Ordenación y búsqueda

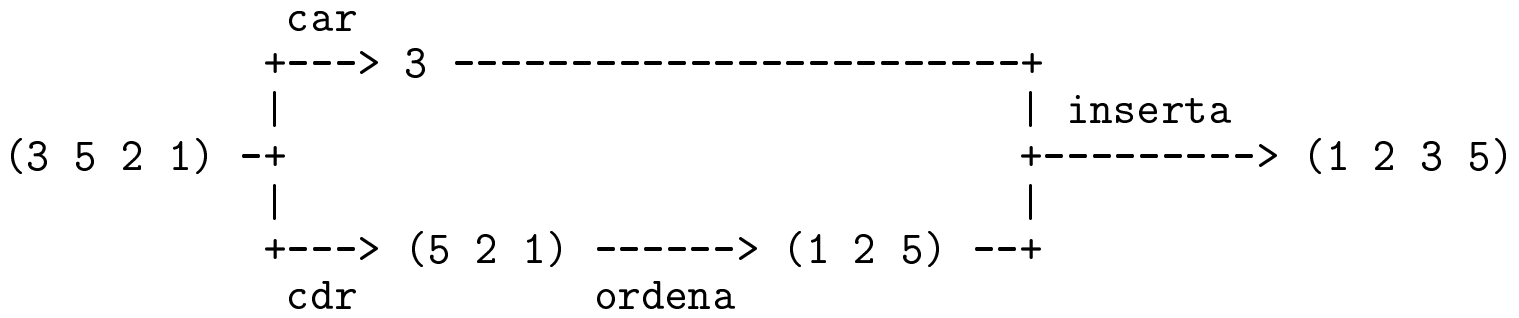
José A. Alonso y María J. Hidalgo

Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Ordenación por inserción

- **Esquema de ordenación por inserción**



Ordenación por inserción

- **Procedimiento de ordenación por inserción de listas**

```
;;; (ordena-por-insercion '(3 1 2))  =>  (1 2 3)
(define ordena-por-insercion
  (lambda (l)
    (if (= 1 (length l))
        l
        (inserta (car l)
                 (ordena-por-insercion (cdr l))))))

;;; (inserta 3 '(1 2 6))  =>  (1 2 3 6)
(define inserta
  (lambda (x l)
    (cond
     ((null? l) (list x))
     ((< x (car l)) (cons x l))
     (else (cons (car l)
                 (inserta x (cdr l)))))))
```

Ordenación por inserción

- **Traza de ordenación por inserción**

```
(ordena-por-insercion '(3 1 2))
=> (inserta 3 (ordena-por-insercion '(1 2)))
=> (inserta 3 (inserta 1 (ordena-por-insercion '(2))))
=> (inserta 3 (inserta 1 '(2)))
=> (inserta 3 '(1 2))
=> (inserta 3 '(1 2))
=> (cons 1 (inserta 3 '(2)))
=> (cons 1 (cons 2 (inserta 3 '())))
=> (cons 1 (cons 2 '(3)))
=> (cons 1 '(2 3))
=> (1 2 3)
```

Ordenación por mezcla ("mergesot")

```
;;; (separa-en-listas-ordenadas '(2 4 1 3 2 1)) => ((2 4)(1 3)(2)(1))
(define separa-en-listas-ordenadas
  (lambda (l)
    (cond ((null? l) '())
          ((null? (cdr l)) (list l))
          (else (let ((aux (separa-en-listas-ordenadas (cdr l))))
                  (if (> (car l) (cadr l))
                      (cons (list (car l)) aux)
                      (cons (cons (car l) (car aux))
                            (cdr aux)))))))))

;;; (mezcla '(2 3 4) '(1 2 3))  =>  (1 2 2 3 3 4)
(define mezcla
  (lambda (l1 l2)
    (cond ((null? l1) l2)
          ((null? l2) l1)
          ((<= (car l1) (car l2))
           (cons (car l1) (mezcla (cdr l1) l2)))
          (else (cons (car l2) (mezcla l1 (cdr l2)))))))
```

Ordenación por mezcla ("mergesot")

```
;;; (mezcla-pares '((2 4) (1 3) (2) (1)))  =>  ((1 2 3 4) (1 2))
;;; (mezcla-pares '((1 3) (2) (0 4 5)))    =>  ((1 2 3) (0 4 5))
;;; (mezcla-pares '((1 3) (2)))            =>  ((1 2 3))
;;; (mezcla-pares '((1 3)))                =>  ((1 3))
;;; (mezcla-pares ())                      =>  ()
(define mezcla-pares
  (lambda (sublistas)
    (cond ((null? sublistas) '())
          ((null? (cdr sublistas)) sublistas)
          (else (cons (mezcla (car sublistas) (cadr sublistas))
                      (mezcla-pares (cddr sublistas)))))))

;;; (mezcla-listas '((2 4) (1 3) (2) (1)))  =>  (1 1 2 2 3 4)
(define mezcla-listas
  (lambda (grupos)
    (if (null? (cdr grupos))
        (car grupos)
        (mezcla-listas (mezcla-pares grupos)))))
```

Ordenación por mezcla ("mergesot")

```
;;; (ordena-por-mezcla '(2 4 1 3 2 1))  =>  (1 1 2 2 3 4)
(define ordena-por-mezcla
  (lambda (l)
    (mezcla-listas (separa-en-listas-ordenadas l))))
```

- **Comparación de la ordenación de listas por inserción y por mezcla**

```
> (define l500 (crea-lista 500))
;Evaluation took 10 mSec (0 in gc) 2517 cells work, 38 bytes other
#<unspecified>
> (ordena-por-insercion l500)
;Evaluation took 19620 mSec (8860 in gc) 753497 cells work, 31 bytes other
(1 2 3 ... 498 499 500)
> (ordena-por-mezcla l500)
;Evaluation took 280 mSec (80 in gc) 22366 cells work, 31 bytes other
(1 2 3 ... 498 499 500)
```

Ordenación rápida ("quicksort")

- **Traza de ordenación rápida**

```
(ordena-rapida '(5 3 6 2 8 4))
=> (ordena-rapida-aux 5 '(3 6 2 8 4) '() '())
=> (ordena-rapida-aux 5 '(6 2 8 4) '(3) '())
=> (ordena-rapida-aux 5 '(2 8 4) '(3) '(6))
=> (ordena-rapida-aux 5 '(8 4) '(2 3) '(6))
=> (ordena-rapida-aux 5 '(4) '(2 3) '(8 6))
=> (ordena-rapida-aux 5 '() '(4 2 3) '(8 6))
=> (append (ordena-rapida '(4 2 3))
           (cons 5 (ordena-rapida '(8 6))))
=> (append '(2 3 4) (cons 5 '(6 8)))
=> (2 3 4 5 6 8)
```

Ordenación rápida ("quicksort")

```
;;; (ordena-rapida '(2 4 1 3 2 1))  =>  (1 1 2 2 3 4)
(define ordena-rapida
  (lambda (l)
    (if (or (null? l) (null? (cdr l)))
        l
        (ordena-rapida-aux (car l) (cdr l) () ()))))

(define ordena-rapida-aux
  (lambda (pivote l menores mayores)
    (cond ((null? l)
           (append (ordena-rapida menores)
                   (cons pivote (ordena-rapida mayores))))
          ((< pivote (car l))
           (ordena-rapida-aux pivote (cdr l)
                              menores
                              (cons (car l) mayores)))
          (else (ordena-rapida-aux pivote (cdr l)
                                   (cons (car l) menores)
                                   mayores)))))
```

Comparación de procedimientos de ordenación

- **Tiempo (en milésimas de segundo) para listas de distintos tamaños ordenadas de manera creciente o aleatorias**

	Creciente			Aleatoria		
ordena	250	500	1.000	250	500	1.000
por inserción	2.060	7.070	44.400	4.290	3.610	27.590
por mezcla	90	200	570	130	310	790
rapida	2.540	10.630	50.750	150	340	1.140

Ordenación genérica por inserción

```
;;; ((ordena-por-insercion-p >) '(3 1 2))  =>  (3 2 1)
;;; ((ordena-por-insercion-p string<?) '("Luis" "Ana"))  =>  ("Ana" "Luis")
(define ordena-por-insercion-p
  (lambda (pred)
    (lambda (l)
      (if (= 1 (length l))
          l
          ((inserta-p pred) (car l)
                            ((ordena-por-insercion-p pred) (cdr l)))))))

;;; ((inserta-p >) 3 '(6 2 1))  =>  (6 3 2 1)
(define inserta-p
  (lambda (pred)
    (lambda (x l)
      (cond
       ((null? l) (list x))
       ((pred x (car l)) (cons x l))
       (else (cons (car l)
                   ((inserta-p pred) x (cdr l))))))))
```

Ordenación de vectores por inserción

- **Esquema de ordenación de vectores por inserción**

```
+---+-------------------------+
| i | Vector ordenado hasta i |
+---+-------------------------+
| 0 | #(5       4 2 3)        |
| 1 | #(4 5       2 3)        |
| 2 | #(2 4 5       3)        |
| 3 | #(2 3 4 5)              |
+---+-------------------------+
```

Ordenación de vectores por inserción

- **Procedimiento de ordenación de vectores por inserción**

```
;;; (define v (vector 3 1 2))          =>  #<unspecified>
;;; v                                   =>  #(3 1 2)
;;; (ordena-vector-por-insercion! v)    =>  #(1 2 3)
;;; v                                   =>  #(1 2 3)
(define ordena-vector-por-insercion!
  (lambda (v)
    (let ((n (vector-length v)))
      (bucle 1                                      ; inicio
             n                                      ; fin
             1                                      ; incremento
             (lambda (i) (inserta-vector! i v)))    ; procedimiento
      v)))
```

Ordenación de vectores por inserción

- **Esquema de cálculo de `inserta-vector!`**

```
k = 4
v = #(1 3 4 5 2 7)
val = v[4] = 2
```

m	v[m-1]	¿val > v[m-1]?	v
4	5	No	#(1 3 4 2 5 7)
3	4	No	#(1 3 2 4 5 7)
2	3	No	#(1 2 3 4 5 7)
1	1	Sí	#(1 2 3 4 5 7)

Ordenación de vectores por inserción

- **Definición de** `inserta-vector!`

```
;;; (inserta-vector! 3 #(1 3 5 0 2 7))  =>  #(0 1 3 5 2 7)
(define inserta-vector!
  (lambda (k v)
    (let ((val (vector-ref v k)))
      (letrec
          ((inserta-aux
            (lambda (m)
              (if (zero? m)
                  (vector-set! v 0 val)
                  (let ((temp (vector-ref v (- m 1))))
                    (cond ((> val temp) (vector-set! v m val))
                          (else (vector-set! v m temp)
                                (inserta-aux (- m 1)))))))))
        (inserta-aux k)
        v))))
```

Búsqueda

- **Búsqueda lineal**

```
;;; (indice-por-busqueda-lineal #(a b c) 'b)  =>  1
;;; (indice-por-busqueda-lineal #(a b c) 'd)  =>  -1
(define indice-por-busqueda-lineal
  (lambda (v val)
    (let ((n (vector-length v)))
      (letrec ((busca-desde
                (lambda (i)
                  (cond ((= i n) -1)
                        ((equal? (vector-ref v i) val) i)
                        (else (busca-desde (+ i 1)))))))
        (busca-desde 0)))))
```

Búsqueda

- **Búsqueda binaria**

```
;;; (indice-por-busqueda-binaria #(2 3 7) 3)  =>  1
;;; (indice-por-busqueda-binaria #(2 3 7) 9)  =>  -1
(define indice-por-busqueda-binaria
    (lambda (v val)
      (letrec
          ((busca-entre
            (lambda (izquierda derecha)
              (if (< derecha izquierda)
                  -1
                  (let* ((centro (quotient (+ izquierda derecha) 2))
                         (valor-central (vector-ref v centro)))
                    (cond ((< val valor-central)
                           (busca-entre izquierda (- centro 1)))
                          ((> val valor-central)
                           (busca-entre (+ centro 1) derecha))
                          (else centro)))))))
        (busca-entre 0 (- (vector-length v) 1)))))
```

Búsqueda

- **Comparación de procedimientos de búsqueda**

```
> (define v10000 ((genera-vector (lambda (i) i)) 10000))
;Evaluation took 690 mSec (240 in gc) 80041 cells work, 40042 bytes other
#<unspecified>
> (indice-por-busqueda-lineal v10000 9999)
;Evaluation took 550 mSec (100 in gc) 40016 cells work, 33 bytes other
9999
> (indice-por-busqueda-binaria v10000 9999)
;Evaluation took 0 mSec (0 in gc) 140 cells work, 33 bytes other
9999
```

Bibliografía

- **[Springer–94]**
  Cap. 10: "Sorting and searching".