



## Tema 9. Eficiencia

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

- **Ejemplo 1:** Inmersión mediante *parámetros acumuladores*

$\text{inversa } [] = []$

$\text{inversa } (x:s) = \text{inversa } s ++ [x]$

?  $\text{inversa } [1..n]$

tiempo:  $O(n^2)$

$\text{inversaE } s = \text{invConc } [] s$

$\text{invConc } t [] = t$

$\text{invConc } t (x:s) = \text{invConc } (x:t) s$

?  $\text{inversaE } [1..n]$

tiempo:  $O(n)$

---

Ejercicio: Probar que  $\text{invConc } t s = \text{inversa } s ++ t$



## Eficiencia mediante inmersión

- Ejemplo 2: Inmersión mediante *resultados adicionales*

$\text{fib } 0 = 0$

$\text{fib } 1 = 1$

$\text{fib } n = \text{fib } (n-1) + \text{fib } (n-2)$

?  $\text{fib } n$

tiempo:  $O(k^n)$

$\text{fibE } n = \text{fst } (\text{fibDos } n)$

$\text{fibDos } 0 = (0, 1)$

$\text{fibDos } n = (b, a+b)$

where  $(a, b) = \text{fibDos } (n-1)$

?  $\text{fibE } n$

tiempo:  $O(n)$

---

Ejercicio: Probar que  $\text{fibDos } n = (\text{fib } n, \text{fib } (n+1))$



## 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)

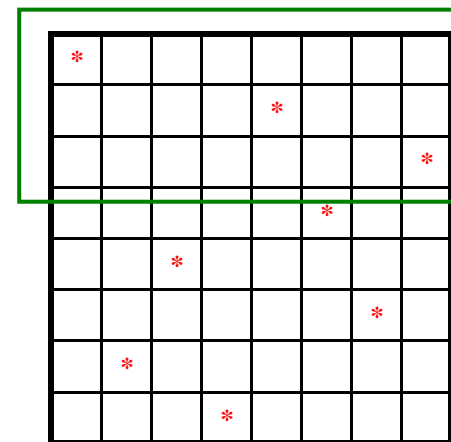
*El mismo algoritmo que sirve para generar todas las soluciones de un problema de “backtracking”, genera la primera solución de forma eficiente.*

- 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!*

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

unos = 1: unos

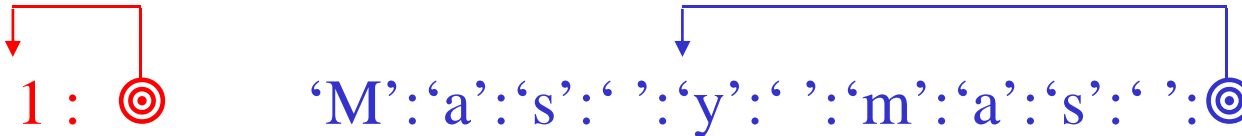
```
mas = “Mas ” ++ ymas where ymas = “y mas ” ++ ymas
```

*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):*





## Estructuras cíclicas (2)

Supongamos la función **forever**  $\mathbf{x} = [\mathbf{x}, \mathbf{x}, \mathbf{x}, \dots]$ .

- Una posible definición es:

**forever**  $\mathbf{x} = \mathbf{x} : \text{forever } \mathbf{x}$


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

**forever** 1  $\Rightarrow$  1:forever 1  $\Rightarrow \dots \Rightarrow$  1:1:1:1:forever 1

- Otra posible definición es:

**foreverC**  $\mathbf{x} = \mathbf{s}$  where  $\mathbf{s} = \mathbf{x} : \mathbf{s}$

que crea la estructura cíclica:

**foreverC** 1  $\Rightarrow$  1 : 

---

Ej: Ver consumo de memoria ( $\text{:set} + s$ ) 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

$\Rightarrow$  s

$\Rightarrow$  1: map (2\*)  $\odot$

$\Rightarrow$  1: 2: map (2\*)  $\odot$

$\Rightarrow$  1: 2: 4: map (2\*)  $\odot$

$\Rightarrow$  ...





## Estructuras cíclicas (4)

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

iterate (2\*) 1

$\Rightarrow 1: \text{map } (2^*) (\text{iterate } (2^*) 1)$

$\Rightarrow 1: \text{map } (2^*) (1: (\text{map } (2^*) (\text{iterate}(2^*)1)))$

$\Rightarrow 1: 2: \text{map } (2^*) (\text{map } (2^*) (\text{iterate } (2^*) 1))$

$\Rightarrow 1: 2: \text{map } (2^*) (\text{map } (2^*) (1: \text{map } (2^*) (\text{iterate } (2^*) 1))))$

$\Rightarrow 1: 2: \text{map } (2^*) (2: \text{map } (2^*) (\text{map } (2^*) (\text{iterate } (2^*) 1))))$

$\Rightarrow 1: 2: 4: \text{map } (2^*) (\text{map } (2^*) (\text{map } (2^*) (\text{iterate } (2^*) 1))))$



## Aplicación: Números de Hamming

---

➤ **Problema:**

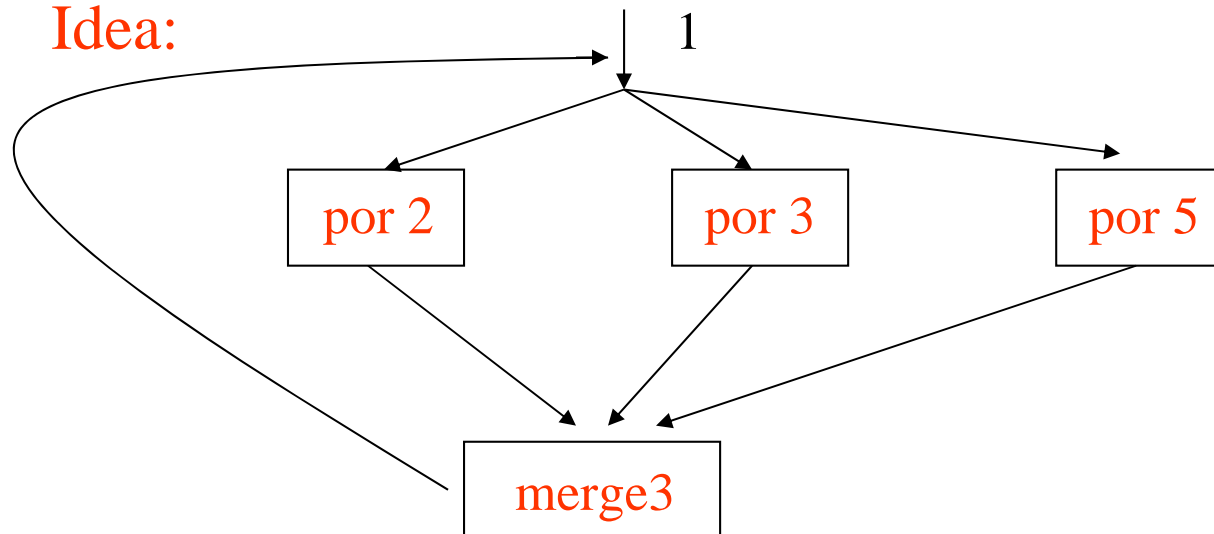
Escribir un programa que produzca una lista infinita de números con las siguientes propiedades:

- 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:

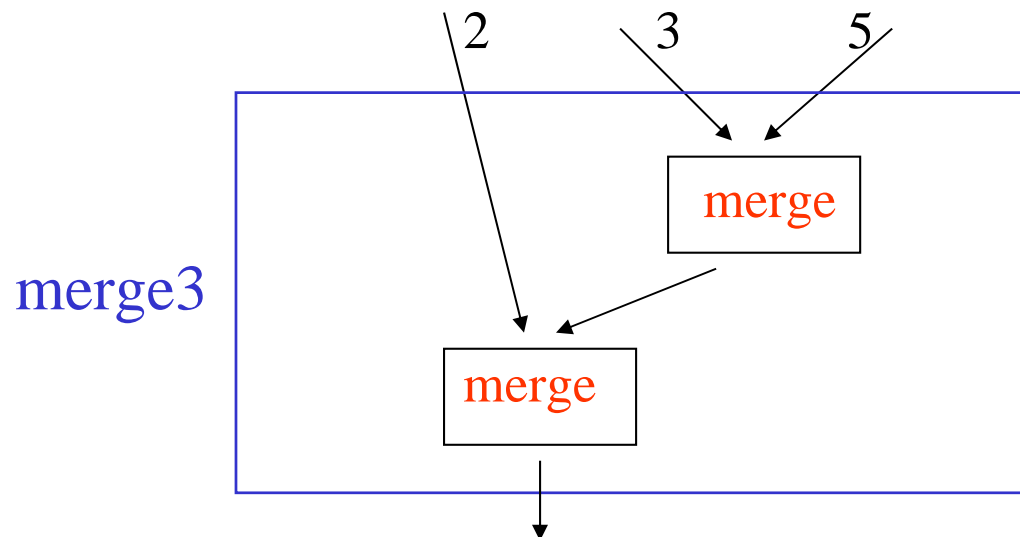


➤ Programa:

```
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)

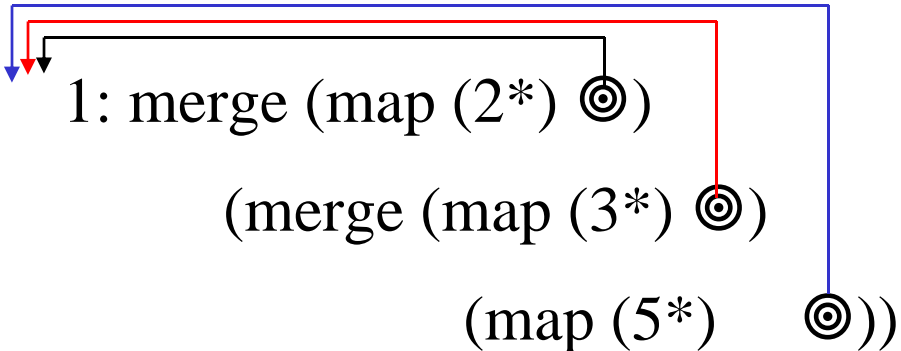
donde  $\text{merge3 as bs cs} = \text{merge as (merge bs cs)}$



$y$	$\text{merge } (x:s) (y:t)$	$x == y = x : \text{merge } s \ t$
		$x < y = x : \text{merge } s (y:t)$
		$x > y = y : \text{merge } (x:s) \ t$

## Solución: Números de Hamming (3)

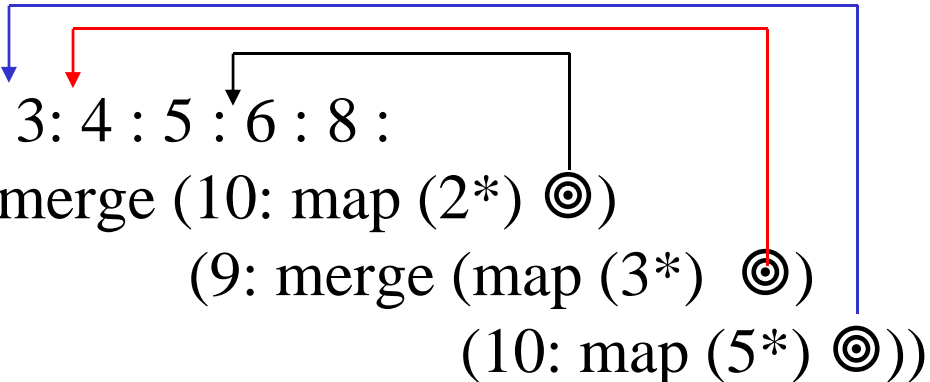
hamming =



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! )