Tema 9. Eficiencia

Hay expresiones equivalentes para resolver un problema pero distintas en el coste de evaluarlas.

- Ejemplo 1: Inmersión mediante *parámetros acumuladores*

```
inversa [] = []
inversa (x:s) = inversa s ++ [x]
```

? inversa [1..n]

tiempo: $O(n^2)$

```
inversaE s = invConc [] s
invConc t [] = t
invConc t (x:s) = invConc (x:t) s
```

? inversaE [1..n]

tiempo: $O(n)$

<u>Ejercicio</u>: Probar que invConc t s = inversa s ++ t

Eficiencia mediante inmersión

- Ejemplo 2: Inmersión mediante *resultados adicionales*

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

? fib n

tiempo: $O(k^n)$

```
fibE n = fst (fibDos n)
fibDos 0 = (0,1)
fibDos n = (b, a+b)
              where (a,b) = fibDos (n-1)
```

? fibE n

tiempo: $O(n)$

Ejercicio: Probar que fibDos n = (fib n , fib (n+1))

Evaluación perezosa y eficiencia (1)

Un mismo algoritmo con evaluación perezosa puede ser más eficiente que con evaluación impaciente.

- Ejemplo 3: Mínimo de una lista (*mediante* ordIns)

```
minimoLista = head . ordIns
ordIns = foldr ins []          -- ordenación por inserción
         where
         ins x [] = [x]
         ins x (y:s) = if x<=y then (x:y:s) else (y: ins x s)
```

Con ev. perezosa tiempo = $O(n)$ *y con impaciente* $O(n^2)$ *para hallar el mínimo de una lista de* n *elementos.*

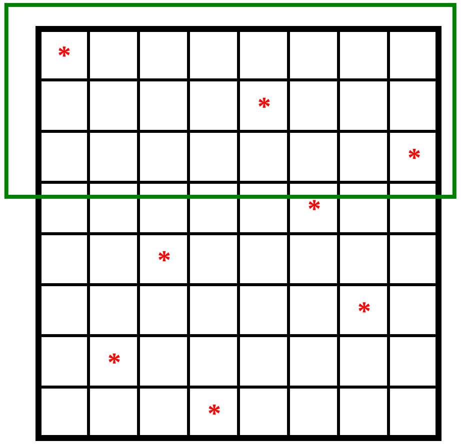
Ejercicio: Dar los pasos de ? minimoLista [8,6,2,7,5]

Evaluación perezosa y eficiencia (2)

El mismo algoritmo que sirve para generar <u>todas</u> las soluciones de un problema de "backtracking", genera la primera solución de forma eficiente.

- Ejemplo 4: Problema de las 8 reinas

  [1,5,8,6,3,7,2,4] *representa al tablero*



soluciones = reinas 8

reinas 0 = [[]]

reinas m = [p++[n] | p <– reinas (m-1), n <– [1..8] ,
seguro p n]

Evaluación perezosa y eficiencia (3)

(sigue el problema de la 8 reinas)

```
seguro p n = and [not (mata (i,j) (k,n)) | (i,j) <– zip [1..h] p]
             where h = length p ; k = h+1
mata (i,j) (k,n)  =  (j == n) || (i+j == k+n) || (i-j == k-n)
```

? soluciones

[[1,5,8,6,3,7,2,4],[1,6,8,3,7,4,2,5],[1,7,4,6,8,2,5,3],......

(lista con las 92 soluciones)

? head soluciones

[1,5,8,6,3,7,2,4]

¡SOLO da los pasos necesarios para calcular 1 solución!

Estructuras cíclicas (1)

Al igual que las funciones, las estructuras de datos pueden definirse de forma recursiva:

unos = 1: unos

mas = “Mas ”++ymas **where** ymas = “y mas ” ++ymas

Estos dos ejemplos definen listas infinitas:

unos => [1,1,1,1,1,...]

mas => “Mas y mas y mas y mas y mas ....”

que se guardan en memoria mediante un grafo (finito):

1 : ◎ ‘M’:‘a’:‘s’:‘ ’:‘y’:‘ ’:‘m’:‘a’:‘s’:‘ ’:◎

Estructuras cíclicas (2)

Supongamos la función **forever x = [x,x,x,…]**.

- *Una posible definición es:* **forever x = x: forever x**

  Con esta definición ***NO*** *se crea una estructura cíclica (la representación crece en memoria tras cada paso)*

  forever 1 ⇒ 1:forever 1 ⇒ ... ⇒ 1:1:1:1:forever 1

- *Otra posible definición es:* **foreverC x = s where s= x:s**

  que crea la estructura cíclica:

  foreverC 1 ⇒ 1 :

Ej: Ver consumo de memoria (:set +s) para

sum (take 1000000 (**f** 1)) *con* **f** = forever *versus* **f** = foreverC

Estructuras cíclicas (3)

Definición de la función **iterate** *usando una estructura cíclica:*

iterate f x = s where s = x: map f s

Primeros pasos de evaluación de la expresión **iterate (2*) 1**

iterate (2*) 1

⇒ s

⇒ 1: map (2*) ◎

⇒ 1: 2: map (2*) ◎

⇒ 1: 2: 4: map (2*) ◎

⇒ ...

Estructuras cíclicas (4)

Sin embargo, definiendo **iterate f x = x: map f (iterate f x)**
los pasos serían :

iterate (2*) 1

⇒ 1: map (2*) (iterate (2*) 1)

⇒ 1: map (2*) (1: (map (2*) (iterate(2*)1))

⇒ 1: 2: map (2*) (map (2*) (iterate (2*) 1))

⇒ 1: 2: map (2*) (map (2*)(1: map (2*) (iterate (2*) 1)))

⇒ 1: 2: map (2*) (2: map (2*) (map (2*) (iterate (2*) 1)))

⇒ 1: 2: 4: map (2*) (map (2*) (map (2*) (iterate (2*) 1)))

Aplicación: Números de Hamming

- Problema:

  Escribir un programa que produzca una lista infinita de números con las siguientes propiedades:

  a) La lista es ascendente y sin repetidos

  b) Comienza con el 1

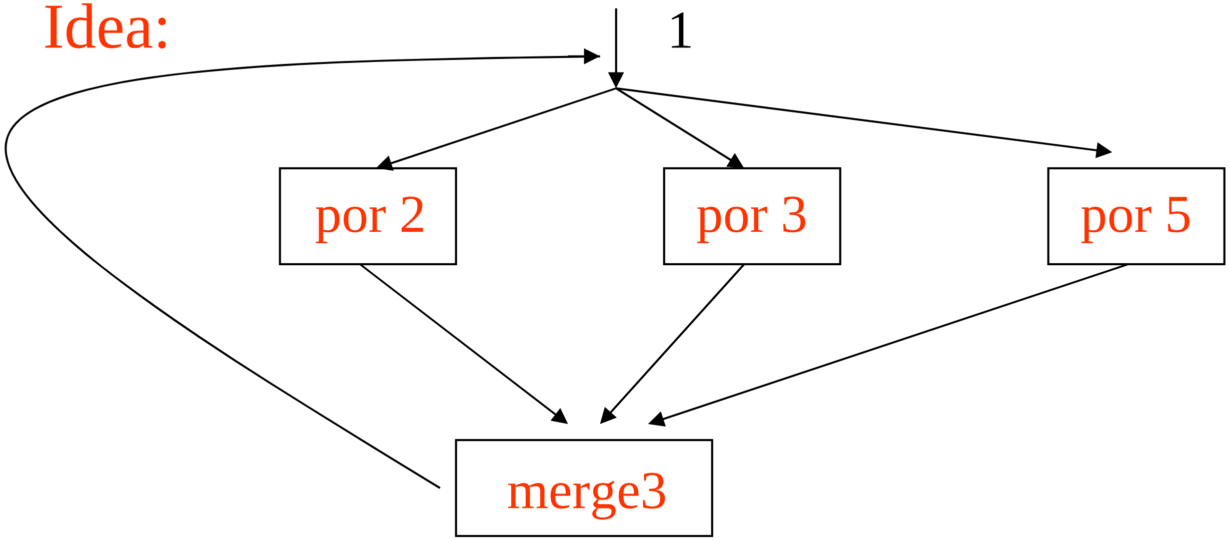
  c) Si x está en la lista, también lo están 2*x, 3*x, 5*x

  d) La lista no contiene otros números

- Lista: 1,2,3,4,5,6,8,9,10,12,15,16,........

Solución: Números de Hamming (1)

- Idea:

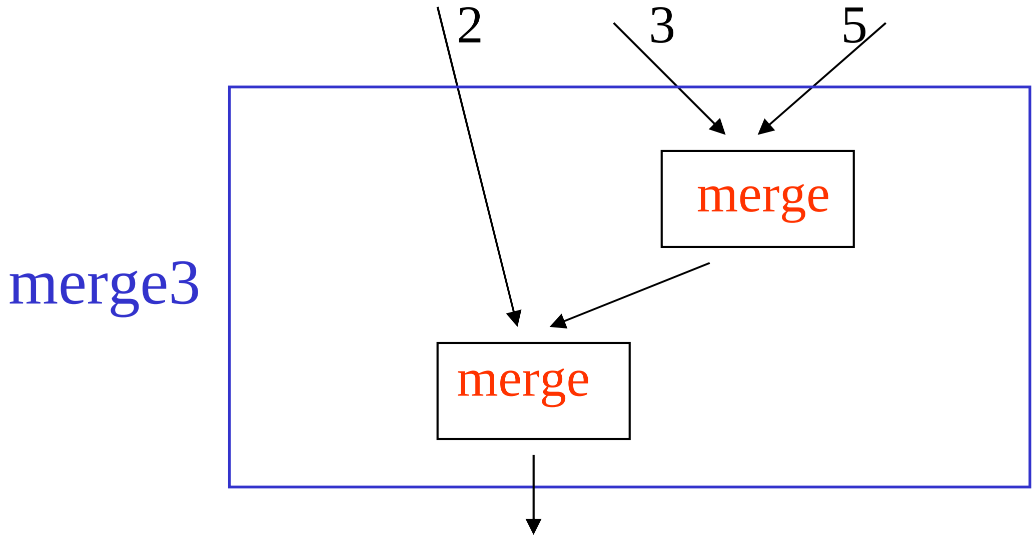


- Programa:

```
hamming = 1: f (hamming)
                where  f s = merge3 (por 2 s) (por 3 s) (por 5 s)
                       por n = map (n*)
```

Solución: Números de Hamming (2)

donde merge3 as bs cs = merge as (merge bs cs)



y merge (x:s) (y:t) | x==y = x: merge s t
| x<y = x: merge s (y:t)
| x>y = y: merge (x:s) t

Solución: Números de Hamming (3)

```
hamming
        =      1: merge (map (2*) ◎)
                   (merge (map (3*) ◎)
                          (map (5*)     ◎))
```

Tras unos pasos se obtiene:

```
hamming
  =  1 : 2 : 3: 4 : 5 : 6 : 8 :
          merge (10: map (2*) ◎)
                (9: merge (map (3*)  ◎)
                         (10: map (5*) ◎))
```

Ejercicios: estructuras cíclicas

Redefinir las siguientes expresiones con estructuras cíclicas (y dar unos pasos de evaluación):

- pares = [2,4..]
- sumas = [sumaHasta i | i <– [1..]]

  where sumaHasta i = sum [1..i]
- factoriales = [fact i | i <– [0..]] (*con* fact n = n!)

Soluciones:

- pares = 2: map (+2) pares
- sumas = 1: zipWith (+) [2..] sumas
- factoriales = 1: zipWith (*) [1..] factoriales