4	1	5	5	1	4			
2	5	3	4	2	4	5	2	3			

Representación de los datos Para modelar una configuración de un frasco con el número de canicas que contiene usaremos una tupla con dos enteros:

```
type Frasco = (Int,Int)
```

Así, la tupla $(2, 3)$ representa el frasco número 2 con 3 canicas.

Una forma de distribuir las m canicas en n frascos, se puede ver como una lista de n frascos, cada uno con un identificador diferente, con un número de canicas. Para el ejemplo de referencia la lista $List((1, 3), (2, 5), (3, 2))$ es una posible distribución de las 10

canicas en los 3 frascos sin violar la capacidad de canicas de cada frasco. También lo es la lista $List((1,4),(2,1),(3,5))$. Por ello definimos el tipo:

```
type Distr = List[Frasco]
```

Y por tanto

```
val res1: List[Distr] =
List(
List((1,0), (2,5), (3,5)), List((1,1), (2,4), (3,5)), List((1,1), (2,5), (3,4)),
List((1,2), (2,3), (3,5)), List((1,2), (2,4), (3,4)), List((1,2), (2,5), (3,3)),
List((1,3), (2,2), (3,5)), List((1,3), (2,3), (3,4)), List((1,3), (2,4), (3,3)),
List((1,3), (2,5), (3,2)), List((1,4), (2,1), (3,5)), List((1,4), (2,2), (3,4)),
List((1,4), (2,3), (3,3)), List((1,4), (2,4), (3,2)), List((1,4), (2,5), (3,1)),
List((1,5), (2,0), (3,5)), List((1,5), (2,1), (3,4)), List((1,5), (2,2), (3,3)),
List((1,5), (2,3), (3,2)), List((1,5), (2,4), (3,1)), List((1,5), (2,5), (3,0))
)
```

sería la lista que se espera como solución al problema del ejemplo.

Para calcular esta solución vamos a ir por pasos.

1.1.1. Canicas posibles en un frasco

Implemente la función `canicasPosiblesFrasco` que reciba f, el identificador de un frasco, $(1 \leq f \leq n)$ y c y devuelva la lista de posibles configuraciones del frasco f.

```
def canicasPosiblesFrasco(f:Int, c:Int):List[Frasco] ={ ... }
```

Por ejemplo una invocación a $canicasPosiblesFrasco(2,5)$ devuelve:

```
val res0: List[Frasco] = List((2,0), (2,1), (2,2), (2,3), (2,4), (2,5))
```

indicando que el frasco 2, puede llegar a tener 0, 1, 2, 3, 4 o 5 canicas.

1.1.2. Canicas posibles por frasco

Implemente la función `canicasPorFrasco` que reciba n, el número de frascos y c y devuelva la lista de canicas posibles por cada frasco. Su implementación debe invocar a la función definida en el punto anterior.

```
def canicasPorFrasco(n:Int, c:Int):List[Distr] ={ ... }
```

Por ejemplo una invocación a $canicasPorFrasco(3,5)$ devuelve:

```
List(
List((1,0), (1,1), (1,2), (1,3), (1,4), (1,5)),
List((2,0), (2,1), (2,2), (2,3), (2,4), (2,5)),
List((3,0), (3,1), (3,2), (3,3), (3,4), (3,5))
)
```

1.1.3. Combinaciones de canicas por frasco

Implemente la función `mezclarLCanicas` que dada una lista de canicas posibles por cada frasco (como la calculada en el ejercicio anterior) devuelva la lista de todas las

combinaciones posibles de canicas del frasco 1, con canicas del frasco 2, ..., con canicas del frasco n.

Note que esta función no tiene en cuenta el número total de canicas a distribuir, por tanto puede calcular combinaciones que no hagan parte de la respuesta buscada en este ejercicio. Su implementación debe invocar a la función definida en el punto anterior.

```
def mezclarLCanicas(lc:List[Distr]):List[Distr] ={ ... }
```

Por ejemplo una invocación a $mezclarLCanicas(canicasPorFrasco(3,5))$ devuelve:

```
List(
List((1,0), (2,0), (3,0)), List((1,0), (2,0), (3,1)), List((1,0), (2,0), (3,2)),
List((1,0), (2,0), (3,3)), List((1,0), (2,0), (3,4)), List((1,0), (2,0), (3,5)),
List((1,0), (2,1), (3,0)), List((1,0), (2,1), (3,1)), List((1,0), (2,1), (3,2)),
List((1,0), (2,1), (3,3)), List((1,0), (2,1), (3,4)), List((1,0), (2,1), (3,5)),
List((1,0), (2,2), (3,0)), List((1,0), (2,2), (3,1)), List((1,0), (2,2), (3,2)),
List((1,0), (2,2), (3,3)), List((1,0), (2,2), (3,4)), List((1,0), (2,2), (3,5)),
List((1,0), (2,3), (3,0)), List((1,0), (2,3), (3,1)), List((1,0), (2,3), (3,2)),
List((1,0), (2,3), (3,3)), List((1,0), (2,3), (3,4)), List((1,0), (2,3), (3,5)),
List((1,0), (2,4), (3,0)), List((1,0), (2,4), (3,1)), List((1,0), (2,4), (3,2)),
List((1,0), (2,4), (3,3)), List((1,0), ...
```

Nótese que en esta lista de distribuciones, cada distribución está compuesta de una lista de n frascos, del 1 al n con las canicas que contiene cada frasco. Entre todas estas distribuciones, algunas contienen las m canicas, y el resto no. Luego necesitamos encontrar de esta lista las que sí contienen m canicas.

1.1.4. Calculando las distribuciones requeridas

Implemente la función `distribucion` que dadas m canicas para distribuir en n frascos, con máximo c canicas por frasco calcule todas las formas de distribuir las canicas en los frascos respetando la capacidad de cada frasco. La salida vendrá en una lista de listas de n frascos ($List((1,c_1),(2,c_2),\ldots,(n,c_n))$) tal que $c_1 + c_2 + \ldots + c_n = m$.

```
def distribucion(m:Int, n:Int, c:Int):List[Distr] = { ... }
```

Por ejemplo una invocación a $distribucion(10,3,5)$ devuelve:

```
List(
List((1,0), (2,5), (3,5)), List((1,1), (2,4), (3,5)), List((1,1), (2,5), (3,4)),
List((1,2), (2,3), (3,5)), List((1,2), (2,4), (3,4)), List((1,2), (2,5), (3,3)),
List((1,3), (2,2), (3,5)), List((1,3), (2,3), (3,4)), List((1,3), (2,4), (3,3)),
List((1,3), (2,5), (3,2)), List((1,4), (2,1), (3,5)), List((1,4), (2,2), (3,4)),
List((1,4), (2,3), (3,3)), List((1,4), (2,4), (3,2)), List((1,4), (2,5), (3,1)),
List((1,5), (2,0), (3,5)), List((1,5), (2,1), (3,4)), List((1,5), (2,2), (3,3)),
List((1,5), (2,3), (3,2)), List((1,5), (2,4), (3,1)), List((1,5), (2,5), (3,0))
)
```

1.2. Cálculo de posibles agrupaciones crecientes

Dado un conjunto de m elementos, se desea calcular todas las posibles maneras de dividir el conjunto en grupos de tamaños diferentes. Por ejemplo, si $m = 6$, el conjunto se podría dividir en:

- 3 conjuntos de tamaños 1, 2 y 3 respectivamente.

- 2 conjuntos de tamaños 1 y 5 respectivamente.
- 2 conjuntos de tamaños 2 y 4 respectivamente.
- 1 conjunto de tamaño 6.

En este ejercicio se quiere que usted implemente una función `agrupaciones` en Scala que reciba un número natural $m > 0$ y devuelva una lista con todas las maneras de agrupar un conjunto de tamaño m en subconjuntos de tamaños diferentes.

```
def agrupaciones(m:Int): List[List[Int]] ={ ... }
```

Por ejemplo la invocación a *agrupaciones*(6) y a *agrupaciones*(10) debe dar como resultado:

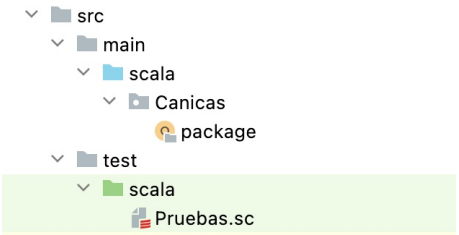
```
val res5: List[List[Int]] = List(List(6), List(1, 5), List(2, 4), List(1, 2, 3))
val res6: List[List[Int]] = List(List(10), List(1, 9), List(2, 8), List(3, 7), List(4, 6),
                                 List(1, 2, 7), List(1, 3, 6), List(1, 4, 5), List(2, 3, 5),
                                 List(1, 2, 3, 4))
```

Su solución debe usar expresiones for y utilizar la función `distribucion` definida en el ejercicio anterior. Puede utilizar las funciones auxiliares que considere necesarias.

2. Entrega

2.1. Paquete *Canicas* 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, `canicasPosiblesFrasco, canicasPorFrasco, mezclarLCanicas, distribucion` y `agrupaciones` deben ser implementadas en un paquete de Scala denominado `Canicas`. **Utilice expresiones for en donde sea pertinente.** En ese paquete debe venir un archivo denominado `package.scala` que debe tener la forma siguiente:

```
package object Canicas {
  type Frasco = (Int,Int)

  type Distr = List[Frasco]
```

```
  def canicasPosiblesFrasco(f:Int, c:Int):List[Frasco] ={
...  }

  def canicasPorFrasco(n:Int, c:Int):List[Distr] = {
...
  }

  def mezclarLCanicas(lc:List[Distr]):List[Distr] = {
...
  }

  def distribucion(m:Int, n:Int, c:Int):List[Distr] = {
...
  }

  def agrupaciones(m:Int): List[List[Int]] ={
...
  }
}
```

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 Canicas._
canicasPosiblesFrasco(2,5)
val cpf= canicasPorFrasco(3,5)
mezclarLCanicas(cpf)
distribucion(10,3,5)
distribucion(3,1,3)
agrupaciones(6)
agrupaciones(10)
```

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 de colecciones y expresiones for, informe de corrección y conclusiones.

2.2.1. Informe de uso de colecciones y expresiones for

Tal como se ha visto en clase, el uso de colecciones y expresiones for 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 de colecciones y expresiones for 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, `canicasPosiblesFrasco, canicasPorFrasco, mezclarLCanicas, distribucion` y `agrupaciones`, 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 25 de abril 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.