

# Ordenamientos

## Ejercicio 33:

Se ingresan en un vector 20 números reales ( float) y la computadora los ordena:

- a)por burbujeo
- b)por burbujeo mejorado (o con bandera)
- c)por inserción
- d)por selección

## EL ORDENAMIENTO DE ELEMENTOS PUEDE SER:

- **Ordenación Interna.-** En memoria principal (arrays, listas).
- **Ordenación Externa.-** En memoria secundaria. (dispositivos de almacenamiento externo.- archivos y Bases de datos).

## TIPOS DE ORDENAMIENTOS:

Los más usuales son:

### Directos:

- POR **BURBUJEO** (Compara e intercambia elementos)
- POR **SELECCIÓN** (Selecciona el más pequeño y lo intercambia)
- POR **INSERCIÓN** (Inserta los elementos en una sublista ordenada)

### Indirectos:

- ORDENACIÓN **RÁPIDA** (o Quick Sort - divide una lista en dos partes)

## COMPLEJIDAD

La complejidad del algoritmo tiene que ver el rendimiento. Para ello tenemos que identificar una operación fundamental que realice nuestro algoritmo, que en este caso es comparar. Luego contamos cuántas veces el algoritmo necesita comparar. Si en una lista de  $n$  términos realiza  $n$  comparaciones la complejidad es  $O(n)$ . Algunos ejemplos de complejidades comunes son:

- o  **$O(1)$**  : Complejidad constante.
- o  **$O(n^2)$**  : Complejidad cuadrática.
- o  **$O(n \log(n))$**  : Complejidad logarítmica.

Puede decirse que un algoritmo de complejidad  $O(n)$  es más rápido que uno de complejidad  $O(n^2)$ .

Otro aspecto a considerar es la diferencia entre el peor y el mejor caso. Cada algoritmo se comporta de modo diferente de acuerdo a cómo se le entregue la información; por eso es conveniente estudiar su comportamiento en casos extremos, como cuando los datos están prácticamente ordenados o muy desordenados.

## ORDENAMIENTO POR BURBUJEO

El bubble sort, funciona de la siguiente manera: Se va comparando cada elemento del arreglo con el siguiente; si un elemento es mayor que el que le sigue, entonces se intercambian; esto producirá que en el arreglo quede como su último elemento, el más grande. Este proceso deberá repetirse recorriendo todo el arreglo hasta que no ocurra ningún intercambio. Los elementos que van quedando ordenados ya no se comparan.

## EJEMPLO: Ordenamiento por Burbuja o bubble sort

Consiste en **comparar pares de elementos** adyacentes e **intercambiarlos entre sí hasta que estén todos ordenados**. Tenemos un array de **6 números**: {40,21,4,9,10,35}:

Primera pasada:

{21,40,4,9,10,35} <-- Se cambia el 21 por el 40.  
{21,4,40,9,10,35} <-- Se cambia el 40 por el 4.  
{21,4,9,40,10,35} <-- Se cambia el 9 por el 40.  
{21,4,9,10,40,35} <-- Se cambia el 40 por el 10.  
{21,4,9,10,35,40} <-- Se cambia el 35 por el 40.

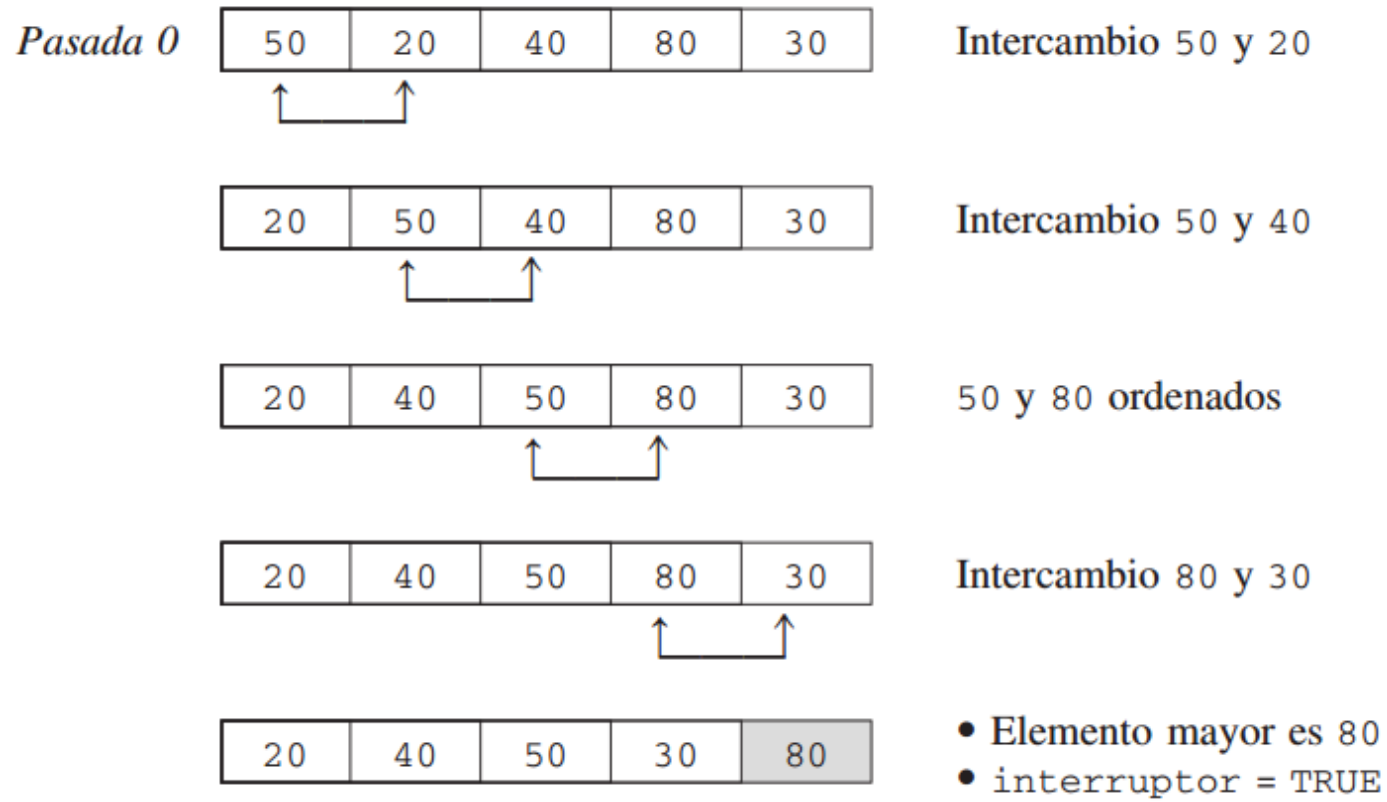
Segunda pasada:

{4,21,9,10,35,40} <-- Se cambia el 21 por el 4.  
{4,9,21,10,35,40} <-- Se cambia el 9 por el 21.  
{4,9,10,21,35,40} <-- Se cambia el 21 por el 10.

Ya están ordenados, para comprobarlo habría que hacer una tercera pasada.

## ORDENAMIENTO POR BURBUJEEO MEJORADO

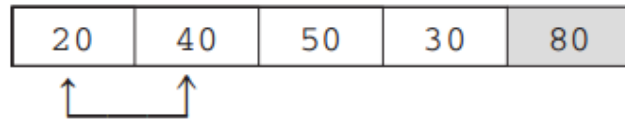
Constituye una mejora ya que el algoritmo termina inmediatamente cuando los datos ya están ordenados. Detecta que los datos ya están ordenados porque no se producen intercambios (bandera = 0 al terminar el ciclo interno). Siendo  $O(n^2)$  (Peor caso). Ejemplo:



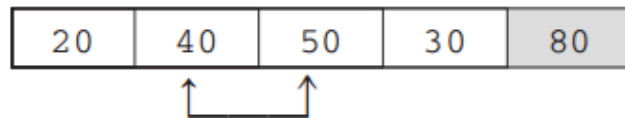


## ORDENAMIENTO POR BURBUJEEO MEJORADO

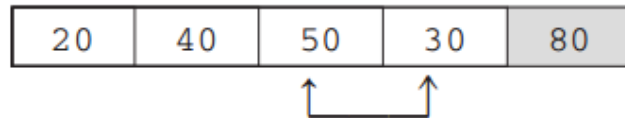
En la pasada 1:



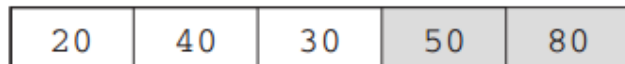
20 y 40 ordenados



40 y 50 ordenados



Se intercambian 50 y 30




- 50 y 80 elementos mayores y ordenados
- interruptor = TRUE

## ORDENAMIENTO POR BURBUJEEO MEJORADO


En la pasada 2, sólo se hacen dos comparaciones:

20	40	30	50	80
----	----	----	----	----



20 y 40 ordenados


20	30	40	50	80
----	----	----	----	----



- Se intercambian 40 y 30
- interruptor = TRUE

En la pasada 3, se hace una única comparación de 20 y 30, y no se produce intercambio:

20	30	40	50	80
----	----	----	----	----



20 y 30 ordenados

20	30	40	50	80
----	----	----	----	----

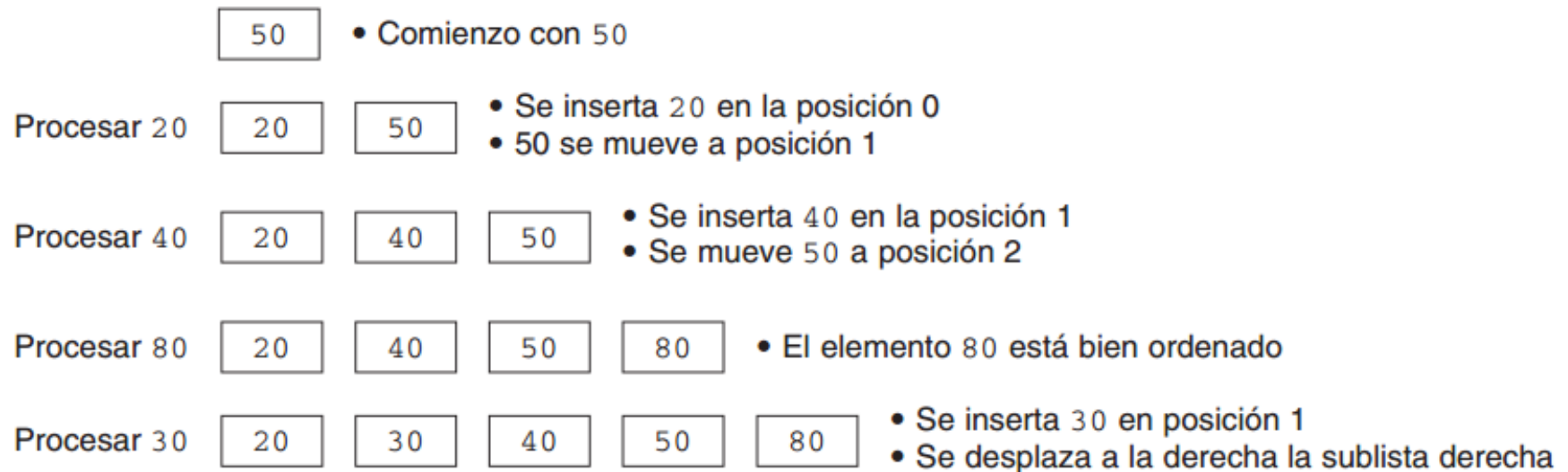
- Lista ordenada
- interruptor = FALSE

## ORDENAMIENTO POR INSERCIÓN

El insertion sort es una manera muy natural de ordenar. Consiste en tomar uno por uno los elementos de un arreglo y lo coloca en su posición con respecto a los anteriormente ordenados. Así empieza con el segundo elemento y lo ordena con respecto al primero. Luego sigue con el tercero y lo coloca en su posición ordenada con respecto a los dos anteriores, así sucesivamente hasta recorrer todas las posiciones del arreglo.

## ORDENAMIENTO POR INSERCIÓN

El método de ordenación por inserción es similar al proceso típico de ordenar tarjetas de nombres (cartas de una baraja) por orden alfabético, que consiste en insertar un nombre en su posición correcta dentro de una lista o archivo que ya está ordenado. Así el proceso en el caso de la lista de enteros  $A = 50, 20, 40, 80, 30$ .



## ORDENAMIENTO POR SELECCIÓN

Consiste en encontrar el menor de todos los elementos del arreglo o vector e intercambiarlo con el que está en la primera posición. Luego el segundo mas pequeño, y así sucesivamente hasta ordenarlo todo. Su implementación requiere  $O(n^2)$  comparaciones e intercambios para ordenar una secuencia de elementos.

## ORDENAMIENTO POR SELECCIÓN

A[0]	A[1]	A[2]	A[3]	A[4]
51	21	39	80	36

↓  
*pasada 0*

21	51	39	80	36
----	----	----	----	----

↓  
*pasada 1*

21	36	39	80	51
----	----	----	----	----

↓  
*pasada 2*

21	36	39	80	51
----	----	----	----	----

↓  
*pasada 3*

21	36	39	51	80
----	----	----	----	----

*Pasada 0.* Seleccionar 21  
Intercambiar 21 y A[0]

*Pasada 1.* Seleccionar 36  
Intercambiar 36 y A[1]

*Pasada 2.* Seleccionar 39  
Intercambiar 39 y A[2]

*Pasada 3.* Seleccionar 51  
Intercambiar 51 y A[3]

Lista ordenada

## ORDENAMIENTO RÁPIDO (QUICKSORT)

- El método se basa en dividir los  $n$  elementos de la lista a ordenar en dos partes o particiones separadas por un elemento: una *partición izquierda*, un elemento *central* denominado *pivote* o elemento de partición, y una *partición derecha*.
- La partición o división se hace de tal forma que todos los elementos de la primera sublista (partición izquierda) son menores que todos los elementos de la segunda sublista (partición derecha).
- Las dos sublistas se ordenan entonces independientemente.
- Para dividir la lista en particiones (*sublistas*) se elige uno de los elementos de la lista y se utiliza como *pivote* o *elemento de partición*.
- Si se elige una lista cualquiera con los elementos en orden aleatorio, se puede seleccionar cualquier elemento de la lista como pivote, por ejemplo, el primer elemento de la lista.

## ORDENAMIENTO RÁPIDO (QUICKSORT)

- Si la lista tiene algún orden parcial conocido, se puede tomar otra decisión para el pivote.
- Idealmente, el pivote se debe elegir de modo que se divida la lista exactamente por la mitad, de acuerdo al tamaño relativo de las claves. Por ejemplo, si se tiene una lista de enteros de 1 a 10, 5 o 6 serían pivotes ideales, mientras que 1 o 10 serían elecciones «pobres» de pivotes.
- Estas dos listas parciales se ordenan recursivamente utilizando el mismo algoritmo; es decir, se llama sucesivamente al propio algoritmo *quicksort*.
- La lista final ordenada se consigue concatenando la primera sublista, el pivote y la segunda lista, en ese orden, en una única lista.
- La primera etapa de *quicksort* es la división o «particionado» recursivo de la lista, hasta que todas las sublistas constan de sólo un elemento.



## ORDENAMIENTO RÁPIDO (QUICKSORT)

1. *Lista original*

5	2	1	7	9	3	8	7
---	---	---	---	---	---	---	---

pivote elegido

5
---

↑  
sublista izquierda1 (elementos menores que 5)

2	1	3
---	---	---

sublista derecha1 (elementos mayores o iguales a 5)

9	8	7
---	---	---

2. El algoritmo se aplica a la sublista izquierda

*Sublista Izda1*    2    1    3

↑  
*pivote*

*sublista Izda 1*

*sublista Dcha 3*

*Sublista Izda1*

*Izda*

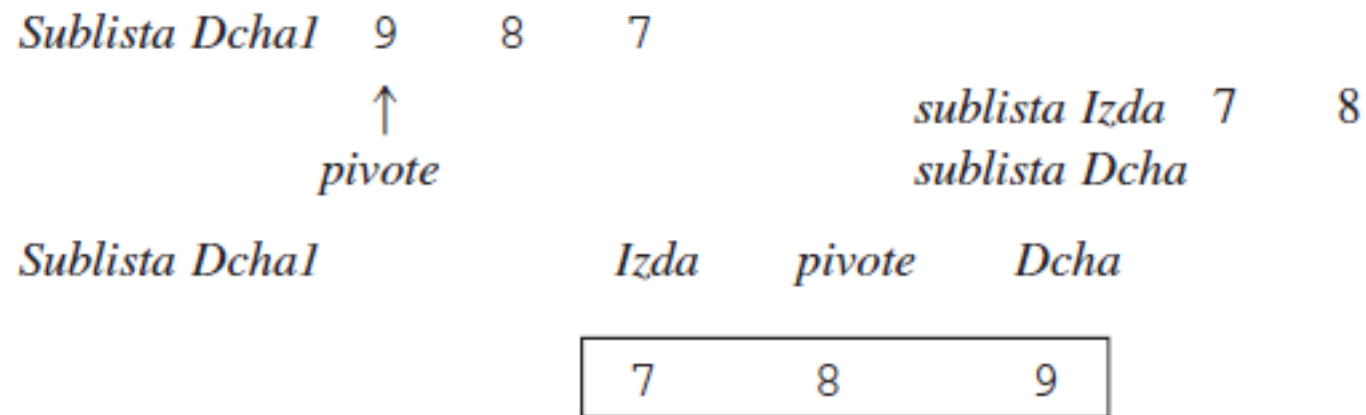
*pivote*

*Dcha*

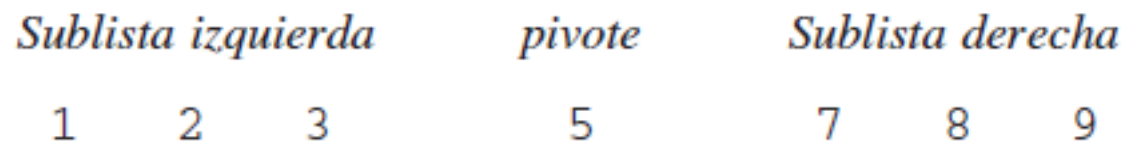
1	2	3
---	---	---

## ORDENAMIENTO RÁPIDO (QUICKSORT)

3. El algoritmo se aplica a la sublista derecha



4. *Lista ordenada final*





## ORDENAMIENTO RÁPIDO (QUICKSORT)

La etapa 2 requiere mover todos los elementos menores al pivote a la parte izquierda del array y los elementos mayores a la parte derecha. Para ello se recorre la lista de izquierda a derecha utilizando un índice  $i$  que se inicializa en la posición más baja (*inferior*) buscando un elemento mayor al pivote. También se recorre la lista de derecha a izquierda buscando un elemento menor. Para hacer esto se utilizará un índice  $j$  inicializado en la posición más alta (*superior*).

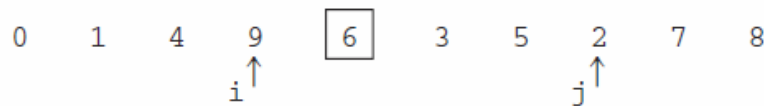
El índice  $i$  se detiene en el elemento 8 (mayor que el pivote) y el índice  $j$  se detiene en el elemento 0 (menor que el pivote).



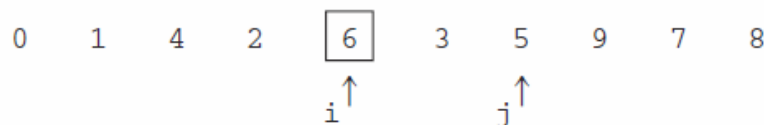
Ahora se intercambian 8 y 0 para que estos dos elementos se sitúen correctamente en cada sublista; y se incrementa el índice  $i$ , y se decrementa  $j$  para seguir los intercambios.



A medida que el algoritmo continúa,  $i$  se detiene en el elemento mayor, 9, y  $j$  se detiene en el elemento menor, 2.

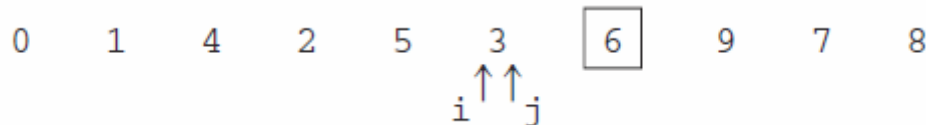


Se intercambian los elementos mientras que  $i$  y  $j$  no se crucen. En caso contrario se detiene este bucle. En el caso anterior se intercambian 9 y 2.

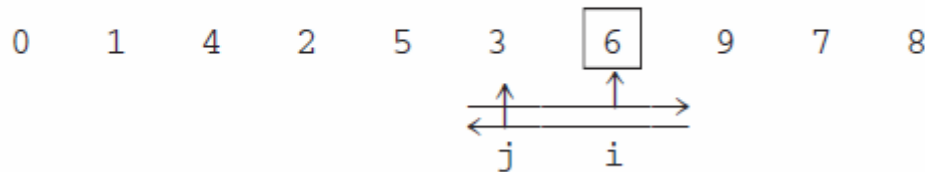


## ORDENAMIENTO RÁPIDO (QUICKSORT)

Continúa la exploración y ahora el contador  $i$  se detiene en el elemento 6 (que es el pivote) y el índice  $j$  se detiene en el elemento menor 5.



Los índices tienen actualmente los valores  $i = 5$ ,  $j = 5$ . Continúa la exploración hasta que  $i > j$ , acaba con  $i = 6$ ,  $j = 5$ .

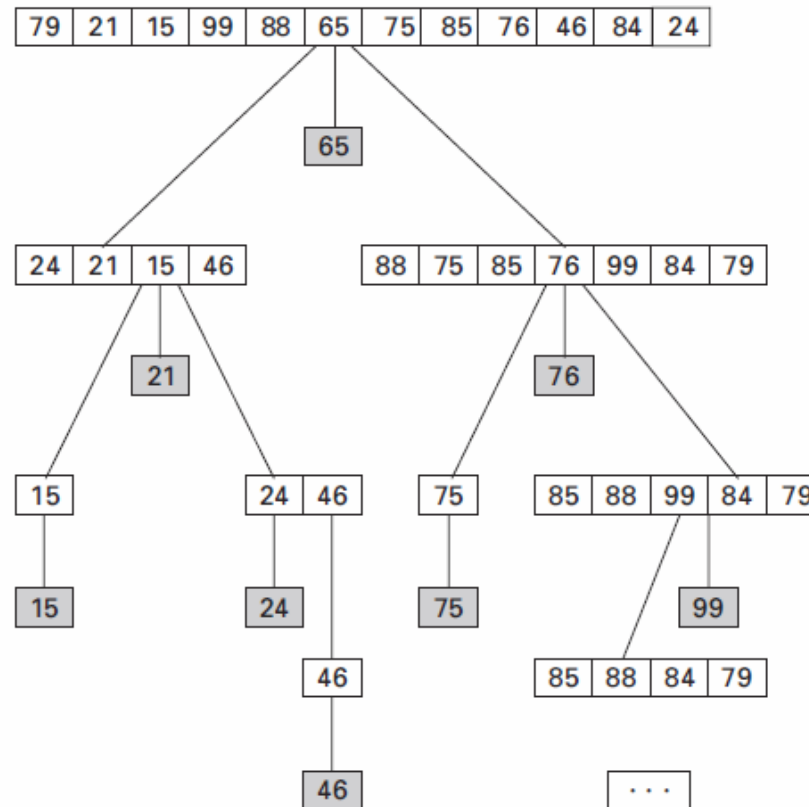


En esta posición los índices  $i$  y  $j$  han cruzado posiciones en el array y en este caso se detiene la búsqueda y no se realiza ningún intercambio ya que el elemento al que accede  $j$  está ya correctamente situado. Las dos sublistas ya han sido creadas, la lista original se ha dividido en dos particiones:

<i>sublista_izquierda</i>	<i>pivote</i>	<i>sublista_derecha</i>										
<table><tr><td>0</td><td>1</td><td>4</td><td>2</td><td>5</td><td>3</td></tr></table>	0	1	4	2	5	3	<table><tr><td>6</td></tr></table>	6	<table><tr><td>9</td><td>7</td><td>8</td></tr></table>	9	7	8
0	1	4	2	5	3							
6												
9	7	8										

## ORDENAMIENTO RÁPIDO (QUICKSORT)

El primer problema a resolver en el diseño del algoritmo de *quicksort* es seleccionar el pivote. Aunque la posición del pivote, en principio, puede ser cualquiera, una de las decisiones más ponderadas es aquella que considera el pivote como el elemento central o próximo al central de la lista. La Figura 6.2 muestra las operaciones del algoritmo para ordenar la lista  $a[]$  de  $n$  elementos enteros.



Izquierda: 24, 21, 15, 46

Pivote: 65

Derecha: 88, 75, 85, 76, 99, 84, 79

Ordenación rápida eligiendo como pivote el elemento central.

## Algoritmos de ordenamiento:

### Internos:

- Inserción directa.
- Inserción binaria.
- Selección directa.
- Burbuja.
- Shake.
- Intercambio directo.
- Shell.
- Inserción disminución incremental.
- Heap.
- Tournament.
- Ordenamiento de árbol.
- Quick sort.
- Sort particionado.
- Merge sort.
- Radix sort.
- Cálculo de dirección.

## Algoritmos de ordenamiento:

### Externos:

- Straight merging.
- Natural merging.
- Balanced multiway merging.
- Polyphase sort.
- Distribution of initial runs.

**Fin Ordenamientos**