

# Secuencias

Mauro Jaskelioff

23/05/2016

- ▶ *Seq* es un TAD para representar secuencias de elementos.
- ▶ A continuación veremos algunas de sus operaciones y las especificaremos en términos de la noción matemática de secuencias.
- ▶ Denotaremos la longitud (cant. de elementos) de una secuencia  $s$  como  $|s|$ .
- ▶ Denotamos las secuencias como  $\langle x_0, x_1, x_2, \dots, x_n \rangle$ .
- ▶ Un índice  $i \in \mathbb{N}$  es válido en  $s$  si  $0 \leq i < |s|$ .
- ▶ Para  $i$  válido y secuencia  $s$ , notamos con  $s_i$  a su  $i$ -ésima proyección.
- ▶  $s'$  es subsecuencia de  $s$  si existe secuencia  $I$  de índices válidos en  $s$  estrictamente crecientes tal que  $s'_i = s_{I_i}$ .

Sea  $s$  una secuencia, y  $x$  un elemento.

▶  $empty : Seq\ a$

Representa la secuencia  $\langle \rangle$ .

▶  $singleton : a \rightarrow Seq\ a$

$singleton\ x$  evalúa a la secuencia  $\langle x \rangle$ .

▶  $length : Seq\ a \rightarrow \mathbb{N}$

$length\ s$  evalúa a  $|s|$ .

▶  $nth : Seq\ a \rightarrow \mathbb{N} \rightarrow a$

Si  $i$  es un índice válido, entonces  $nth\ s\ i$  evalúa a  $s_i$ .

## Más operaciones

- ▶  $tabulate : (\mathbb{N} \rightarrow a) \rightarrow \mathbb{N} \rightarrow Seq\ a$   
 $tabulate\ f\ n$  evalúa a la secuencia  $\langle f\ 0, f\ 1, f\ 2, \dots, f\ (n-1) \rangle$
- ▶  $map : (a \rightarrow b) \rightarrow Seq\ a \rightarrow Seq\ b$   
 $map\ f\ s$  evalúa a la secuencia  $\langle f\ s_0, f\ s_1, f\ s_2, \dots, f\ s_{|s|-1} \rangle$
- ▶  $filter : (a \rightarrow Bool) \rightarrow Seq\ a \rightarrow Seq\ a$   
 $filter\ p\ s$  evalúa a la subsecuencia más larga en la que  $p$  vale para todo sus elementos
- ▶  $append : Seq\ a \rightarrow Seq\ a \rightarrow Seq\ a$   
 $append\ s\ t$  es la secuencia  $\langle s_0, s_1, \dots, s_{|s|-1}, t_0, t_1, \dots, t_{|t|-1} \rangle$
- ▶  $take, drop : Seq\ a \rightarrow \mathbb{N} \rightarrow Seq\ a$   
 $take\ s\ n$  evalúa a la secuencia  $\langle s_0, s_1, \dots, s_{\min(|s|, n)-1} \rangle$   
 $drop\ s\ n$  evalúa a la secuencia  $\langle s_n, \dots, s_{|s|-1} \rangle$

# Vista como árbol

- ▶ La siguiente operación nos permite ver a las secuencias como si fueran un árbol

$showt : Seq\ a \rightarrow TreeView\ a$

**data**  $TreeView\ a = EMPTY \mid ELT\ a \mid NODE\ (Seq\ a)\ (Seq\ a)$

Si  $|s| = 0$ ,  $showt\ s$  evalúa a  $EMPTY$ .

Si  $|s| = 1$ ,  $showt\ s$  evalúa a  $ELT\ s_0$ .

Si  $|s| > 1$ ,  $showt\ s$  evalúa a  $NODE\ (take\ s\ \frac{|s|}{2})\ (drop\ s\ \frac{|s|}{2})$ .

- ▶ Notar que *Treeview* no es un tipo recursivo.

# La operación *foldr*

- ▶  $foldr : (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow Seq\ a \rightarrow b$
- ▶ Intuitivamente,  $foldr\ \oplus\ e\ s = s_0 \oplus (s_1 \oplus (\dots (s_{|s|-1} \oplus e) \dots))$
- ▶ El argumento  $e$  corresponde a la secuencia vacía.
- ▶  $foldr$  va aplicando la función  $\oplus$  a todos los elementos de la secuencia en un orden en particular (de derecha a izquierda).
- ▶ Ej: Definir una operación tal que la secuencia se recorra de izq a derecha.

$$foldl : (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow Seq\ a \rightarrow b$$

# La operación *reduce*

- ▶  $reduce : (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow Seq\ a \rightarrow a$
- ▶ En  $reduce \oplus e$ , el valor  $e$  corresponde a la secuencia vacía.
- ▶ Si  $\oplus$  es asociativa,  $reduce \oplus e\ s$  evalúa a la suma con respecto a  $\oplus$  de la secuencia  $s$ .
  - ▶ Si  $e$  es neutro respecto de  $\oplus$ , coincide con  $foldr \oplus e$  y  $foldl \oplus e$ .
- ▶ Si  $\oplus$  no es asociativa, obtenemos diferentes resultados dependiendo del orden de reducción.
- ▶ Limitarse a funciones asociativas no es una buena idea
  - ▶ ¡La suma y multiplicación de punto flotante no es asociativa!
- ▶ Por lo tanto, para especificar el comportamiento de *reduce* debemos especificar el orden de reducción
- ▶ Notar que el orden de reducción es parte del TAD y no de una implementación en particular.

## Orden de reducción de *reduce*

- ▶ Para especificar el orden de reducción ponemos los datos en un árbol de combinación

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
toTree  : Seq a → Tree a
toTree s = case |s| of
    1 → (Leaf s0)
    n → Node (toTree (take s pp))
              (toTree (drop s pp))
    where pp = 2 ↑ ilg (n - 1)
```

- ▶ Si  $|s| = 2^k$ , el resultado es un árbol binario perfecto.
- ▶ Ahora podemos definir reduce sobre árboles:

```
reduceT ⊕ (Leaf x) = x
reduceT ⊕ (Node l r) = (reduceT ⊕ l) ⊕ (reduceT ⊕ r)
```



## Orden de reducción de *reduce* (cont.)

- ▶ Ahora podemos *especificar reduce* para secuencias:
  - ▶ Si  $|s| = 0$ ,  $reduce \oplus b s$  evalúa a  $b$
  - ▶ Si  $|s| > 0$ , y  $reduceT \oplus (toTree s)$  evalúa a  $v$ , entonces  $reduce \oplus b s$  evalúa a  $(b \oplus v)$ .
- ▶ Con esto no estamos dando una implementación de *reduce*, sólo especificando el orden de reducción.
- ▶ Ejemplo: Dado  $s = \langle s_0, s_1, s_2, s_3, s_4, s_5, s_6 \rangle$ ,

$$reduce \oplus b s = b \oplus (((s_0 \oplus s_1) \oplus (s_2 \oplus s_3)) \oplus ((s_4 \oplus s_5) \oplus s_6))$$

- ▶ Ejercicio: Dar el orden de reducción para una secuencia de longitud 10.

## Divide & Conquer con *reduce*

- Podemos definir un algoritmo divide and conquer genérico.

```
dyc s = case showt s of  
  EMPTY → val  
  ELT v → base v  
  NODE l r → let (l', r') = dyc l || dyc r  
               in combine l' r'
```

sólo tenemos que instanciar las “cajas”

- En una línea:

```
reduce combine val (map base s)
```

# MCSS con reduce

- ▶ El problema de encontrar la máxima subsecuencia contigua se puede resolver con el patrón *dyc*.
- ▶ La solución devuelve una tupla con
  1. el resultado deseado
  2. el máximo prefijo
  3. el máximo sufijo
  4. la suma total
- ▶ Definimos

$val = (0, 0, 0, 0)$

$base\ v = \mathbf{let}\ v' = \max\ v\ 0\ \mathbf{in}\ (v', v', v', v)$

$combine\ (m, p, s, t)\ (m', p', s', t') = (\max\ (s + p')\ m\ m',$   
 $\max\ p\ (t + p'),$   
 $\max\ s'\ (s + t'),$   
 $t + t')$

- ▶ La solución es:  $reduce\ combine\ val\ (map\ base\ s)$

# La operación *scan*

- ▶  $scan : (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow Seq\ a \rightarrow (Seq\ a, a)$
- ▶ Cuando  $\oplus$  es asociativa,  $scan\ \oplus\ b\ s$ , es equivalente a

$$scan\ \oplus\ b\ s = (tabulate\ (\lambda i \rightarrow reduce\ \oplus\ b\ (take\ s\ i))\ |s|, \\ reduce\ \oplus\ b\ s)$$

- ▶ Esta es una especificación. Como implementación sería muy ineficiente.
- ▶ Cuidado: ¡Sólo especifica el comportamiento para  $\oplus$  asociativa!
- ▶ Notablemente, *scan* se puede implementar con

$$W(scan\ \oplus\ l\ s) = O(|s|) \quad S(scan\ \oplus\ i\ s) = O(\lg |s|)$$

# MCSS usando *scan*

- ▶ El problema MCSS es

$$\begin{aligned} mcss\ s &= \max_{0 \leq i \leq j \leq |s|} \left( \sum_{k=i}^{j-1} s_k \right) \\ &= \max_{0 \leq i \leq j \leq |s|} \left( \sum_{k=0}^{j-1} s_k - \sum_{l=0}^{i-1} s_l \right) \\ &= \max_{0 \leq i \leq j \leq |s|} (X_j - X_i) \quad \text{donde } X = scan + 0\ s \\ &= \max_{0 \leq j \leq |s|} (X_j - \min_{0 \leq i \leq j} X_i) \end{aligned}$$

- ▶ Esto nos da el algoritmo:

```
mcSS s = let x = scan + 0 s
          m = scan min ∞ x
          in max (tabulate (\j → xj - mj) |s|)
```

# Arreglos Persistentes

- ▶ Se puede acceder en tiempo constante a cualquier índice pero no se puede hacer updates destructivos.
- ▶ Tienen operaciones para crear arreglos a partir de una lista, funciones y a partir de otros arreglos.
  - ▶  $length :: Arr\ a \rightarrow Int$   
 $length\ p$  devuelve el tamaño del arreglo  $p$ .
  - ▶  $nth :: Arr\ a \rightarrow Int \rightarrow a$   
 $nth\ p\ i$  es la proyección  $i$ -ésima del arreglo  $p$ .
  - ▶  $fromList :: [a] \rightarrow Arr\ a$   
Construye un arreglo a partir de una lista.
  - ▶  $tabulate :: (Int \rightarrow a) \rightarrow Int \rightarrow Arr\ a$   
 $tabulate\ f\ n$  construye un arreglo  $p$  de tamaño  $n$  tal que  $nth\ p\ i == f\ i$  para todo  $0 \leq i < n$ .
  - ▶  $subarray :: Arr\ a \rightarrow Int \rightarrow Int \rightarrow Arr\ a$   
 $subarray\ a\ i\ l$  construye el subarreglo de  $a$  que comienza en el índice  $i$  y es de longitud  $l$ .

## Costos de operaciones en arreglos persistentes

<b>Operación</b>	<b>W</b>	<b>S</b>
<i>length p</i> <i>nth p i</i>	$O(1)$	$O(1)$
<i>fromList xs</i>	$O( xs )$	$O( xs )$
<i>tabulate f n</i>	$O\left(\sum_{i=0}^n W(f\ i)\right)$	$O\left(\max_{i=0}^n S(f\ i)\right)$
<i>subarray a i l</i>	$O(1)$	$O(1)$

# Especificación de costo basada en arreglos

- Podemos implementar secuencias usando arreglos con los siguientes costos:

<b>Operación</b>	<b>W</b>	<b>S</b>
<i>empty</i>		
<i>singleton</i>		
<i>length</i>		
<i>nth</i>	$O(1)$	$O(1)$
<i>take s n</i>		
<i>drop s n</i>		
<i>showt s</i>		
<i>append s t</i>	$O( s  +  t )$	



## Especificación de costo basada en arreglos (cont.)

Operación	W	S
<i>tabulate f n</i>	$O\left(\sum_{i=0}^{n-1} W(f\ i)\right)$	$O\left(\max_{i=0}^{n-1} S(f\ i)\right)$
<i>map f s</i>	$O\left(\sum_{x \in s} W(f\ x)\right)$	$O\left(\max_{x \in s} S(f\ x)\right)$
<i>filter f s</i>	$O\left(\sum_{x \in s} W(f\ x)\right)$	$O\left(\lg  s  + \max_{x \in s} S(f\ x)\right)$
<i>reduce</i> $\oplus$ <i>b s</i>	$O( s )$	$O(\lg  s )$
<i>scan</i> $\oplus$ <i>b s</i>		

- ▶ Para *reduce* y *scan* suponemos que  $W_f \in O(1)$ .

# Costos de Funciones de Alto Orden

- ▶ Vimos varias operaciones de alto orden para secuencias
  - ▶  $tabulate : (\mathbb{N} \rightarrow a) \rightarrow \mathbb{N} \rightarrow Seq\ a$
  - ▶  $map : (a \rightarrow b) \rightarrow Seq\ a \rightarrow Seq\ b$
  - ▶  $filter : (a \rightarrow Bool) \rightarrow Seq\ a \rightarrow Seq\ a$
  - ▶  $reduce : (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow Seq\ a \rightarrow a$
  - ▶  $scan : (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow Seq\ a \rightarrow (Seq\ a, a)$
- ▶ Para las 3 primeras dimos el costo para cualquier función argumento. Por ej:

$$W(map\ f\ s) = O\left(\sum_{x \in s} W(f\ x)\right) \quad S(map\ f\ s) = O\left(\max_{x \in s} S(f\ x)\right)$$

- ▶ Para  $reduce$  y  $scan$ ,  $W$  es  $O(|s|)$  y  $S$  es  $O(\lg |s|)$  sólo si la función argumento es  $O(1)$ .

## Análisis de costos de *reduce*

- ▶ En *reduce*  $\oplus b$  *s*, el orden de reducción afecta el resultado obtenido si la  $\oplus$  no es asociativa.
- ▶ Si  $\oplus \notin O(1)$ , el orden de reducción también afecta los costos.
- ▶ Consideremos el siguiente algoritmo de ordenación:

*sort s = reduce merge empty (map singleton s)*

$W(\text{merge } s \ t) = O(|s| + |t|)$ ,  $S(\text{merge } s \ t) = O(\lg(|s| + |t|))$

- ▶ Supongamos que *reduce* tiene el siguiente orden de reducción:

$$(x_0 \oplus (x_1 \oplus (\dots \oplus x_{n-1}) \dots))$$

- ▶ ¿Cuál es el costo de *sort*?

## Análisis de costos de *reduce* (cont.)

- ▶ Calculemos el trabajo. Sea  $n = |s|$

$$W(\text{sort } s) \leq \sum_{i=1}^{n-1} c \cdot (1 + i) \in O(n^2)$$

- ▶ Calculemos la profundidad

$$S(\text{sort } s) \leq \sum_{i=1}^{n-1} c \cdot \lg(1 + i) \in O(n \lg n)$$

- ▶ ¿Qué algoritmo de ordenación implementamos?
- ▶ Si en cambio usamos el orden de reducción dado anteriormente

$$W(\text{sort } s) \in O(n \lg n) \quad \Leftarrow \text{¡Un cambio notable!}$$

## Análisis de costos de *reduce* (cont.)

- ▶ El orden de reducción afecta no sólo el resultado ( $\oplus$  no asociativa), si no también el costo.
- ▶ Para definir el costo en general de *reduce* definimos el conjunto

$$\mathcal{O}_r(\text{reduce } \oplus b s) = \{\text{aplicaciones de } \oplus \text{ en el árbol de reducción}\}$$

- ▶ El costo de reduce es:

$$W(\text{reduce } \oplus b s) = O \left( |s| + \sum_{(x \oplus y) \in \mathcal{O}_r(\oplus, b, s)} W(x \oplus y) \right)$$

$$S(\text{reduce } \oplus b s) = O \left( \lg |s| \max_{(x \oplus y) \in \mathcal{O}_r(\oplus, b, s)} S(x \oplus y) \right)$$

# Implementación de *scan*

- ▶  $scan : (a \rightarrow a \rightarrow a) \rightarrow a \rightarrow Seq\ a \rightarrow (Seq\ a, a)$
- ▶ Si  $\oplus$  es asociativa,  $scan \oplus b\ s$ , es equivalente a

$$scan \oplus b\ s = (tabulate (\lambda i \rightarrow reduce \oplus b (take\ s\ i)) |s|, \\ reduce \oplus b\ s)$$

- ▶ ¡No es una implementación eficiente!
- ▶ Ejercicio: Suponiendo  $\oplus \in O(1)$  y la implementación de más arriba ¿cuál es el trabajo y profundidad de  $scan \oplus b\ s$ ?
- ▶ La operación *scan* parece ser inherentemente secuencial.
  - ▶ ¿Cómo implementarla en paralelo?

# Divide & Conquer vs. Contracción

- ▶ Pensemos como resolver el problema con Divide & Conquer:
  - ▶ Partimos la secuencia en dos y llamamos recursivamente. . .
  - ▶ ¡pero la mitad derecha depende de la izquierda!
- ▶ Otra técnica similar a D&Q es *contraer* la entrada.
  - ▶ Para resolver un problema resolvemos una instancia más chica del mismo problema
  - ▶ A diferencia de D&Q, hacemos solo una llamada recursiva.
  - ▶ O sea, tenemos los siguientes pasos
    1. Contraemos la instancia del problema a una instancia (mucho) más chica (contracción).
    2. Resolvemos recursivamente la instancia chica
    3. Usamos la solución de la instancia chica para resolver la grande (expansión).
  - ▶ Útil en algoritmos paralelos:
    - ▶ contracción y expansión son usualmente paralelizables.
    - ▶ Si contraemos a una fracción, la recursión tiene prof. logarítmica

## Ejemplo de contracción: *reduce*

- ▶ Consideremos

$$\text{reduce} + 0 \langle 2, 3, 6, 2, 1, 4 \rangle$$

- ▶ Contraemos la secuencia de la siguiente manera:

$$\begin{aligned} & \langle 2 + 3, 6 + 2, 1 + 4 \rangle \\ = & \\ & \langle 5, 8, 5 \rangle \end{aligned}$$

- ▶ Obtenemos una secuencia más chica (¡la mitad!)
- ▶ Calculamos recursivamente  $\text{reduce} + 0 \langle 5, 8, 5 \rangle = 18$
- ▶ En este caso la expansión es la identidad.
- ▶ El resultado es 18.



# Contracción/expansión en *scan*

- ▶ Supongamos que  $\oplus$  es asociativa
- ▶  $scan \oplus b \langle x_0, x_1, x_2, x_3 \rangle =$   
 $(\langle b, b \oplus x_0, b \oplus x_0 \oplus x_1, b \oplus x_0 \oplus x_1 \oplus x_2 \rangle, b \oplus x_0 \oplus x_1 \oplus x_2 \oplus x_3)$
- ▶ Contraemos  $\langle x_0, x_1, x_2, x_3 \rangle$  a  $\langle x_0 \oplus x_1, x_2 \oplus x_3 \rangle$
- ▶ Llamamos recursivamente

$$scan \oplus b \langle x_0 \oplus x_1, x_2 \oplus x_3 \rangle =$$
$$(\langle b, b \oplus x_0 \oplus x_1 \rangle, b \oplus x_0 \oplus x_1 \oplus x_2 \oplus x_3)$$

- ▶ Comparemos

$$(\langle b, \quad b \oplus x_0 \oplus x_1 \quad \rangle, b \oplus x_0 \oplus x_1 \oplus x_2 \oplus x_3)$$

$$(\langle b, b \oplus x_0, b \oplus x_0 \oplus x_1, b \oplus x_0 \oplus x_1 \oplus x_2 \rangle, b \oplus x_0 \oplus x_1 \oplus x_2 \oplus x_3)$$

## Contracción/expansión en *scan*

- ▶ Dada la entrada  $s$  y el resultado  $s'$  del llamado recursivo sobre la contracción, queremos obtener  $r$

$$\begin{aligned} s &= \langle x_0, & x_1, & & x_2, & & & x_3 \rangle \\ s' &= \langle b, & & b \oplus x_0 \oplus x_1 & & & & \rangle, b \oplus x_0 \oplus x_1 \oplus x_2 \oplus x_3 \\ r &= \langle b, b \oplus x_0, b \oplus x_0 \oplus x_1, b \oplus x_0 \oplus x_1 \oplus x_2 \rangle, b \oplus x_0 \oplus x_1 \oplus x_2 \oplus x_3 \end{aligned}$$

- ▶ El total lo obtenemos del resultado del llamado recursivo  $s'$
- ▶ Si  $i$  es par,  $r_i = s'_{\frac{i}{2}}$
- ▶ Si  $i$  es impar,  $r_i = s'_{\lfloor \frac{i}{2} \rfloor} + s_{i-1}$
- ▶ Este proceso nos da un algoritmo paralelo para implementar *scan*
- ▶ También fija el orden de reducción.

## Orden de reducción de *scan*

- ▶ Si la operación  $\oplus$  no es asociativa el orden de reducción es importante.
- ▶ Con el algoritmo anteriormente descrito obtenemos

$$\begin{aligned} \text{scan } \oplus b \langle x_0, x_1, x_2, x_3, x_4, x_5 \rangle = \\ & ( \langle b, \\ & \quad b \oplus x_0, \\ & \quad b \oplus (x_0 \oplus x_1), \\ & \quad (b \oplus (x_0 \oplus x_1)) \oplus x_2, \\ & \quad b \oplus ((x_0 \oplus x_1) \oplus (x_2 \oplus x_3)), \\ & \quad (b \oplus ((x_0 \oplus x_1) \oplus (x_2 \oplus x_3))) \oplus x_4 \rangle, \\ & \quad b \oplus (((x_0 \oplus x_1) \oplus (x_2 \oplus x_3)) \oplus (x_4 \oplus x_5)) \\ & ) \end{aligned}$$

- ▶ Notar que **no** coincide con el orden de reducción de *reduce* sobre los prefijos!

- ▶ Calculamos costos para la implementación con arreglos
- ▶ Suponemos que  $\oplus$  tiene costo constante
- ▶ El costo de *scan* para una secuencia de longitud  $n$  es:

$$W(n) = W(n/2) + kn$$

$$S(n) = S(n/2) + k$$

- ▶ Por lo tanto

$$W(n) \in O(n)$$

$$S(n) \in O(\lg n)$$

- ▶ Para definir el costo para  $\oplus$  de costo arbitrario definimos:

$$\mathcal{O}_s(\text{scan } \oplus b s) = \{\text{aplicaciones de } \oplus \text{ en el árbol de reducción}\}$$

## Especificación de costo basada en arreglos

$$W(\text{reduce} \oplus b s) = O \left( |s| + \sum_{(x \oplus y) \in \mathcal{O}_r(\oplus, b, s)} W(x \oplus y) \right)$$

$$S(\text{reduce} \oplus b s) = O \left( \lg |s| \max_{(x \oplus y) \in \mathcal{O}_r(\oplus, b, s)} S(x \oplus y) \right)$$

$$W(\text{scan} \oplus b s) = O \left( |s| + \sum_{(x \oplus y) \in \mathcal{O}_s(\oplus, b, s)} W(x \oplus y) \right)$$

$$S(\text{scan} \oplus b s) = O \left( \lg |s| \max_{(x \oplus y) \in \mathcal{O}_s(\oplus, b, s)} S(x \oplus y) \right)$$

$\mathcal{O}_r(\oplus, b, s)$  es el conj. de aplicaciones de  $\oplus$  en *reduce*.

$\mathcal{O}_s(\oplus, b, s)$  es el conj. de aplicaciones de  $\oplus$  en *scan*.

# Trabajando con elementos ordenados

- ▶ Supongamos que trabajamos con secuencias  $Seq\ a$ , donde existe un orden total para los elementos en  $a$ .
- ▶ Esto induce un orden lexicográfico sobre secuencias.
- ▶ También podemos extender el TAD de secuencias con otras operaciones
- ▶  $maxS : Seq\ a \rightarrow \mathbb{N}$ , que devuelve el índice de un máximo en una secuencia no vacía.
  - ▶ Ejercicio: implementar  $maxS$

- ▶ Sea  $a$  un tipo con un orden total (el tipo de las claves)
- ▶ Sea  $b$  un tipo cualquiera (el tipo de los datos)
- ▶ La función  $collect : Seq(a \times b) \rightarrow Seq(a \times Seq b)$  recolecta todos los datos asociados a cada clave.
- ▶ La secuencia resultado está ordenada según el orden de  $a$ .
- ▶ Ejemplo:

$$\begin{aligned} collect \langle \langle (3, "EDyAII"), \\ (1, "Prog1"), \\ (2, "Prog2"), \\ (2, "EDyAI"), \\ (3, "SOI") \rangle \rangle \\ = \langle \langle (1, \langle "Prog1" \rangle), \\ (2, \langle "Prog2", "EDyAI" \rangle), \\ (3, \langle "EDyAII", "SOI" \rangle) \rangle \rangle \end{aligned}$$

## Implementación de *collect*

- ▶ *collect* se puede implementar en dos pasos
  1. Ordenar la entrada según las claves.  
Esto tiene como efecto juntar claves iguales
  2. Juntar todas los valores de claves iguales.
- ▶ Ordenar tiene  $W(n) \in O(W_c n \lg n)$  y  $S(n) \in O(S_c \lg^2 n)$   
 $W_c$  y  $S_c$  es el trabajo y profundidad de la comparación de claves.
- ▶ Juntar todas las claves es  $W(n) \in O(n)$  y  $S(n) \in O(\lg n)$ .
- ▶ Por lo tanto el costo está dominado por el costo de la ordenación.



# Map-reduce

- ▶ El paradigma “map-reduce” fue inventado por Google para el procesamiento paralelo sobre enormes colecciones de datos.
- ▶ Fuera de Google, es muy utilizada la implementación Hadoop de Apache.
- ▶ A pesar de su nombre, se hace un *map*, seguido de un *collect*, seguido de varios *reduce*.
- ▶ Ejemplo: Asociar a cada palabra clave una secuencia de documentos donde ésta ocurre.
  - ▶ *map apv* se aplica a una secuencia de documentos, donde *apv* transforma cada documento en secuencias de pares clave/valor
  - ▶ Esto da una secuencia de secuencias de pares clave/valor que se aplana con *join*.
  - ▶ Luego se aplica *collect*
  - ▶ Luego cada par de clave y secuencia de valores es “reducido” a un resultado por *red*.

## Map-reduce (cont)

- ▶ Generalmente *apv* y *red* son funciones secuenciales
- ▶ El paralelismo viene de aplicarlas en un *map*

$$\begin{aligned} \text{mapCollectReduce } apv \text{ red } s = & \mathbf{let} \text{ pairs } = \text{join } (\text{map } apv \text{ } s) \\ & \text{groups } = \text{collect } pairs \\ & \mathbf{in} \text{ map red groups} \end{aligned}$$

- ▶ ¿Cuántas veces aparece una palabra en una colección de documentos?

$$\begin{aligned} apv \text{ } d &= \text{map } (\lambda w \rightarrow (w, 1)) (\text{words } d) \\ red \text{ } (w, s) &= (w, \text{reduce } (+) 0 \text{ } s) \\ \text{countWords } s &= \text{mapCollectReduce } apv \text{ red } s \end{aligned}$$

- ▶ Secuencias
- ▶ TAD para modelar secuencias.
- ▶ Facilitan la programación paralela.
- ▶ Análisis de costos en funciones con alto orden.
  - ▶ El orden de reducción afecta el costo además del resultado
- ▶ Implementación de *scan*
  - ▶ Notablemente se logra buen paralelismo
- ▶ Paradigma Map-Reduce de Google.