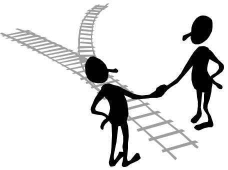
Algorítmica

Capítulo 2: Algoritmos Divide y Vencerás

Tema 5: Búsqueda y ordenación

- Algoritmos de búsqueda
- Algoritmos de ordenación
  - Ordenación por mezcla
  - Quicksort



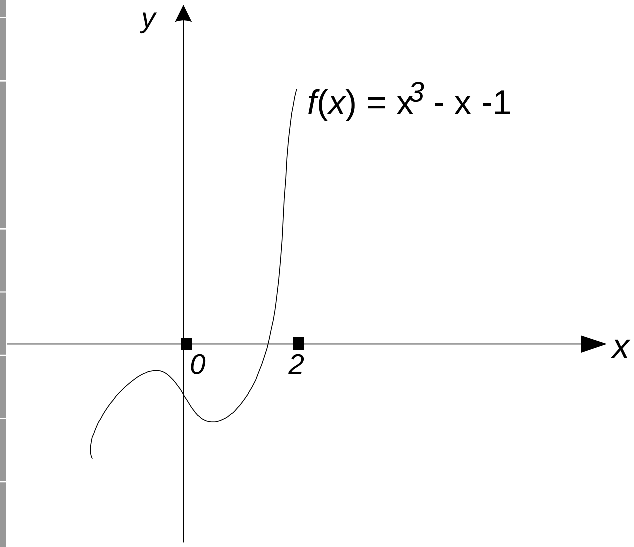
Métodos de Búsqueda y Ordenación

- Los problemas más comunes en la informática son la búsqueda y la ordenación.
  - Número de preguntas diarias en Google: 3 billones de búsquedas por día, i.e. 90 billones al mes (2015)
- Por lo tanto, la eficiencia de la búsqueda es importante
- La ordenación consiste en ordenar los elementos de un conjunto con el fin de acelerar la búsqueda.
- Los algoritmos Divide y Vencerás son idóneos para resolver buena parte de los problemas de ordenación.
- Pero, los métodos de búsqueda anteceden al uso de los computadores.

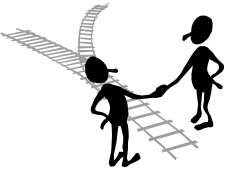
Algoritmos de busqueda: Bisección

- Encontrar la raíz de $x^3 - x - 1 = 0$, con un error ≤ 0.01

n	a_n	b_n	x_n	$f(x_n)$
1	0.0-	2.0+	1.0	-1.0
2	1.0-	2.0+	1.5	0.875
3	1.0-	1.5+	1.25	-0.296875
4				
5				
6				
7				
8	1.3125-	1.328125+	1.3203125	-0.018711



$x \approx 1.3203125$



Algoritmos de busqueda

- Búsqueda lineal (secuencial)
  - Granada
  - Palencia
  - Castellón
  - Cáceres
  - Pamplona
  - Murcia
  - Huesca
  - Santiago
  - Valladolid
  - Soria
  - Gerona
  - Guadalajara
  - Logroño
  - Bilbao

- 14 items
- Número minimo de comparaciones = 1
- Número maximo de comparaciones = 13
- En promedio: 13/2 = 6 o 7 comparaciones

No es un método muy eficiente

Por tanto no se usará ¡nunca!

- Búsqueda lineal sobre una lista ordenada
- Búsqueda binaria

Búsqueda lineal en una lista ordenada

Bilbao
Cáceres
Castellón
Gerona
Granada
Guadalajara
Huesca
Logroño
Murcia
Palencia
Pamplona
Santiago
Soria
Valladolid

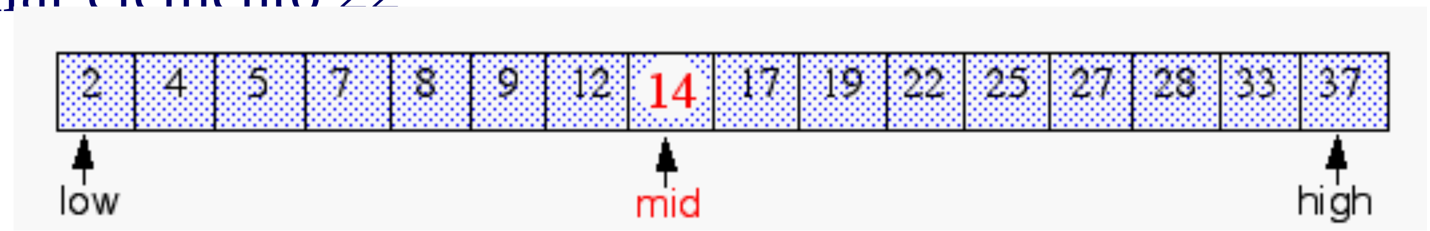
- 14 items
- Número minimo de comparaciones = 1
- Número maximo de comparaciones = 13
- Número promedio: 13/2 = 6 o 7

¿Por que entonces es mas eficiente buscar sobre una lista ordenada que en una no ordenada?

¿Deberiamos ordenar siempre la lista antes de buscar?

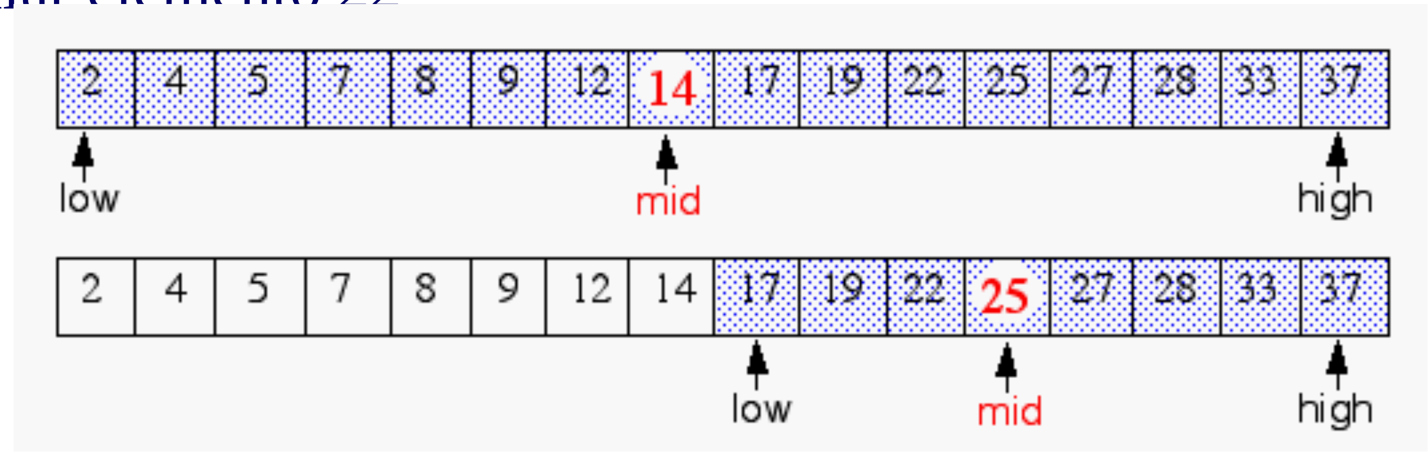
Algoritmo Búsqueda Binaria

Hallar elemento 22



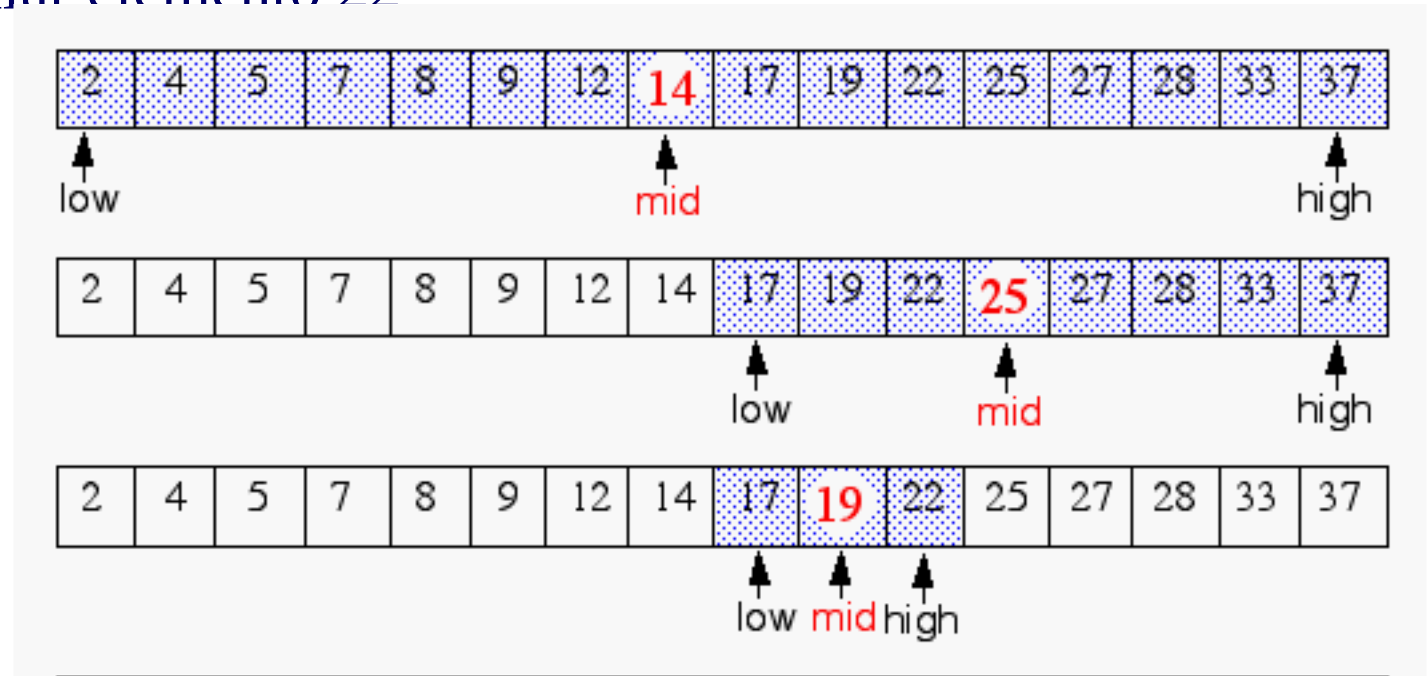
Algoritmo Búsqueda Binaria

Hallar elemento 22



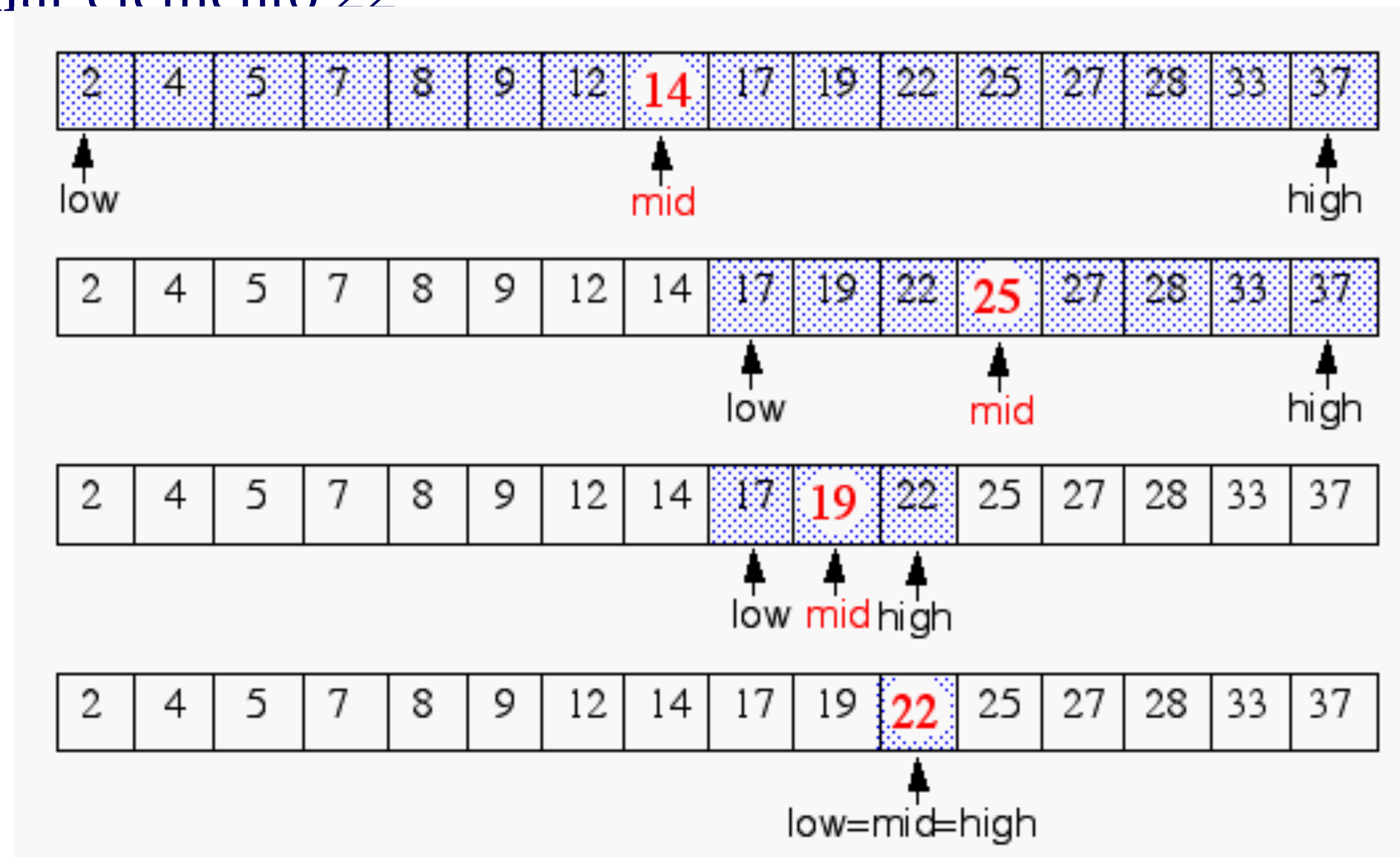
Algoritmo Búsqueda Binaria

Hallar elemento 22



Algoritmo Búsqueda Binaria

Hallar elemento 22



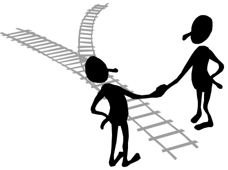
Búsqueda lineal en una lista ordenada

```
Funcion BuscaBin (T[1, 2, ..., n])
    If n = 0 or x < T[1]
    then return 0
Return Binrec (T, x)
```

```
Funcion Binrec (T[i, ...,j], x)
  if i = j then return i
  k = i + j + 1/ div 2
  if x < t[k]
  then return binrec (t[i, ..., k – 1], x)
  else return binrec (t[k, ..., j]
```

- $T(n) = O(1) \qquad\qquad \text{si } n \leq 0$

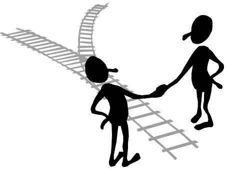
  $\qquad = T(n/2) + a \qquad \text{si } n \geq 0$
- $T(n) = c_1 \log n$, entonces $T(n)$ es $O(\log n)$.
- La búsqueda binaria mas que ser una técnica DV pura, es un caso de simplificación.



Algoritmos de Ordenación

- El esquema general de ordenación Divide y Vencerás es el siguiente

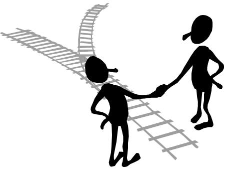
```
Algoritmo General de Ordenacion con Divide y Vencerás
Begin Algoritmo
    Iniciar Ordenar(L)
    Si L tiene longitud mayor de 1 Entonces
    Begin
            Partir la lista en dos listas, izquierda y derecha
            Iniciar Ordenar(izquierda)
            Iniciar Ordenar(derecha)
            Combinar izquierda y derecha
     End
End Algoritmo
```



Ordenacion por mezcla

- Aplicamos el método DV a la resolución del siguiente problema de ordenación
- **Problema**: Dados n elementos de la misma naturaleza, ordenarlos en orden no decreciente
- Divide y Vencerás:
  - Si n=1 terminar (toda lista de 1 elemento está ordenada)
  - Si n>1, partir la lista de elementos en dos o mas subcolecciones; ordenar cada una de ellas; combinar en una sola lista.

Pero, **¿Como hacer la partición?**

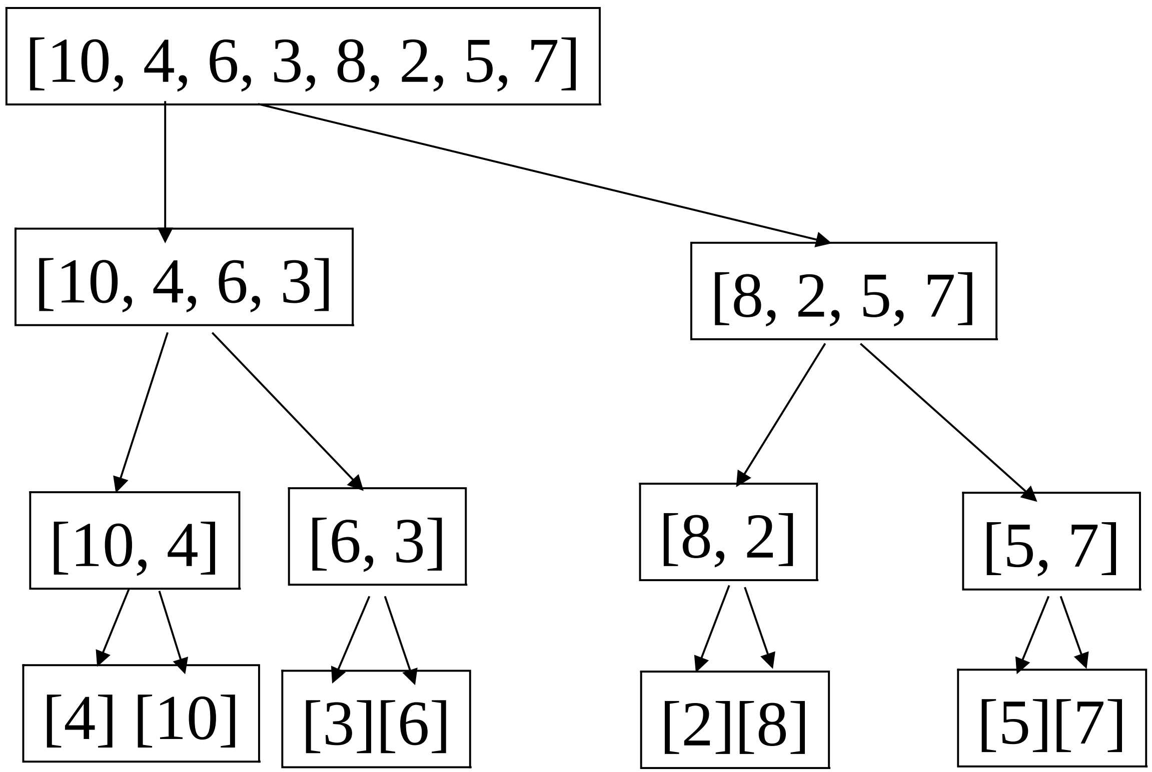


Ordenación por mezcla

- Buscamos hacer una partición equilibrada de la lista en dos partes A y B
- En A habra *n/k* elementos, y en B el resto
- Ordenamos entónces A y B recursivamente
- Combinamos las listas ordenadas A y B usando un procedimiento llamado ***mezcla***, que combina las dos listas en una sola
- El problema queda resuelto
- Las diferentes posibilidades nos las va a dar el valor k que escojamos

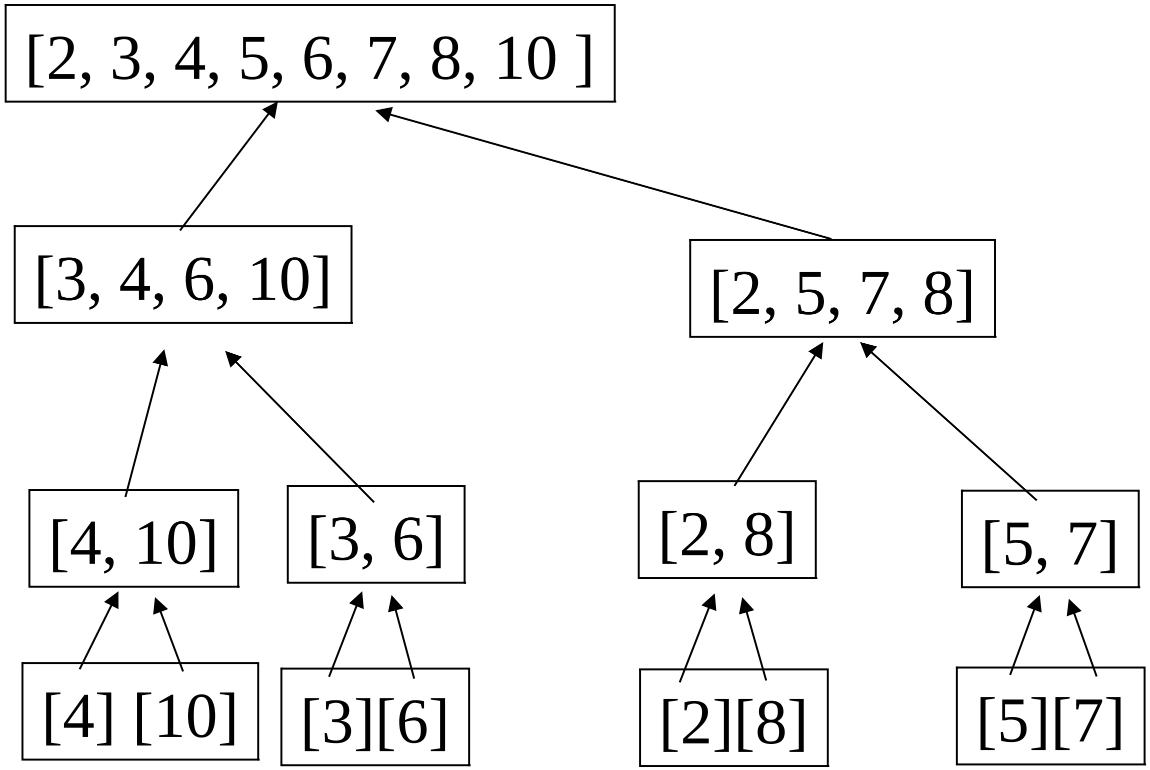
Ejemplo

- Sea k=2
- Partimos la lísta en otras dos de tamaños n/2



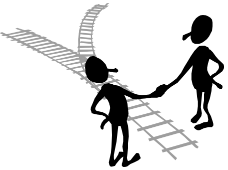
Ejemplo

- La operación de mezcla para k=2 produciría



Un código de ordenación por mezcla

```
void mergeSort(Comparable [a, int left, int right)
{
    // sort a[left:right]
    if (left < right)
    {// at least two elements
            int mid = (left+right)/2;   //midpoint
            mergeSort(a, left, mid);
            mergeSort(a, mid + 1, right);
            merge(a, b, left, mid, right); // merge from a to b
            copy(b, a, left, right);  //copy result back to a
    }
}
```



Cálculo de la eficiencia

- **Ecuación recurrente:**
- Suponemos que n es potencia de 2

$$T(n) = \begin{cases} & \text{si } n=1 \\ T(n/2) + c_2 n & \text{si } n>1,\ n=2^k \end{cases}$$

- Podemos intentar la solución por expansión

$T(n) = 2T(n/2) + c_2n;$ $\qquad$ $T(n/2) = 2T(n/4) + c_2n/2$

$T(n) = 4T(n/4) + 2\ c_2n;$ $\qquad$ $T(n) = 8T(n/8) + 3\ c_2n$

- En general,

$$T(n) = 2^iT(n/2^i) + ic_2n$$

Solucion

- Tomando $n = 2^k$, la expansión termina cuando llegamos a $T(1)$ en el lado de la derecha, lo que ocurre cuando $i=k$

$$T(n) = 2^k T(1) + kc_2 n$$

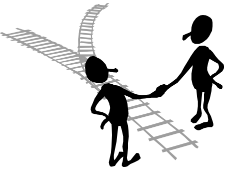
- Como $2^k = n$, entonces $k=\log n$;
- Como además

$$T(1) = c_1$$

- Tenemos

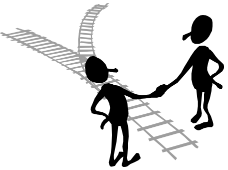
$$T(n) = c_1 n + c_2 n\log n$$

- Por tanto el tiempo para el algoritmo de ordenación por mezcla es $O(n\log n)$



Quick Sort

- También se le conoce con el nombre de Algoritmo de Hoare
- Es el algoritmo (general) de ordenación mas eficiente
- En síntesis
  - Ordena el array A eligiendo un valor clave v entre sus elementos, que actúa como pivote
  - Organiza tres secciones: izquierda, pivote y derecha
  - Todos los elementos en la izquierda son menores que el pivote, Todos los elementos en la derecha son mayores o iguales que el pivote
  - Ordena los elementos en la izquierda y en la derecha, sin requerir ninguna mezcla para combinarlos.
- Lo ideal seria que el pivote se colocara en la mediana para que la parte izquierda y la derecha tuvieran el mismo tamaño



Pseudo Codigo para quicksort

Algoritmo QUICKSORT(S,T)

// Precondición: Existe el conjunto S y es finito.

// Postcondición: los elementos de S son de la misma naturaleza, están dispuestos en una estructura lineal y son ordenables en una estructura T.

Begin Algoritmo

IF TAMAÑO(S) $\leq$ q (umbral) THEN INSERCION(S,T)

ELSE

Elegir cualquier elemento p del array como pivote

Partir S en (S1,S2,S3) de modo que

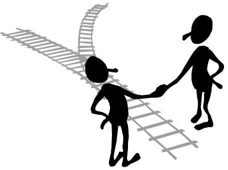
1. $\forall$ $x \in S1$, $y \in S2$, $z \in S3$ se verifique $x<p<z$ and $y=p$
2. TAMAÑO(S1) $<$ TAMAÑO(S) y TAMAÑO(S3) $<$ TAMAÑO(S)

QUICKSORT(S1,T1) // ordena recursivamente partición izquierda

QUICKSORT(S3,T3) // ordena recursivamente partición derecha

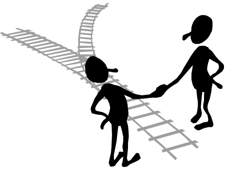
Combinación: T = T1 || S2 || T3 //S2 es el elemento intermedio entre cada mitad ordenada

End Algoritmo



La elección del pivote

- Cada uno podemos diseñar hoy mismo nuestro propio algoritmo Quicksort (otra cosa es que funcione mejor que los que ya hay...): **La elección condiciona el tiempo de ejecucion**
- El pivote puede ser cualquier elemento en el dominio, pero no necesariamente tiene que estar en S
  - Podría ser la media de los elementos seleccionados en S
  - Podría elegirse aleatoriamente, pero la funcion RAND() consume tiempo, que habría que añadirselo al tiempo total del algoritmo
- Pivotes posibles también son la mediana de un mínimo de tres elementos, o el elemento medio de S.



La elección del pivote

- El empleo de la mediana de tres elementos no tiene justificación teórica.
- Si queremos usar el concepto de mediana, deberíamos escoger como pivote la mediana del array porque lo divide en dos sub-arrays de igual tamaño
  - mediana = (n/2)º mayor elemento
  - elegir tres elementos al azar y escoger su mediana; esto suele reducir el tiempo de ejecución aproximadamente en un 5%
- Escoger como pivote el elemento en la posición central del array, dividiendo este en dos mitades

Ejemplo

array:

5	89	35	10	24	15	37	13	20	17	70

tamaño: 11

Con este ejemplo vamos a ver como funcionaría un algoritmo de este tipo “por dentro”, es decir, como se construyen los sub-problemas

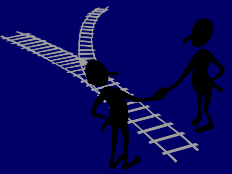
El objetivo no es conocer el algoritmo en si, de cara a posibles implementaciones, sino ilustrar el funcionamiento

Ejemplo

array:

5	89	35	14	24	15	37	13	20	7	70

“elemento pivote”



Ejemplo

array:

5	89	35	14	24	15	37	13	20	7	70

Particion esperada al final:

7	14	5	13	15	35	37	89	20	24	70

Ejemplo

	0										
array:	5	89	35	14	24	15	37	13	20	7	70
	15	89	35	14	24	5	37	13	20	7	70

Ejemplo

	0										
array:	15	89	35	14	24	5	37	13	20	7	70

k=1

Ejemplo

índice:0

array:

15	89	35	14	24	5	37	13	20	7	70

k:1

Ejemplo

índice:0

array:

15	89	35	14	24	5	37	13	20	7	70

k:2

Ejemplo

índice:0

array:

15	89	35	14	24	5	37	13	20	7	70

k:3

Ejemplo

índice:1

array:	15	14	35	89	24	5	37	13	20	7	70

k:3

Ejemplo

índice:1

array:

15	14	35	89	24	5	37	13	20	7	70

k:4

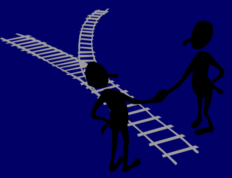
Ejemplo

índice:1

array:

15	14	35	89	24	5	37	13	20	7	70

k:5



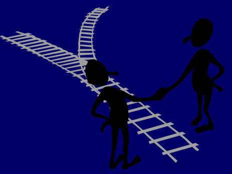
Ejemplo

índice:2

array:

15	14	5	89	24	35	37	13	20	7	70

k:5



Ejemplo

índice:2

array:

15	14	5	89	24	35	37	13	20	7	70

k:6

Ejemplo

índice:2

array:	15	14	5	89	24	35	37	13	20	7	70

k:7 ***etc...***

15 14 5 13 24 35 37 89 20 7 70

15 14 5 13 7 35 37 89 20 24 70

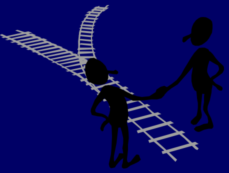
Ejemplo

índice:4

array:

15	14	5	13	7	35	37	89	20	24	70

k:11

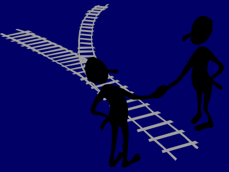


Ejemplo

Último elemento introducido

array:

15	14	5	13	7	35	37	89	20	24	70



Ejemplo

índice:4

array:											
	7	14	5	13	15	35	37	89	20	24	70

$x < 15$ $\qquad$ $15 \leq x$

Ejemplo

El pivote ahora está en la posición correcta

array:	7	14	5	13	15	35	37	89	20	24	70

$\underbrace{7 \quad 14 \quad 5 \quad 13}_{x < 15}$ $\quad 15 \quad$ $\underbrace{35 \quad 37 \quad 89 \quad 20 \quad 24 \quad 70}_{15 \leq x}$

Ejemplo

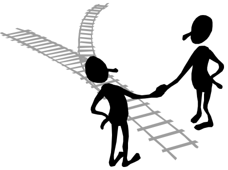
7	14	5	13	15	35	37	89	20	24	70

Ordena

Ordena

7	14	5	13

35	37	89	20	24	70



Algoritmo Quicksort (Hoare)

Procedimiento quicksort (T[i..j])

{ordena un array T[i..j] en orden creciente}

Si j-i es pequeño Entonces Insercion (T[i..j])

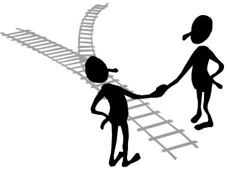
Caso contrario pivot (T[i..j], l)

{tras el pivoteo, $i \leq k < l \Rightarrow T[k] \leq T[l]$ y, $l < k \leq j \Rightarrow T[k] > T[l]$}

quicksort (T[i..l-1])

quicksort (T[l+1..j])

- No es difícil diseñar un algoritmo en tiempo lineal para el pivoteo. Sin embargo, es crucial en la práctica que la constante oculta sea pequeña.
- La mejor forma de pivotear es escoger como pivote, entre los dos primeros elementos del array, el mayor de ellos



Pivoteo lineal

- Sea $p = T[i]$ el pivote.
- Una buena forma de pivotear consiste en explorar el array $T[i..j]$ solo una vez, pero comenzando desde ambos extremos.
- Los punteros k y l se inicialízan en i y j+1 respectivamente.
- El puntero k se incrementa entonces hasta que $T[k] > p$, y el puntero l se disminuye hasta que $T[l] \leq p$. Ahora $T[k]$ y $T[l]$ están intercambiados. Este proceso continua mientras que $k < l$.
- Finalmente, $T[i]$ y $T[l]$ se intercambian para poner el pivote en su posición correcta.

Algoritmo de pivoteo

Procedimiento pivot (T[i..j])

{permuta los elementos en el array T[i..j] de tal forma que al final $i \leq l \leq j$, los elementos de T[i..l-1] no son mayores que p, T[l] = p, y los elementos de T[l+1..j] son mayores que p, donde p es el valor inicial de T[i]}

p = T[i]
k = i; l = j+1;
repetir k = k+1 hasta T[k] > p o $k \geq j$
repetir l = l -1 hasta $T[l] \leq p$
Mientras k < l hacer
    intercambiar T[k] y T[l]
    repetir k = k+1 hasta T[k] > p
    repetir l = l -1 hasta $T[l] \leq p$
intercambiar T[i] y T[l]

Ejemplo

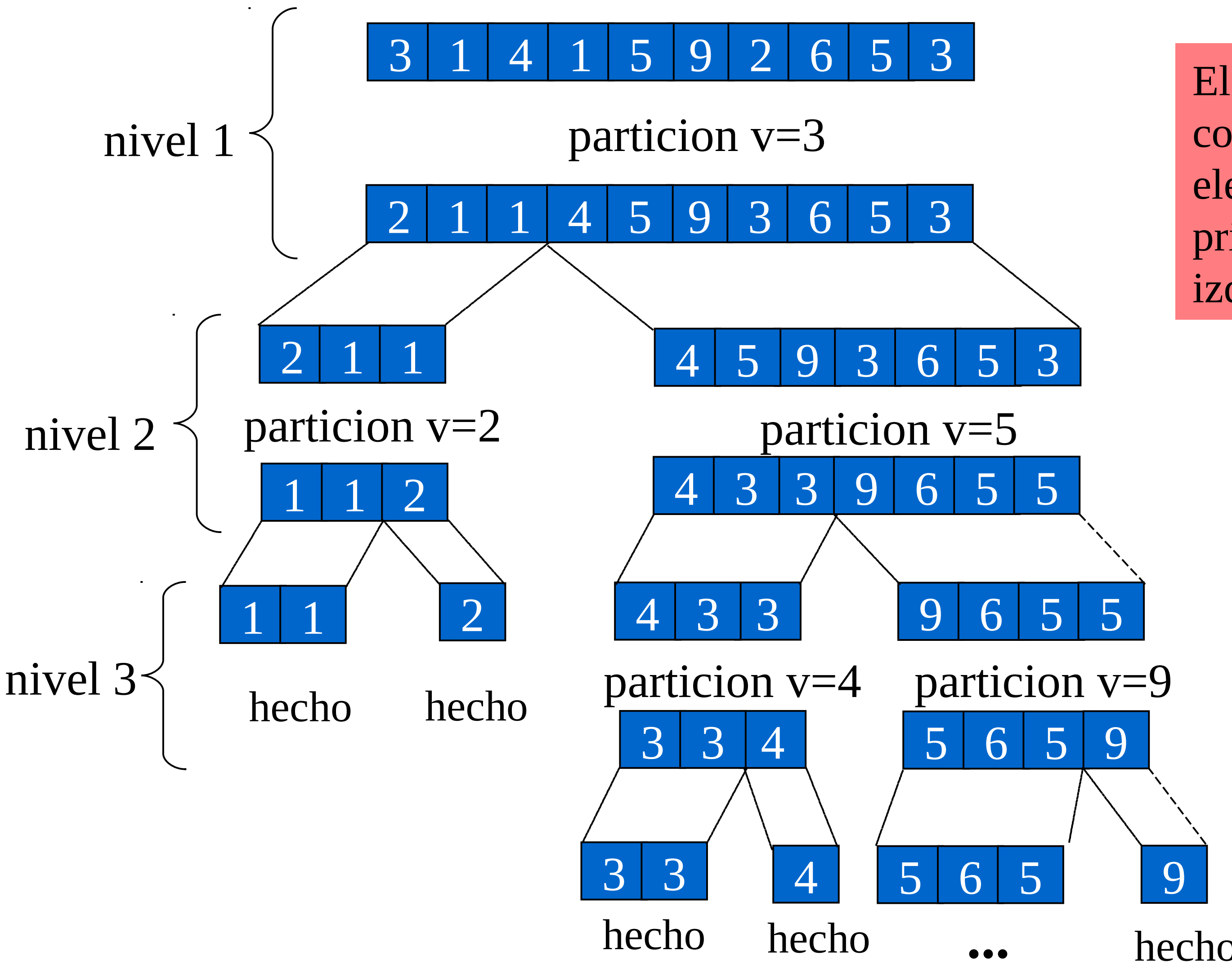
Como hemos dicho el pivote que se usa mas frecuentemente es el mayor de los dos primeros elementos del vector

Si quisieramos ordenar

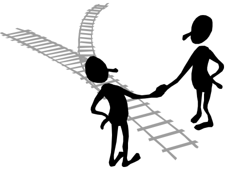
(3, 1, 4, 1, 5, 9, 2, 6, 5, 3)

El pivote sería 3, y el algoritmo se desarrollaría de la siguiente manera

Ejemplo



Elegimos el pivote como el mayor elemento de los dos primeros por la izquierda

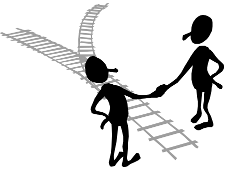


Eficiencia de quicksort

- Si admitimos que
  - El procedimiento de pivoteo es lineal,
  - Quicksort lo llamamos para T[1..n], y
  - Elegimos como peor caso que el pivote es el primer elemento del array,
- Entonces el tiempo del anterior algoritmo es

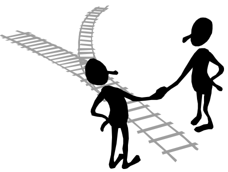
$$T(n) = T(1) + T(n-1) + an$$

- Que evidentemente proporciona un tiempo cuadrático



Análisis de Quicksort

- Recordemos que el algoritmo de ordenación por Inserción hacía aproximadamente $(1/2)n^2$ - 1/n comparaciones, es decir es $O(n^2)$ en el peor caso.
- En el peor caso quicksort es tan malo como el peor caso del método de inserción (y también de selección).
- Es que el número de intercambios que hace quicksort es unas 3 veces el número de intercambios que hace el de inserción.
- Sin embargo, en la práctica quicksort es el mejor algoritmo de ordención que se conoce...
- ¿Que pasara con el tiempo del caso promedio?



Análisis del caso promedio

- Suponemos que la lista está dada en orden aleatorio
- Suponemos que todos los posibles ordenes del array son igualmente probables
- El pivote puede ser cualquier elemento
- Puede demostrarse que en el caso promedio quicksort tiene un tiempo $T(n) = 2\text{n} \ln n + O(n)$, que se debe al número de comparaci ones que hace en promedio en una lista de n elementos
- En definitiva, quicksort, tiene un tiempo promedio O(n log n)