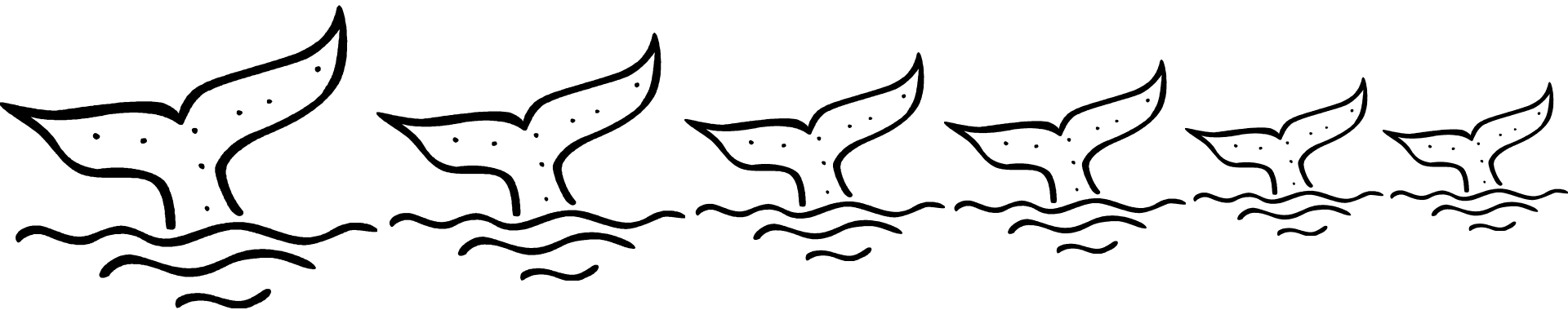


Ordenamiento (Sorting)



El problema del ordenamiento

```
proc ordenar(in/out a:seq< $T_{\leq}$ >) { $T_{\leq}$  un tipo que incluye la relación <  
  require { a = a0}  
  asegura { permutación(a,a0) && ordenado(a) }  
}  
pred ordendo(a:seq< $T_{\leq}$ >) {( $\forall i$ ) ( $\forall j$ )  $0 \leq i < j < |a| \Rightarrow a[i] < a[j]$  }  
pred permutación(a:seq<T>, b:seq<T>)  
  {( $\forall x$ ) apariciones(x,a) = apariciones(b) }
```

Uno de los problemas más clásicos, útiles y estudiados de la informática

- Variante: ordenamiento en memoria secundaria (por ejemplo archivos muy grandes que no caben en RAM)

Selection Sort

$x \in \text{subseq}(s, 0, i)$	$y \in \text{subseq}(s, i, s)$
$\leq y$	$\geq x$
ordenado	?

selectionSort(s)

i=0

while i < |s|-1

min ← seleccionarMin(s,i)

swap(s, i,min)

i++

seleccionaMin(s,i)

min=i

j=i+1

while j<|s|

if s[j]<s[min] then min=j

j++

return j

Invariante:

- los elementos entre la posición 0 y la posición i-1 se encuentran ordenados,
- los elementos entre la posición 0 y la posición i-1 son los i elementos más pequeños del arreglo original,
- El arreglo es una permutación del arreglo original (o sea, elementos entre las posiciones i y |s|-1 son los |s|-i-1 elementos más grandes del arreglo original).

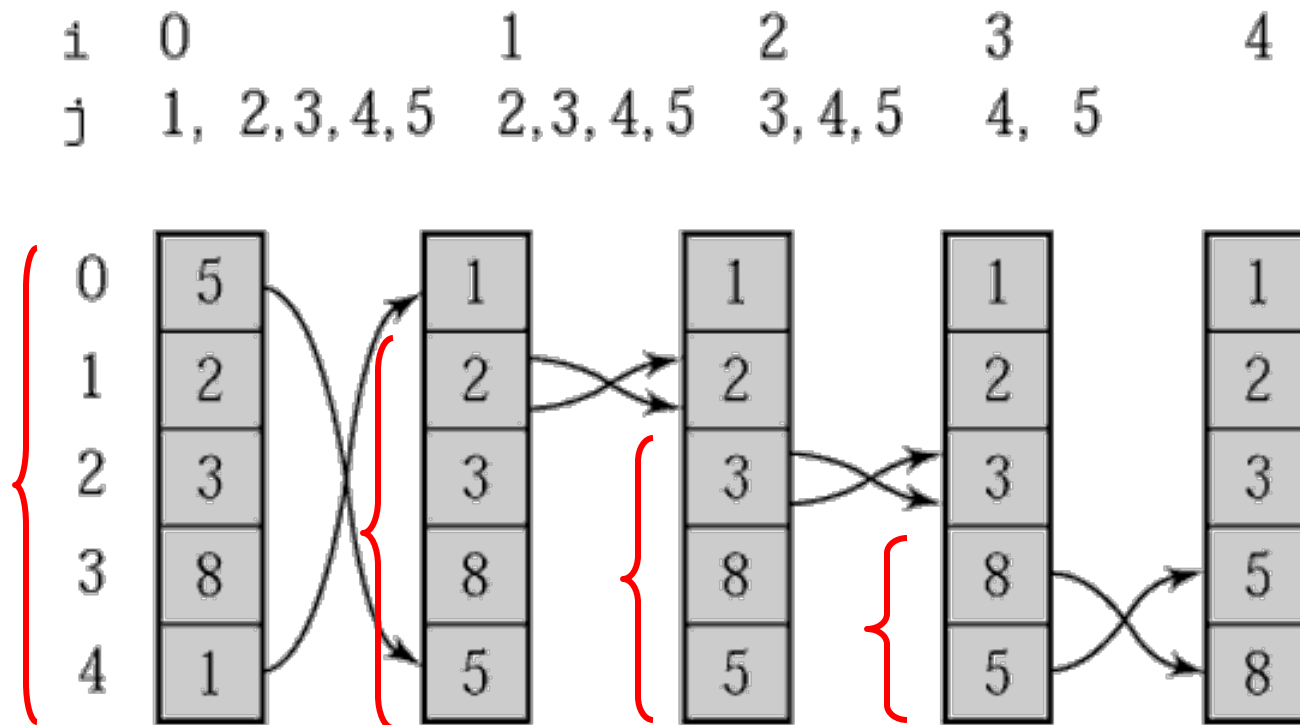
Ejercicio: escribir formalmente el invariante

Ejercicio: ¿cuál es el variante?

Selection Sort (versión recursiva)

- Para ordenar un arreglo de posiciones $i..n-1$
 - Seleccionar el mínimo elemento del arreglo
 - Ubicarlo en la posición i , intercambiándolo con el ocupante original de esa posición.
 - Ordenar a través del mismo algoritmo el arreglo de las posiciones $i+1..n-1$

Selection Sort



Ordenamiento del vector de enteros {5, 2, 3, 8, 1}

Selection Sort - Tiempo de ejecución

- ¿Cómo medimos el tiempo?
 - Cantidad de operaciones
 - Alcanza con contar cantidad de comparaciones
- Arreglo con n elementos
- $n-1$ ejecuciones del ciclo principal
- En la i -ésima iteración hay que encontrar el mínimo de entre $n-i$ elementos y por lo tanto necesitamos $n-i-1$ comparaciones

$$\text{Costo} = \sum_{i=0}^{n-2} (n - i - 1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

- Observar que el costo no depende del eventual ordenamiento parcial o total del arreglo

Insertion Sort

Invariante:

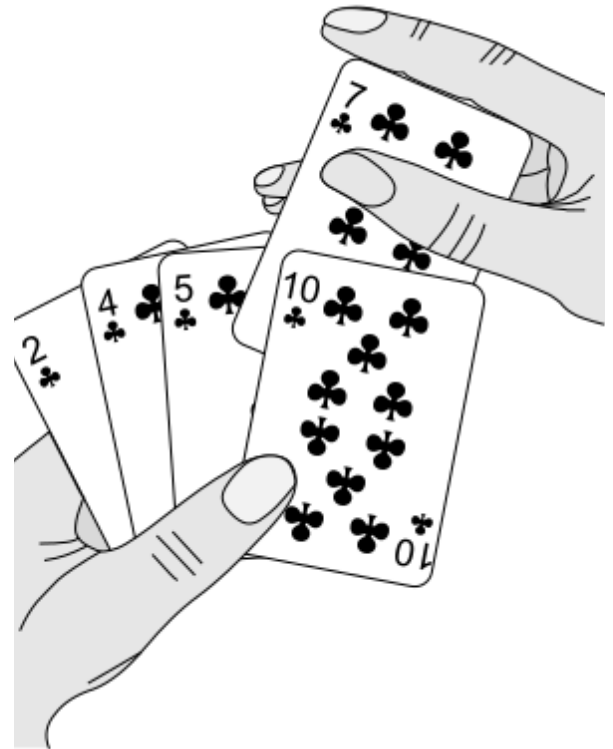
- los elementos entre la posición 0 y la posición $i-1$ son los elementos que ocupaban las posiciones 0 a i del arreglo original,
- los elementos entre la posición 0 y la posición $i-1$ se encuentran ordenados,
- El arreglo es una permutación del arreglo original (o sea que los elementos de las posiciones i hasta $n-1$ son los que ocupaban esas posiciones en el arreglo original).

(ejercicio: escribir el invariante formalmente)

Algoritmo

Repetir para las posiciones sucesivas i del arreglo:

Insertar el i -ésimo elemento en la posición que le corresponde del arreglo $0..i$



Insertion Sort

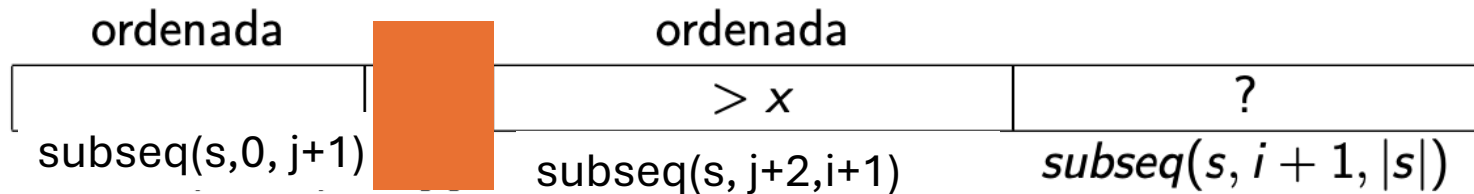
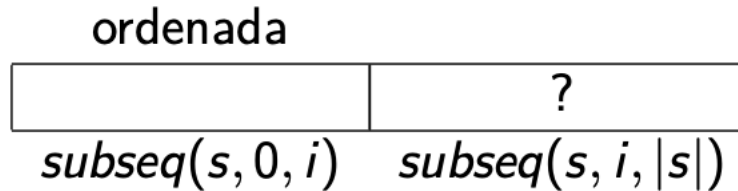
```
insertionSort(s)
```

```
  i=1
```

```
  while i < |s|
```

```
    insert(s,i)
```

```
    i++
```



```
insert(s,i)
```

```
  j=i-1, x = s[i]
```

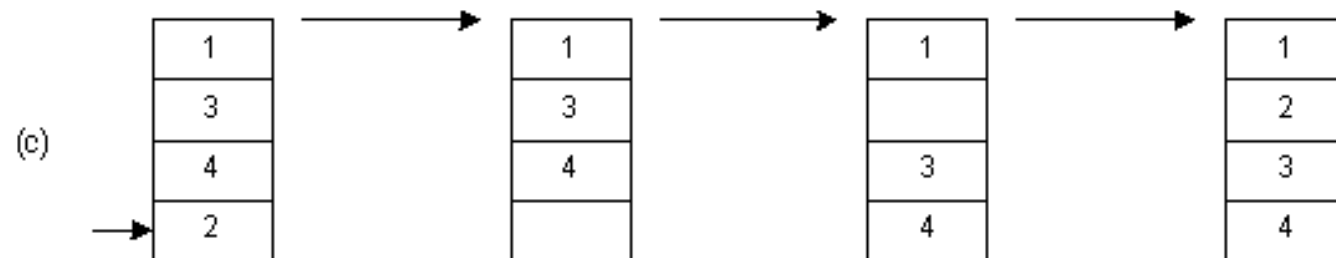
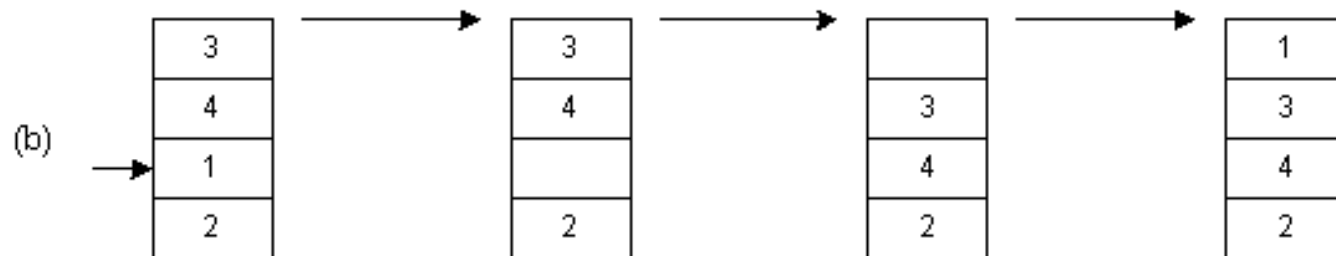
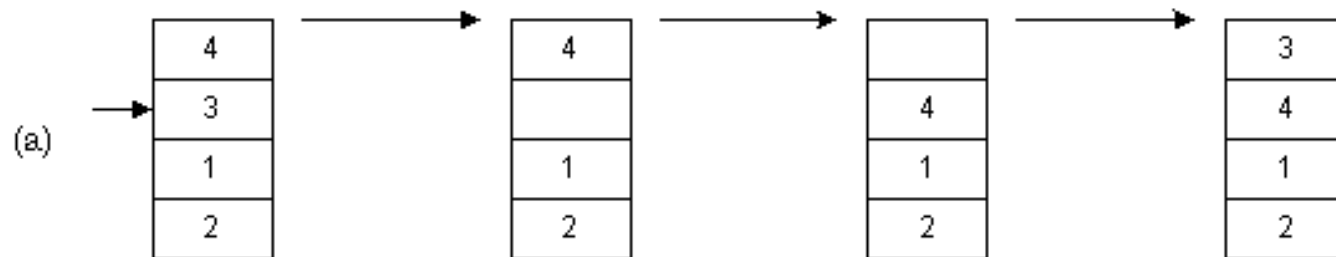
```
  while j >= 0 && s[j] > x
```

```
    s[j+1] = s[j]
```

```
    j--
```

```
  s[j+1] = x
```


Insertion Sort



Insertion Sort - Tiempo de ejecución

- Arreglo con n elementos
- $n-1$ ejecuciones del ciclo principal
- En la i -ésima iteración hay que ubicar al elemento junto a otros $i-1$ elementos y por lo tanto necesitamos $i-1$ comparaciones

$$\text{Costo} = \sum_{i=1}^{n-1} i - 1 = \frac{(n-1)(n-2)}{2}$$

- Observar que si el arreglo está parcialmente ordenado, las cosas mejoran (¿Y si está ordenado al revés?)
- Estabilidad: un algoritmo es estable si mantiene el orden anterior de elementos con igual clave.
 - ¿Para qué sirve la estabilidad?
 - ¿Son estables los algoritmos que vimos hasta ahora?

Estabilidad

Un algoritmo de ordenamiento es estable si dos registros i y j con claves iguales mantienen su orden relativo una vez ordenado el arreglo. Es decir:

Estabilidad

Un algoritmo de ordenamiento es estable si dos registros i y j con claves iguales mantienen su orden relativo una vez ordenado el arreglo. Es decir:

Antes de ordenar



Estabilidad

Un algoritmo de ordenamiento es estable si dos registros i y j con claves iguales mantienen su orden relativo una vez ordenado el arreglo. Es decir:

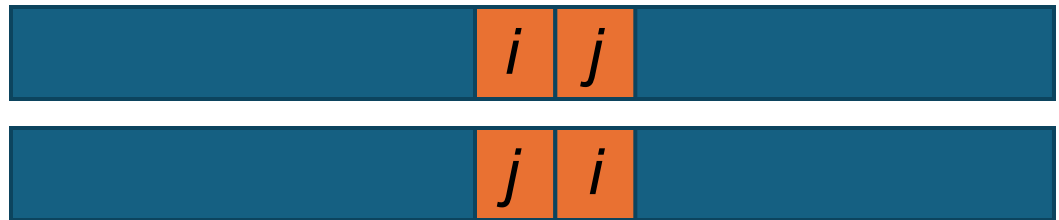
Antes de ordenar



Después de ordenar
(con algoritmo estable)



Después de ordenar
(con algoritmo inestable)



Estabilidad: ejemplo 1

Ordenando por el primer entero del par...

(9, 7)

(5, 3)

(4, 1)

(5, 2)

(9, 2)

(3, 6)

Estabilidad: ejemplo 1

Ordenando por el primer entero del par...

(9, 7)	(3, 6)
(5, 3)	(4, 1)
(4, 1)	(5, 3)
(5, 2)	(5, 2)
(9, 2)	(9, 7)
(3, 6)	(9, 2)



Algoritmo estable

Estabilidad: ejemplo 1

Ordenando por el primer entero del par...

(9, 7)	(3, 6)
(5, 3)	(4, 1)
(4, 1)	(5, 3)
(5, 2)	(5, 2)
(9, 2)	(9, 7)
(3, 6)	(9, 2)



Algoritmo estable

Estabilidad: ejemplo 1

Ordenando por el primer entero del par...

(9, 7)	(3, 6)
(5, 3)	(4, 1)
(4, 1)	(5, 3)
(5, 2)	(5, 2)
(9, 2)	(9, 7)
(3, 6)	(9, 2)



Algoritmo estable

Estabilidad: ejemplo 1

Ordenando por el primer entero del par...

(9, 7)	(3, 6)	(3, 6)
(5, 3)	(4, 1)	(4, 1)
(4, 1)	(5, 3)	(5, 3)
(5, 2)	(5, 2)	(5, 2)
(9, 2)	(9, 7)	(9, 2)
(3, 6)	(9, 2)	(9, 7)



Algoritmo estable

Algoritmo inestable

Estabilidad: ejemplo 1

Ordenando por el primer entero del par...

(9, 7)	(3, 6)	(3, 6)
(5, 3)	(4, 1)	(4, 1)
(4, 1)	(5, 3)	(5, 3)
(5, 2)	(5, 2)	(5, 2)
(9, 2)	(9, 7)	(9, 2)
(3, 6)	(9, 2)	(9, 7)



Algoritmo estable

Algoritmo inestable

Estabilidad: ejemplo 1

Ordenando por el primer entero del par...

(9, 7)	(3, 6)	(3, 6)
(5, 3)	(4, 1)	(4, 1)
(4, 1)	(5, 3)	(5, 3)
(5, 2)	(5, 2)	(5, 2)
(9, 2)	(9, 7)	(9, 2)
(3, 6)	(9, 2)	(9, 7)



Algoritmo estable

Algoritmo inestable

Estabilidad: ejemplo 2

Arreglo ordenado alfabéticamente por nombres

(Bruno, TM)

(Federico, TM)

(Florencia, TT)

(Leandro, TT)

(Martina, TT)

(Valentina, TM)

Estabilidad: ejemplo 2

Ordenar **por turno**

(Bruno, TM)		(Bruno, TM)
(Federico, TM)		(Federico, TM)
(Florencia, TT)		(Valentina, TM)
(Leandro, TT)	→	(Florencia, TT)
(Martina, TT)		(Leandro, TT)
(Valentina, TM)		(Martina, TT)

Algoritmo estable

Estabilidad: ejemplo 2

Ordenar **por turno**

(Bruno, TM)

(Federico, TM)

(Florencia, TT)

(Leandro, TT)

(Martina, TT)

(Valentina, TM)

(Bruno, TM)

(Federico, TM)

(Valentina, TM)

(Florencia, TT)

(Leandro, TT)

(Martina, TT)

Algoritmo estable

(Valentina, TM)

(Bruno, TM)

(Federico, TM)

(Martina, TT)

(Leandro, TT)

(Florencia, TT)

Algoritmo inestable

Estabilidad - Selection Sort

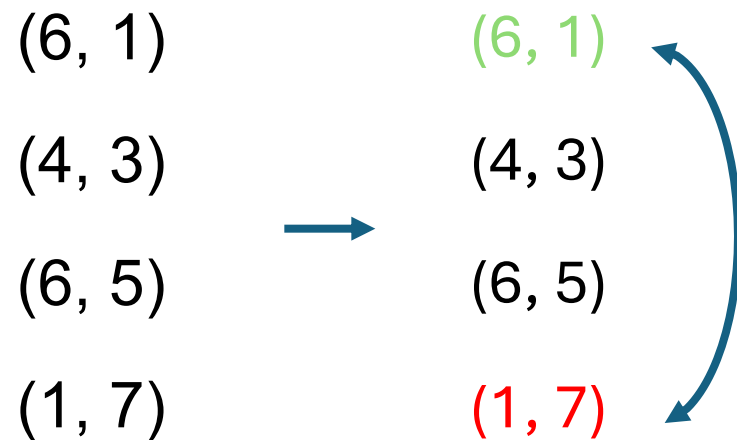
(6, 1)

(4, 3)

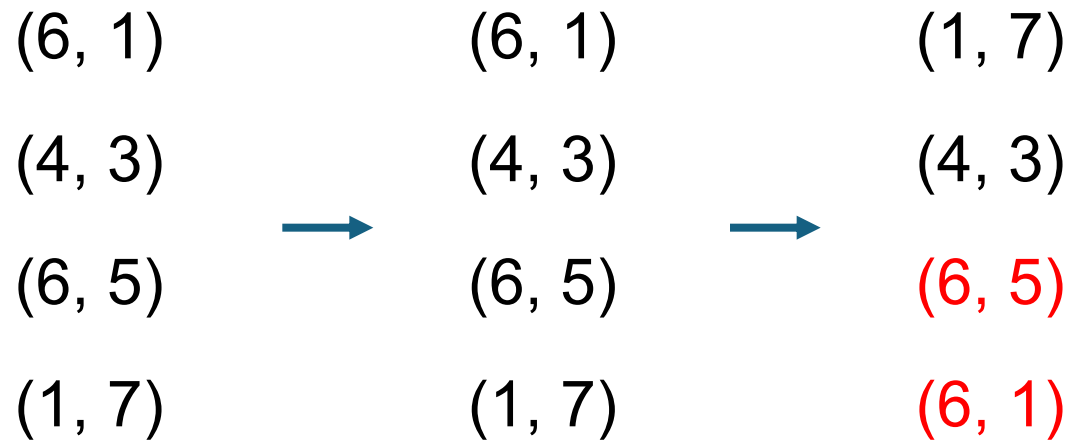
(6, 5)

(1, 7)

Estabilidad - Selection Sort



Estabilidad - Selection Sort



Algoritmo inestable pero...

Merge Sort

Clásico ejemplo de la metodología “Divide & Conquer” (o “Divide y Reinarás”)

La metodología consiste en

- Dividir un problema en problemas similares....pero más chicos
- Resolver los problemas menores
- Combinar las soluciones de los problemas menores para obtener la solución del problema original.

El método tiene sentido siempre y cuando la división y la combinación no sean excesivamente caras

- Algoritmo atribuido por Knuth a Von Neumann (1945)

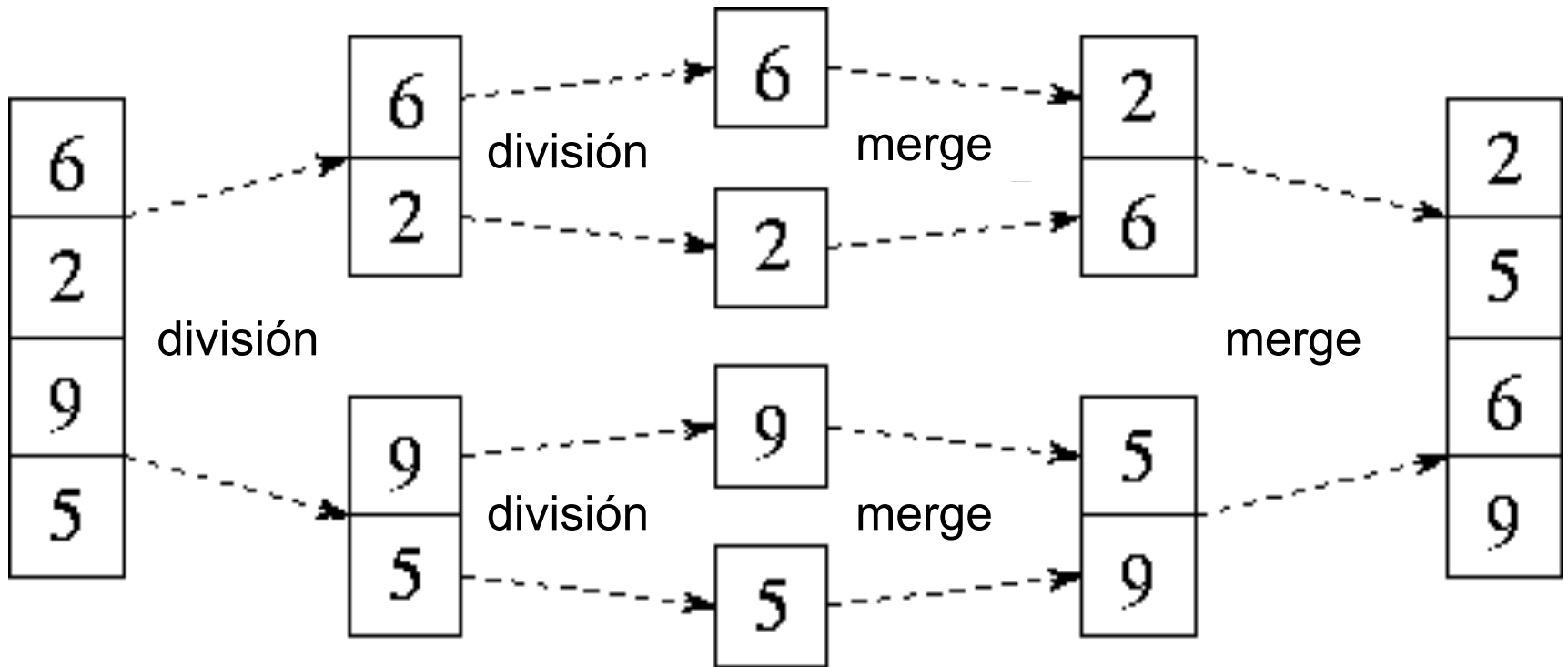
Merge Sort

Algoritmo

- Si $n < 2$ entonces el arreglo ya está ordenado
- En caso contrario
 - Dividir el arreglo en dos partes iguales (o sea ¡por la mitad!)
 - Ordenar recursivamente (o sea a través del mismo algoritmo) ambas mitades.
 - “Aperear” ambas mitades (ya ordenadas) en un único arreglo

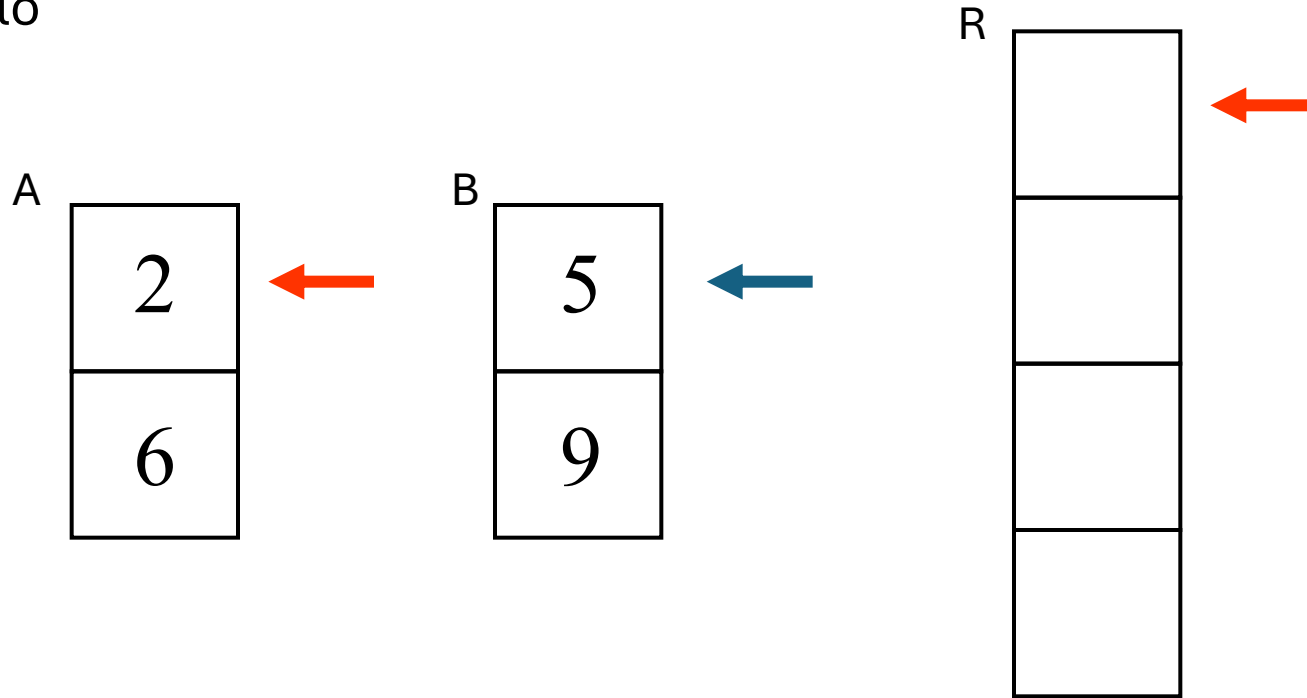
Merge Sort

- Ejemplo



Merge Sort

- Ejemplo

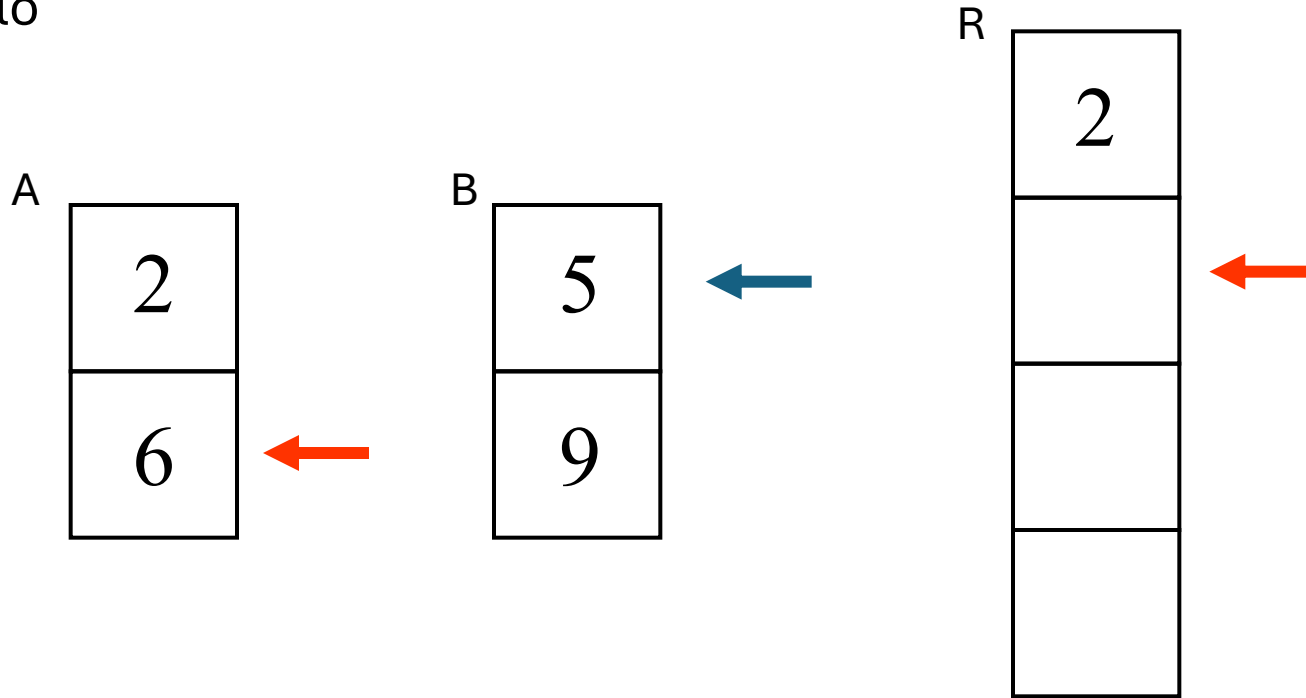


$n=4$

$\#pasos=0$

Merge Sort

- Ejemplo

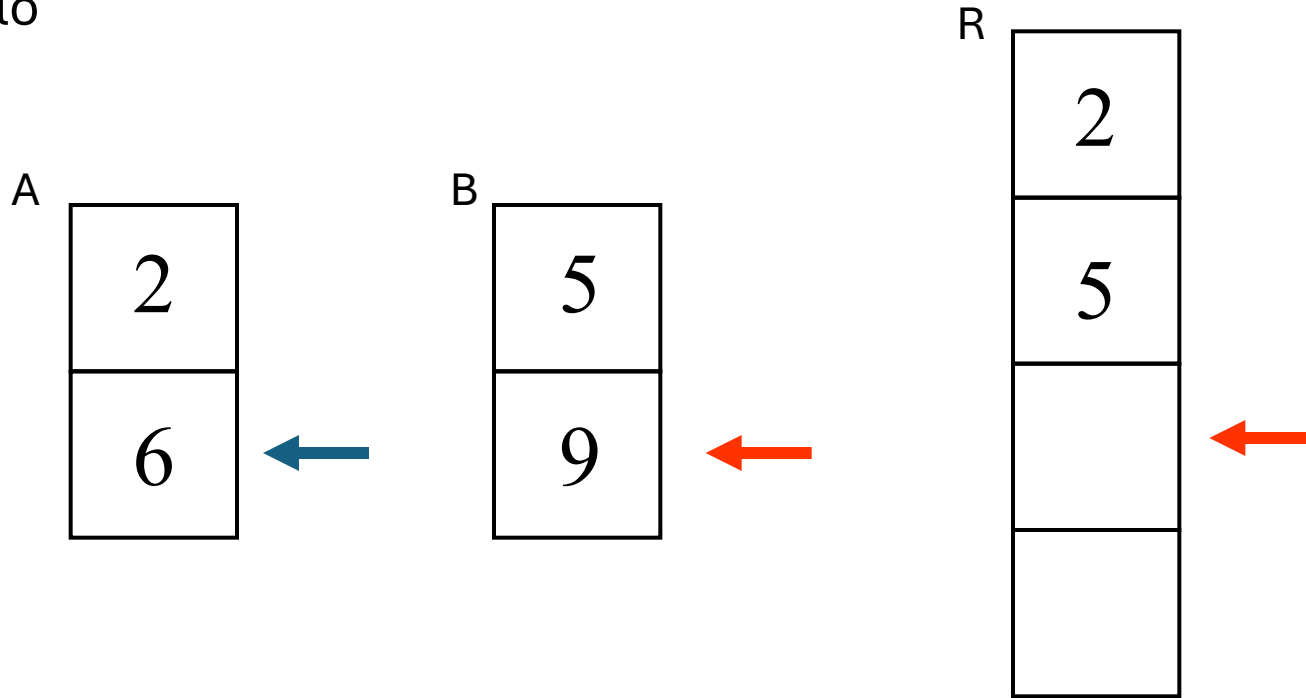


$n=4$

$\#pasos=1$

Merge Sort

- Ejemplo

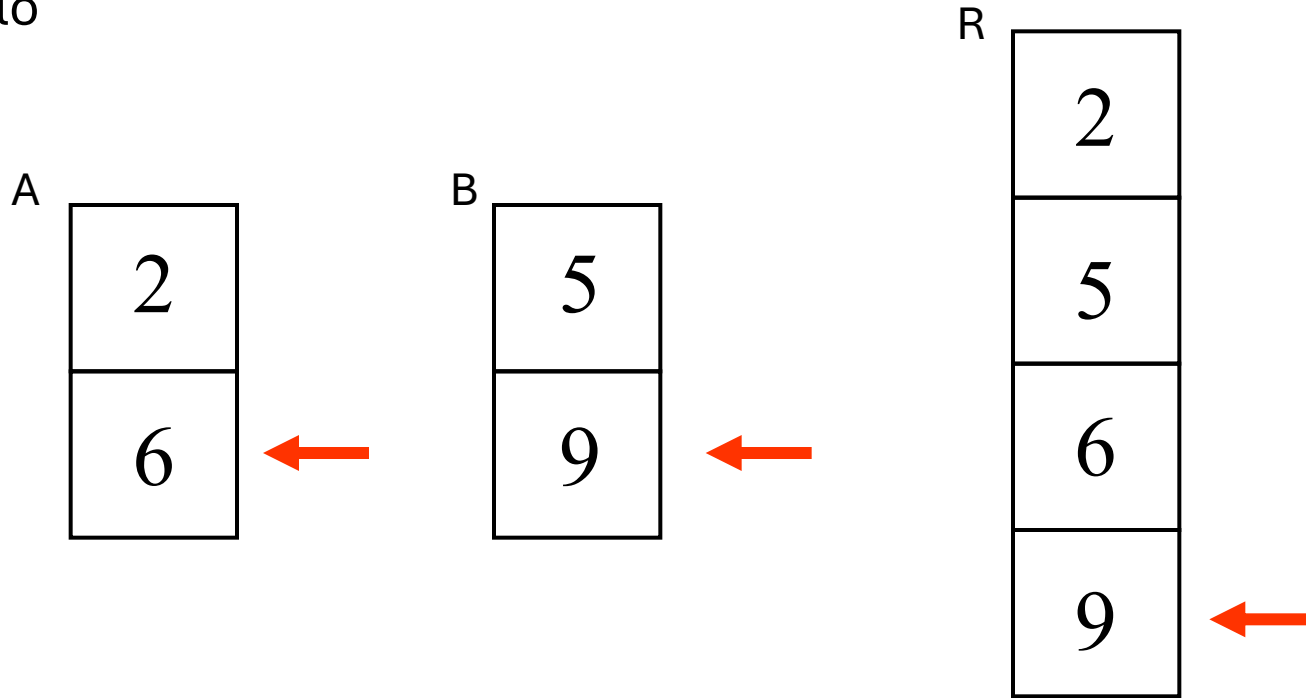


$n=4$

$\#pasos=2$

Merge Sort

- Ejemplo



$n=4$

$\#pasos=3$

Merge Sort

- Análisis

MERGE-SORT $A [0 \dots n - 1]$

1. Si $n=1$, ya está ordenado.
2. Recursivamente ordenar $A [0 \dots n/2-1]$
y $A [n/2 \dots n - 1]$.
3. *Merge* las dos mitades ordenadas

Merge Sort

- Análisis

$T(n)$ **MERGE-SORT** $A [0 \dots n - 1]$

$O(1)$ 1. Si $n=1$, ya está ordenado.

$2T(n/2)$ 2. Recursivamente ordenar $A [0 \dots n/2-1]$
y $A [n/2 \dots n - 1]$.

$O(n-1)$ 3. *Merge* las dos mitades ordenadas

Merge Sort

- Análisis

$T(n)$ **MERGE-SORT** $A [0 \dots n - 1]$

$O(1)$ 1. Si $n=1$, ya está ordenado.

$2T(n/2)$ 2. Recursivamente ordenar $A [0 \dots n/2-1]$
y $A [n/2 \dots n - 1]$.

$O(n-1)$ 3. *Merge* las dos mitades ordenadas

$$T(n) = \begin{cases} O(1) & \text{si } n=1 \\ 2T(n/2) + O(n-1) & \text{si } n>1 \end{cases}$$


Merge Sort

■ Análisis

$$T(n) = 2T\left(\frac{n}{2}\right) + (n - 1) =$$


Merge Sort

■ Análisis

$$T(n) = 2T\left(\frac{n}{2}\right) + (n - 1) =$$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \left(\frac{n}{2} - 1\right)$$

Merge Sort

■ Análisis


$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + (n - 1) = \\ &= 2\left(2T\left(\frac{n}{4}\right) + \left(\frac{n}{2} - 1\right)\right) + (n - 1) = \end{aligned}$$


Merge Sort

■ Análisis

$$\begin{aligned}T(n) &= 2T\left(\frac{n}{2}\right) + (n - 1) = \\&= 2\left(2T\left(\frac{n}{4}\right) + \left(\frac{n}{2} - 1\right)\right) + (n - 1) = \\&= 4T\left(\frac{n}{4}\right) + 2n - 2 - 1 =\end{aligned}$$

Merge Sort

■ Análisis

$$\begin{aligned}T(n) &= 2T\left(\frac{n}{2}\right) + (n - 1) = \\&= 2\left(2T\left(\frac{n}{4}\right) + \left(\frac{n}{2} - 1\right)\right) + (n - 1) = \\&= 4T\left(\frac{n}{4}\right) + 2n - 2 - 1 =\end{aligned}$$

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \left(\frac{n}{4} - 1\right)$$

Merge Sort

■ Análisis

$$\begin{aligned}T(n) &= 2T\left(\frac{n}{2}\right) + (n - 1) = \\&= 2\left(2T\left(\frac{n}{4}\right) + \left(\frac{n}{2} - 1\right)\right) + (n - 1) = \\&= 4T\left(\frac{n}{4}\right) + 2n - 2 - 1 = \quad T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \left(\frac{n}{4} - 1\right) \\&= 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4} - 1\right) + 2n - 2 - 1 =\end{aligned}$$

Merge Sort

■ Análisis

$$\begin{aligned}T(n) &= 2T\left(\frac{n}{2}\right) + (n - 1) = \\&= 2\left(2T\left(\frac{n}{4}\right) + \left(\frac{n}{2} - 1\right)\right) + (n - 1) = \\&= 4T\left(\frac{n}{4}\right) + 2n - 2 - 1 = \\&= 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4} - 1\right) + 2n - 2 - 1 = \\&= 8T\left(\frac{n}{8}\right) + 3n - 4 - 2 - 1 =\end{aligned}$$

Merge Sort

■ Análisis

$$\begin{aligned}T(n) &= 2T\left(\frac{n}{2}\right) + (n - 1) = \\&= 2\left(2T\left(\frac{n}{4}\right) + \left(\frac{n}{2} - 1\right)\right) + (n - 1) = \\&= 4T\left(\frac{n}{4}\right) + 2n - 2 - 1 = \\&= 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4} - 1\right) + 2n - 2 - 1 = \\&= 8T\left(\frac{n}{8}\right) + 3n - 4 - 2 - 1 = \\&= \dots = \\&= 2^i T\left(\frac{n}{2^i}\right) + i n - 2^{i-1} - \dots - 2 - 1 = \\&= \dots =\end{aligned}$$

Merge Sort

■ Análisis

$$T(n) = 2T\left(\frac{n}{2}\right) + (n - 1) =$$

$$= 2\left(2T\left(\frac{n}{4}\right) + \left(\frac{n}{2} - 1\right)\right) + (n - 1) =$$

$$= 4T\left(\frac{n}{4}\right) + 2n - 2 - 1 =$$

$$= 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4} - 1\right) + 2n - 2 - 1 =$$

$$= 8T\left(\frac{n}{8}\right) + 3n - 4 - 2 - 1 =$$

$$= \dots =$$

$$= 2^i T\left(\frac{n}{2^i}\right) + i n - 2^{i-1} - \dots - 2 - 1 = \quad \text{¿Hasta cuándo?}$$

$$= \dots =$$

Merge Sort

■ Análisis

$$T(n) = 2T\left(\frac{n}{2}\right) + (n - 1) =$$

$$= 2\left(2T\left(\frac{n}{4}\right) + \left(\frac{n}{2} - 1\right)\right) + (n - 1) =$$

$$= 4T\left(\frac{n}{4}\right) + 2n - 2 - 1 =$$

$$= 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4} - 1\right) + 2n - 2 - 1 =$$

$$= 8T\left(\frac{n}{8}\right) + 3n - 4 - 2 - 1 =$$

$$= \dots =$$

$$= 2^i T\left(\frac{n}{2^i}\right) + i n - 2^{i-1} - \dots - 2 - 1 = \quad \text{Hasta } 2^i = n \text{ o sea } i = \log n$$

$$= \dots =$$

Merge Sort

■ Análisis

$$T(n) = 2T\left(\frac{n}{2}\right) + (n - 1) =$$

$$= 2^i T\left(\frac{n}{2^i}\right) + i n - 2^{i-1} - \dots - 2 - 1 = \quad \text{Hasta } 2^i = n \text{ o sea } i = \log n$$

$$= 2^{\log n} T\left(\frac{n}{2^{\log n}}\right) + \log n n - 2^{\log n - 1} - 2^{\log n - 2} - \dots - 2 - 1 =$$

Merge Sort

■ Análisis

$$T(n) = 2T\left(\frac{n}{2}\right) + (n - 1) =$$

$$= 2^i T\left(\frac{n}{2^i}\right) + i n - 2^{i-1} - \dots - 2 - 1 = \quad \text{Hasta } 2^i = n \text{ o sea } i = \log n$$

$$= 2^{\log n} T\left(\frac{n}{2^{\log n}}\right) + \log n n - 2^{\log n-1} - 2^{\log n-2} - \dots - 2 - 1 =$$

$$= 2^{\log n} T(1) + \log n n - 2^{\log n-1} - 2^{\log n-2} - \dots - 2 - 1 =$$

Merge Sort

■ Análisis

$$T(n) = 2T\left(\frac{n}{2}\right) + (n - 1) =$$

$$= 2^i T\left(\frac{n}{2^i}\right) + i n - 2^{i-1} - \dots - 2 - 1 = \quad \text{Hasta } 2^i = n \text{ o sea } i = \log n$$

$$= 2^{\log n} T\left(\frac{n}{2^{\log n}}\right) + \log n n - 2^{\log n-1} - 2^{\log n-2} - \dots - 2 - 1 =$$

$$= 2^{\log n} T(1) + \log n n - 2^{\log n-1} - 2^{\log n-2} - \dots - 2 - 1 =$$

$$= 2^{\log n} + \log n n - 2^{\log n-1} - 2^{\log n-2} - \dots - 2 - 1 =$$

Merge Sort

■ Análisis

$$T(n) = 2T\left(\frac{n}{2}\right) + (n - 1) =$$

$$= 2^i T\left(\frac{n}{2^i}\right) + i n - 2^{i-1} - \dots - 2 - 1 = \quad \text{Hasta } 2^i = n \text{ o sea } i = \log n$$

$$= 2^{\log n} T\left(\frac{n}{2^{\log n}}\right) + \log n n - 2^{\log n-1} - 2^{\log n-2} - \dots - 2 - 1 =$$

$$= 2^{\log n} T(1) + \log n n - 2^{\log n-1} - 2^{\log n-2} - \dots - 2 - 1 =$$

$$= 2^{\log n} + \log n n - 2^{\log n-1} - 2^{\log n-2} - \dots - 2 - 1 =$$

$$= O(n \log n)$$

Merge Sort

■ Análisis

$$T(n) = 2T\left(\frac{n}{2}\right) + (n - 1) =$$

$$= 2^i T\left(\frac{n}{2^i}\right) + i n - 2^{i-1} - \dots - 2 - 1 = \text{Hasta } 2^i = n \text{ o sea } i = \log n$$

$$= 2^{\log n} T\left(\frac{n}{2^{\log n}}\right) + \log n n - 2^{\log n-1} - 2^{\log n-2} - \dots - 2 - 1 =$$

$$= 2^{\log n} T(1) + \log n n - 2^{\log n-1} - 2^{\log n-2} - \dots - 2 - 1 =$$

$$= 2^{\log n} + \log n n - 2^{\log n-1} - 2^{\log n-2} - \dots - 2 - 1 =$$

$$= O(n \log n)$$

Esto suponiendo que $n = 2^k$, pero ¿y si no fuera exactamente así?

Quick Sort

Idea *en cierto modo* parecida....(D&Q)

Debido a C.A.R. Hoare (sí, ¡el de las triplas!)

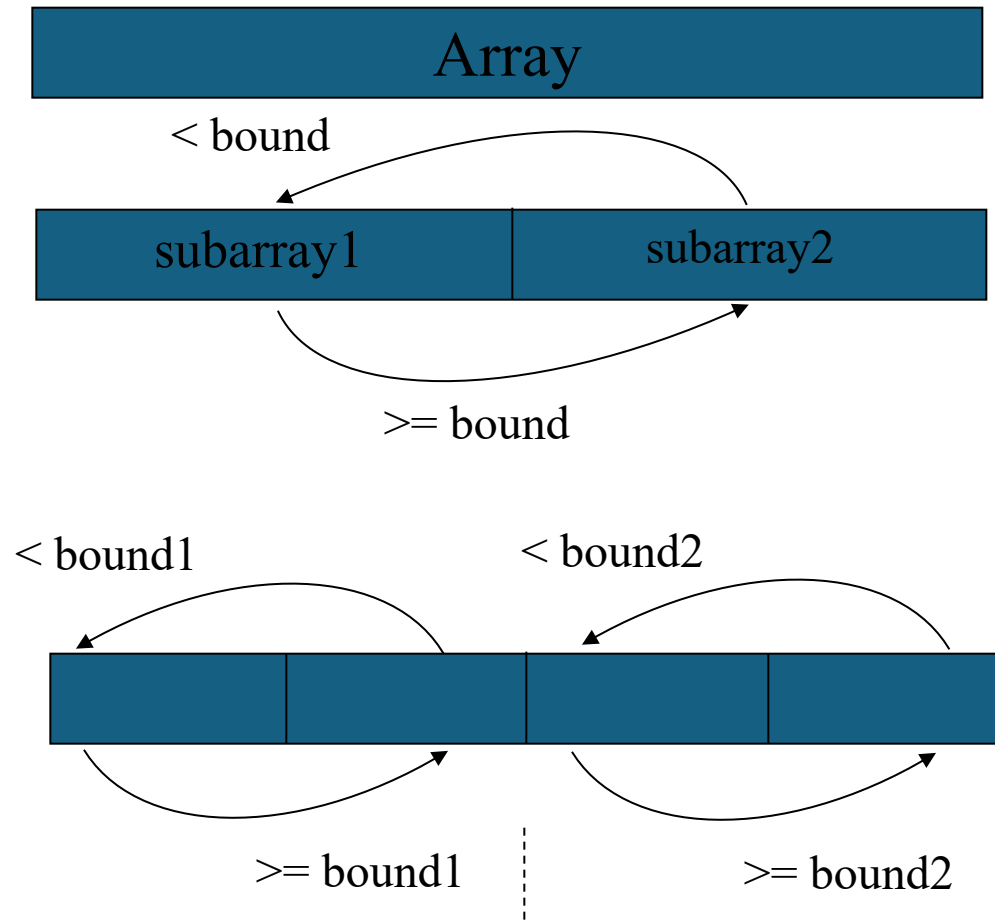
Muy estudiado, analizado y utilizado en la práctica.

Supongamos que conocemos el elemento mediano del arreglo

Algoritmo:

- Separar el arreglo en dos mitades: los elementos menores que el mediano por un lado y los mayores por el otro.
- Ordenar las dos mitades
- ¡Y listo!

Quick Sort/2



- ¿Y si no conocemos el elemento mediano?

Quick Sort (en algún lenguaje)

```
quicksort(a []) {  
    if (|a|>1) {  
        Elegir bound; /* subarray1 y subarray2 */  
        while (haya elementos en a)  
            if (elemento generico < bound)  
                insertar elemento en subarray1;  
            else insertar elemento en subarray2;  
        quicksort(subarray1);  
        quicksort(subarray2);  
    }  
}
```

Quicksort

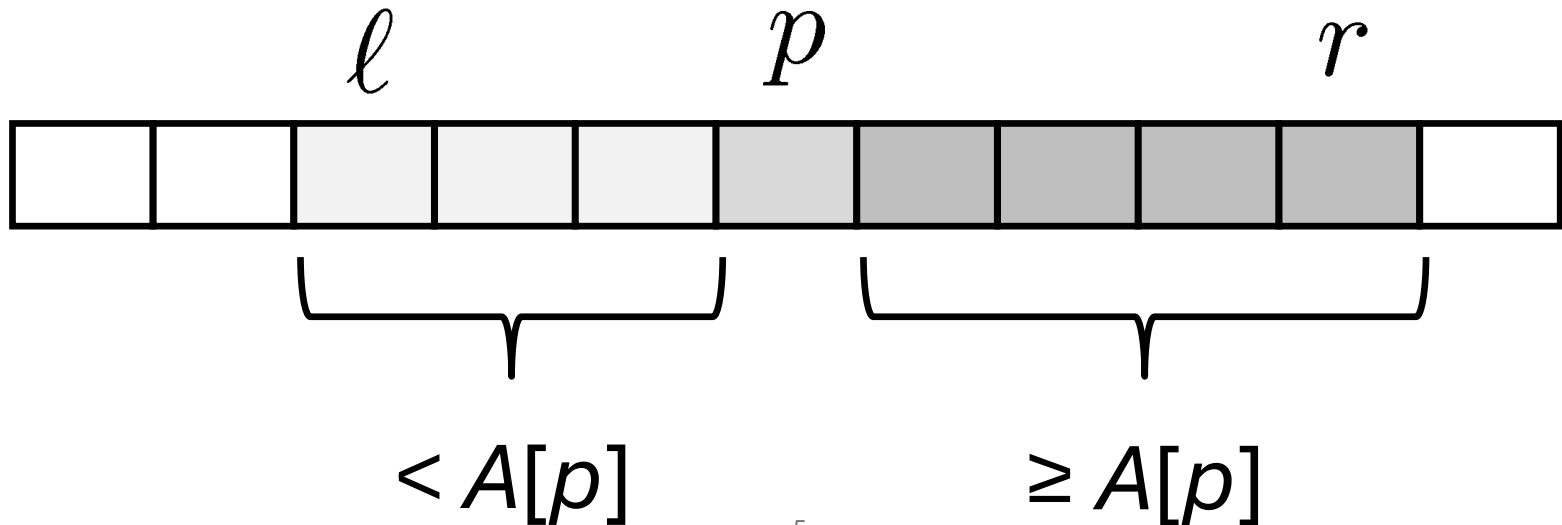
Function quicksort(A, ℓ, r)

if $\ell < r$ **then**

$p \leftarrow \text{partition}(A, \ell, r)$

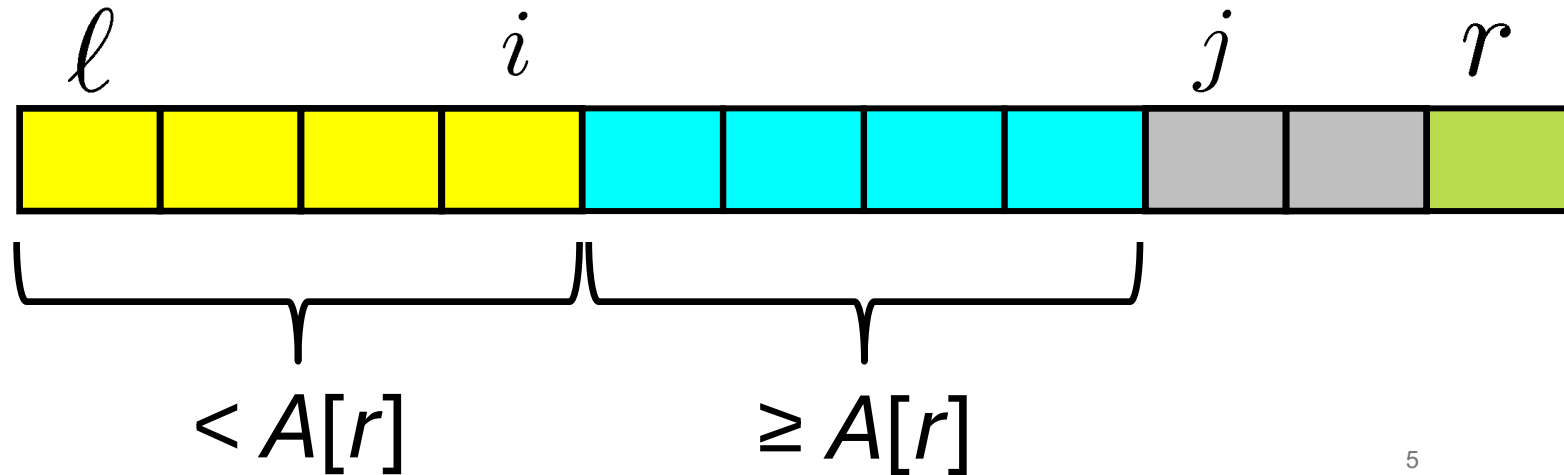
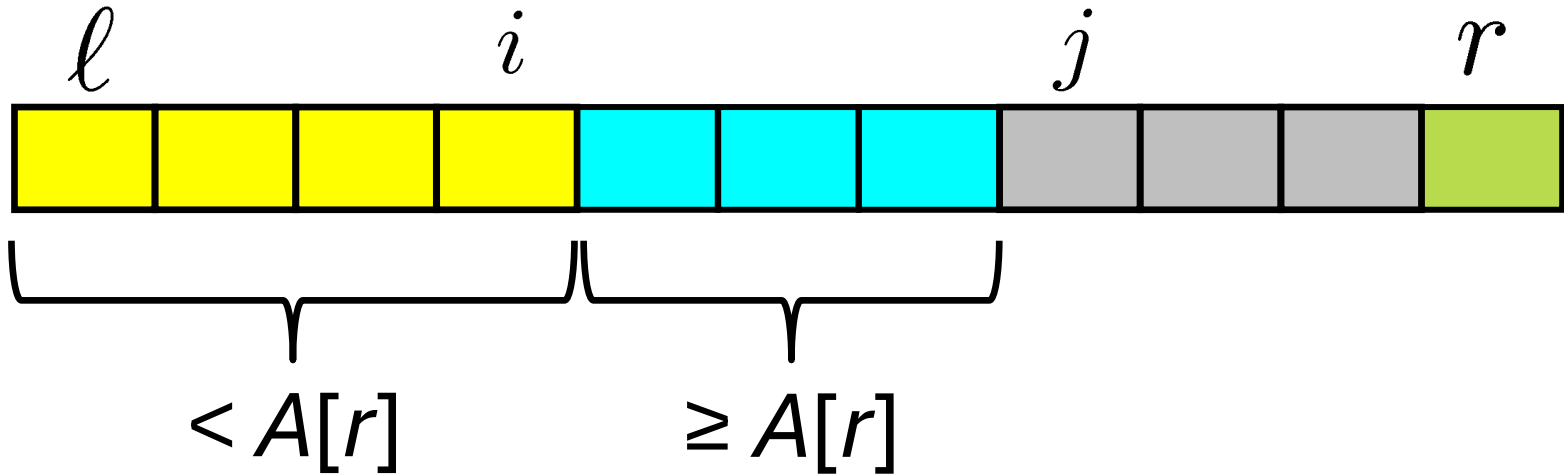
 quicksort($A, \ell, p - 1$)

 quicksort($A, p + 1, r$)



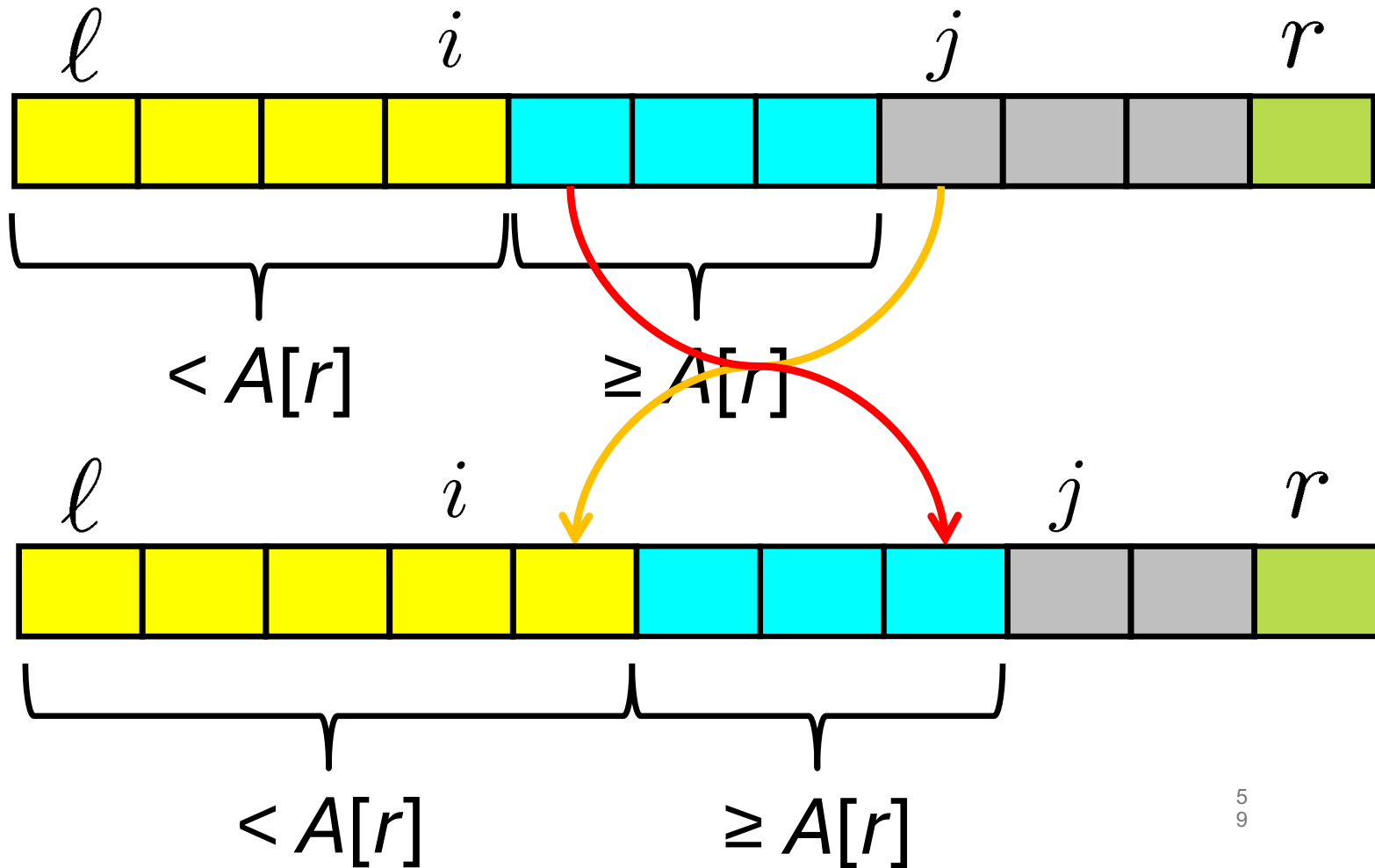
Partition

If $A[j] \geq A[r]$



Partition

If $A[j] < A[r]$



Function $\text{partition}(A, \ell, r)$

$i \leftarrow \ell - 1$

for $j \leftarrow \ell$ **to** $r - 1$ **do**

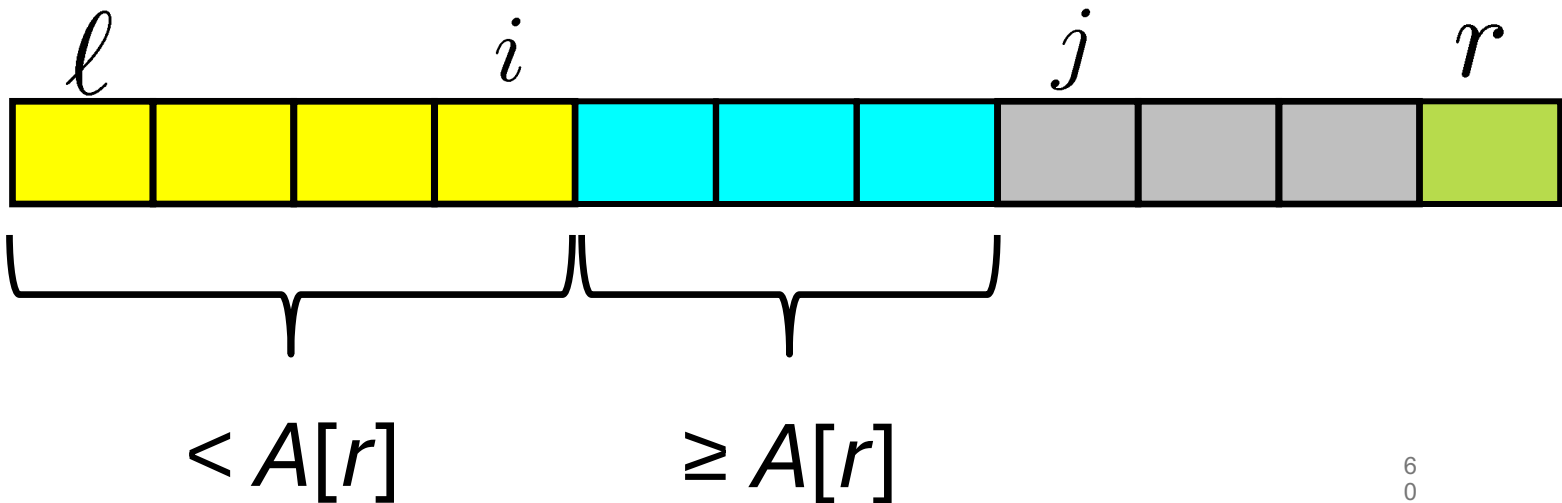
if $A[j] < A[r]$ **then**

$i \leftarrow i + 1$

$A[i] \leftrightarrow A[j]$

$A[i + 1] \leftrightarrow A[r]$

return $i + 1$

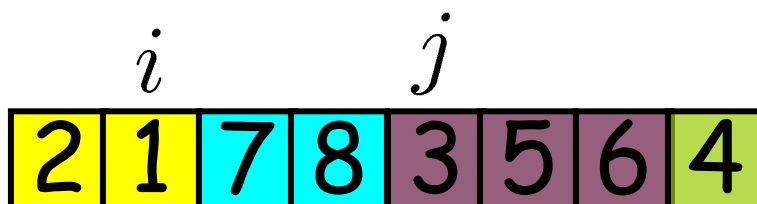
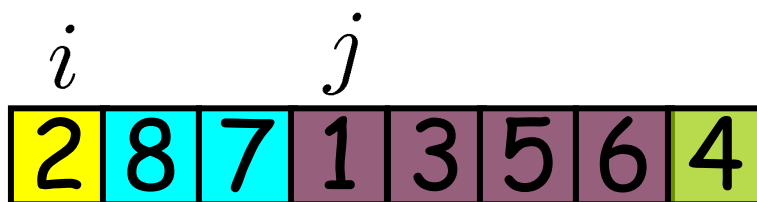
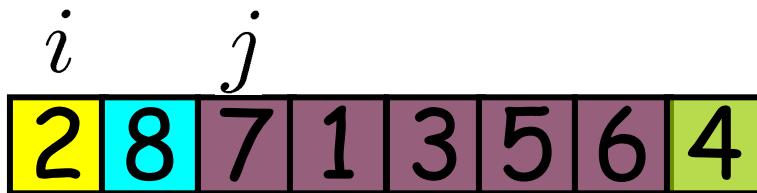
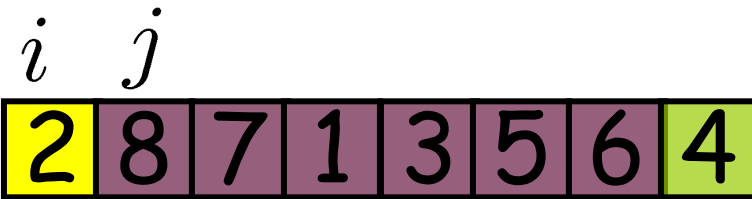


Partition

i ℓ j r

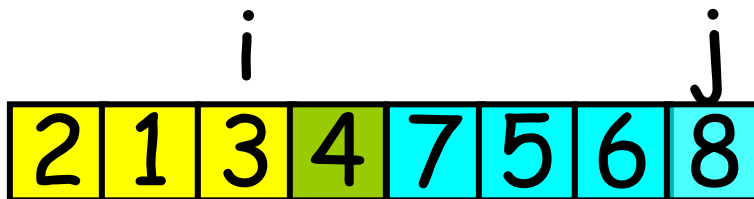
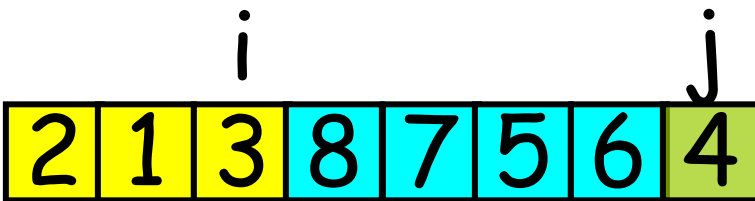
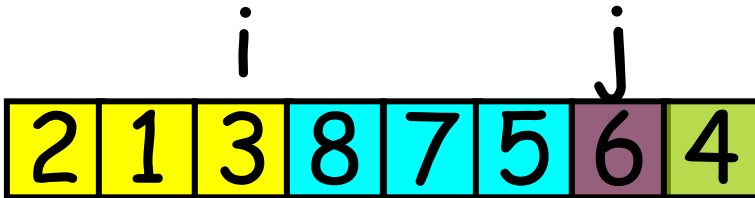
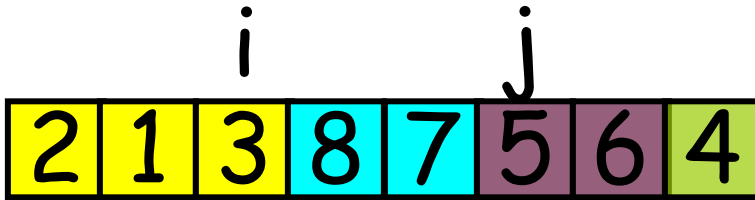
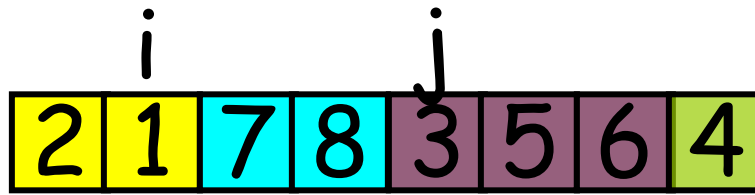


Use last key
as **pivot**



i : last key $< A[r]$

j : next key
to inspect



Move **pivot**
into position

Análisis de Quick Sort

Costo = $O(\text{No. comparaciones})$

Costo $O(n^2)$ en el caso peor

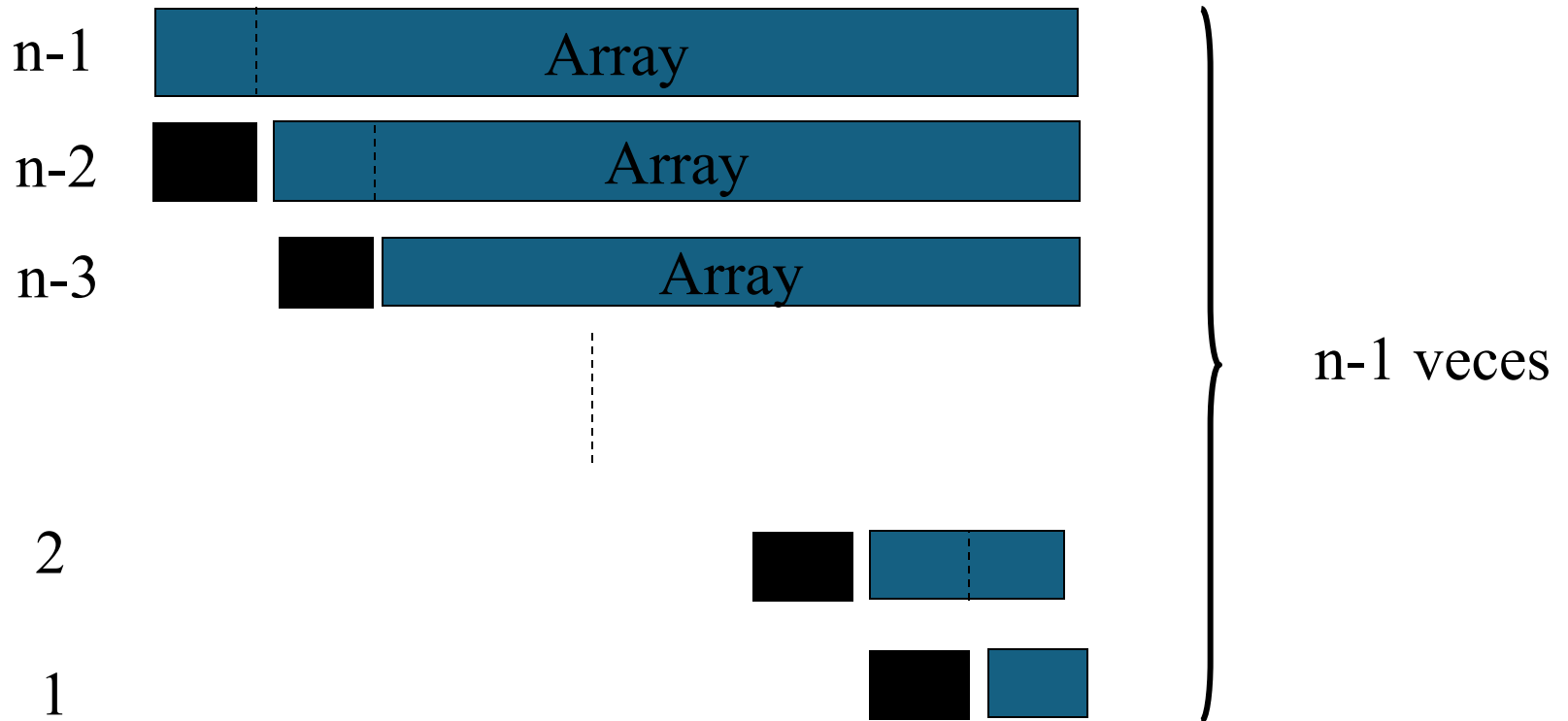
Costo $O(n \log n)$ en el caso mejor y promedio

En la práctica el algoritmo es eficiente

La elección del pivot es fundamental

Quick Sort – Caso peor

No. comparaciones
por sub-array



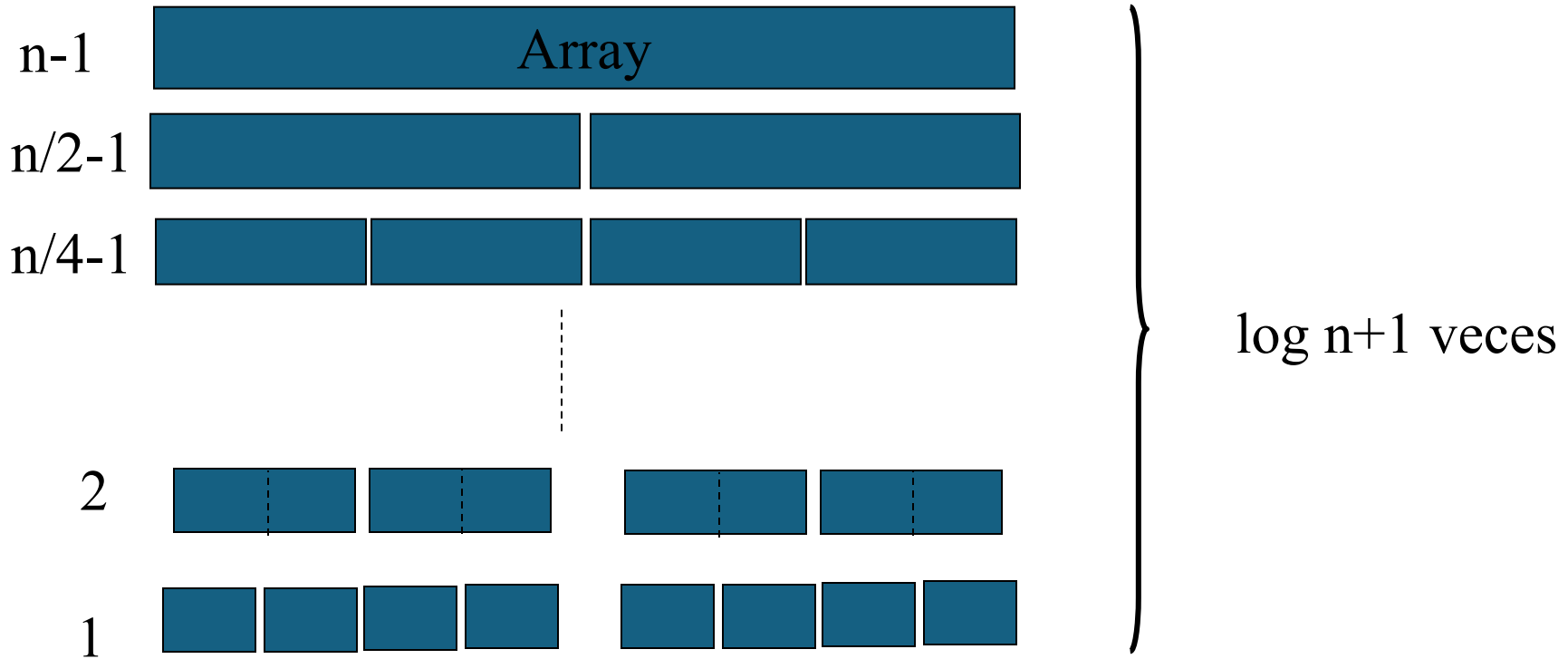
El elemento pivot es siempre el mínimo

$$\text{Costo} = O(n-1+n-2+\dots+2+1) = O(n^2)$$

Quick Sort – Caso mejor

No. comparaciones
por sub-array

n potencia de 2 por simplicidad



$$\text{Costo} = n + 2 \frac{n}{2} + 4 \frac{n}{4} + \cdots + n \frac{n}{n} = \sum_{i=0}^{\log n} 2^i \frac{n}{2^i} = n(\log n + 1)$$

Quick Sort – Caso Promedio

Idea: en un input al azar (proveniente de una distribución uniforme), la probabilidad de que el elemento i -ésimo sea el pivot.....es $1/n$

Tendríamos entonces la recurrencia

$$\begin{aligned} \bullet T(n) &= n+1 + \frac{1}{n} \sum_{1 \leq k \leq n} (T(k-1) + T(n-k)) = \\ &= n+1 + \frac{2}{n} \sum_{1 \leq k \leq n} T(k-1) = O(n \log n) \end{aligned}$$

Ojo, que no siempre podemos suponer que el input proviene de una distribución uniforme.

Pero.....¿y si lo “forzamos”?

Permutando el input o bien...

¡Elegiendo el pivote al azar! (Algoritmos probabilísticos)

Randomized quicksort

Function rand-quicksort(A, ℓ, r)

if $\ell < r$ **then**

$p \leftarrow \text{rand-partition}(A, \ell, r)$

 rand-quicksort($A, \ell, p - 1$)

 rand-quicksort($A, p + 1, r$)

Function rand-partition(A, ℓ, r)

$i \leftarrow \text{rand}(\ell, r)$

$A[i] \leftrightarrow A[r]$

return partition(A, ℓ, r)

(How do we generate random numbers?)

Complejidad de los algoritmos de ordenamiento

Merge Sort: $O(n \log n)$

Quick Sort, Selection Sort, Insertion Sort: $O(n^2)$

Quick Sort: $O(n \log n)$ en el caso mejor

Selection Sort: $O(n^2)$ en todos los casos

Insertion Sort: $O(n)$ en el caso mejor

Pregunta: ¿cuál es la eficiencia máxima (complejidad mínima) obtenible en el caso peor? -> Lower bound

Ordenamiento – límites inferiores

Observación fundamental: todos los algoritmos deben comparar elementos (o sea, ese es nuestro modelo de cómputo)

- Dados a_i , a_k , tres casos posibles: $a_i < a_k$, $a_i > a_k$, o $a_i = a_k$
- Se asume por simplicidad que todos los elementos son distintos
- Se asume entonces que todas las comparaciones tienen la forma $a_i < a_k$, y el resultado de la comparación es verdadero o falso

Nota: si los elementos pueden tener valores iguales entonces se consideran solamente comparaciones del tipo $a_i \leq a_k$

Árboles de decisión

Búsqueda binaria de elemento x en un arreglo:

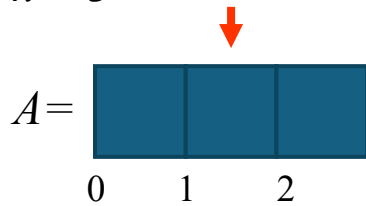
$$n = 3$$



Árboles de decisión

Búsqueda binaria de elemento x en un arreglo:

$n = 3$

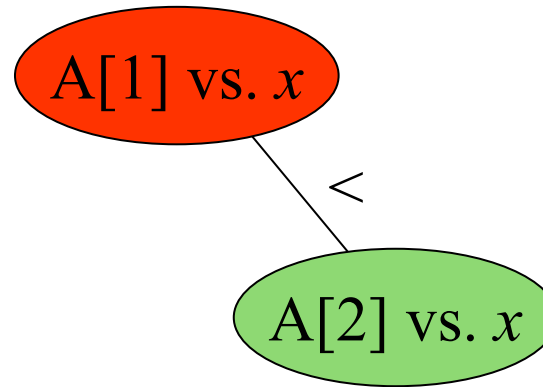
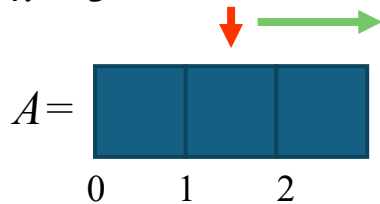


Los nodos internos
representan
decisiones binarias

Árboles de decisión

Búsqueda binaria de elemento x en un arreglo:

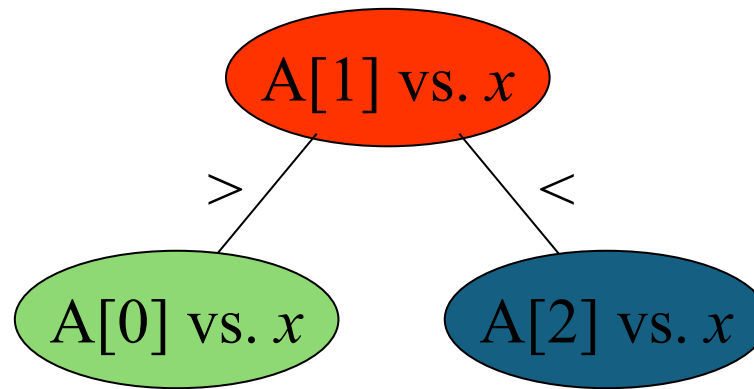
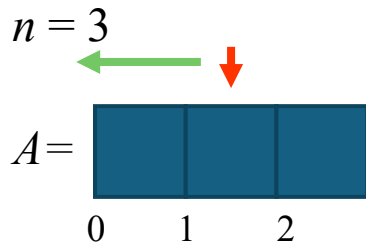
$n = 3$



Los nodos internos
representan
decisiones binarias

Árboles de decisión

Búsqueda binaria de elemento x en un arreglo:

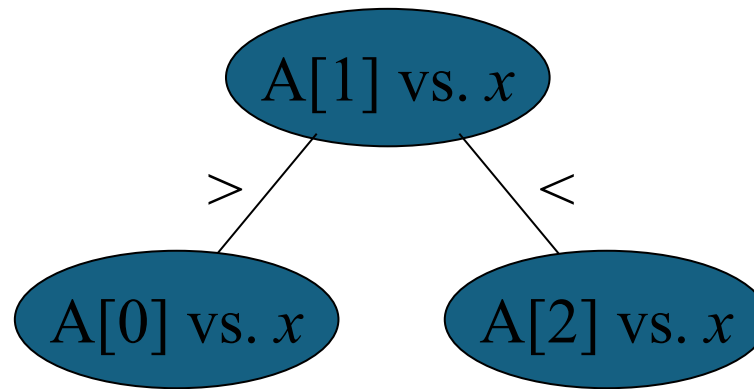
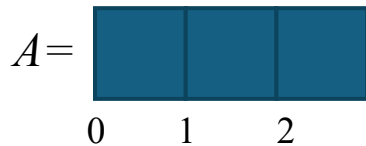


Los nodos internos
representan
decisiones binarias

Árboles de decisión

Búsqueda binaria de elemento x en un arreglo:

$n = 3$

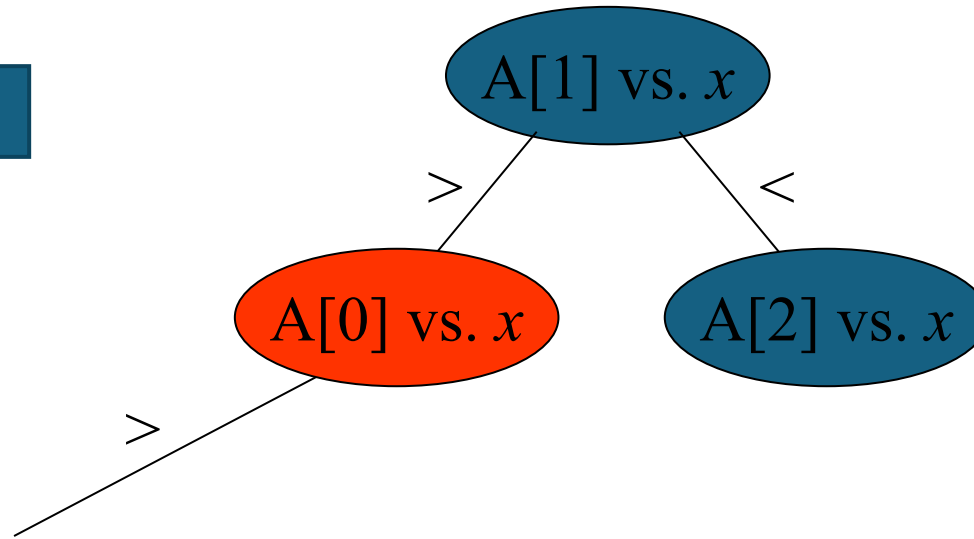
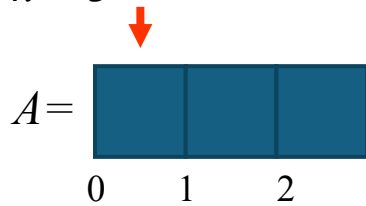


Los nodos internos
representan
decisiones binarias

Árboles de decisión

Búsqueda binaria de elemento x en un arreglo:

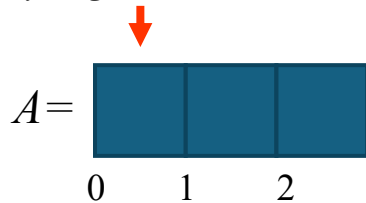
$n = 3$



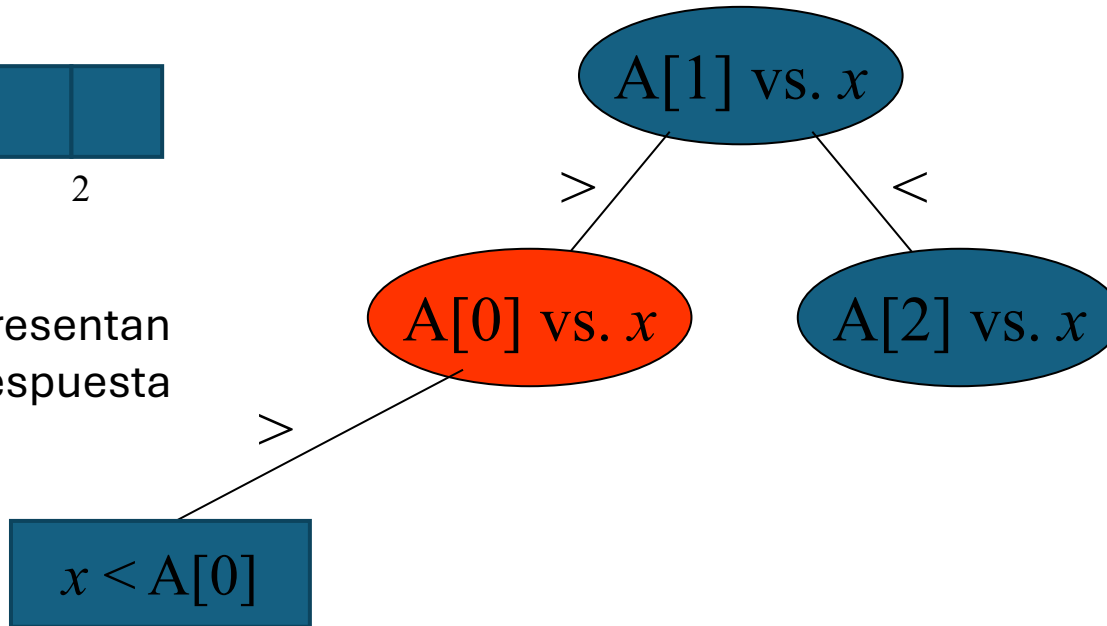
Árboles de decisión

Búsqueda binaria de elemento x en un arreglo:

$n = 3$



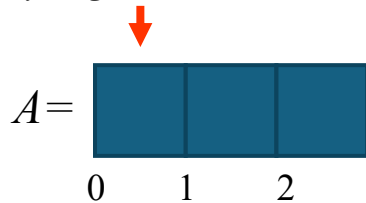
Las hojas representan una posible respuesta



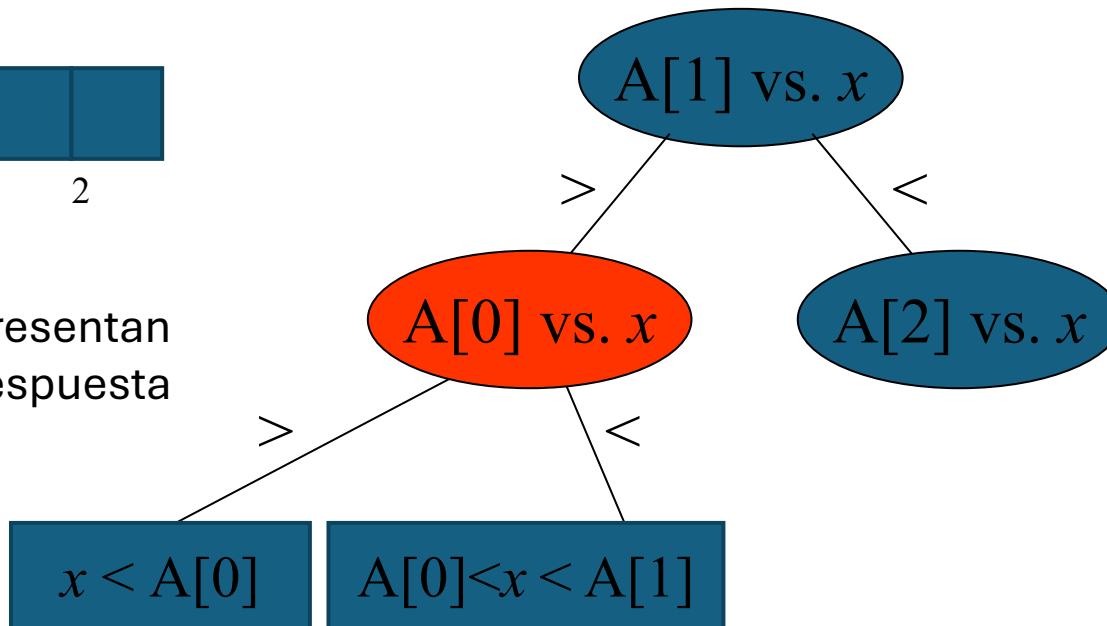
Árboles de decisión

Búsqueda binaria de elemento x en un arreglo:

$n = 3$



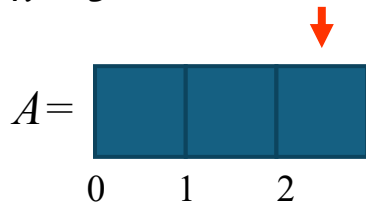
Las hojas representan una posible respuesta



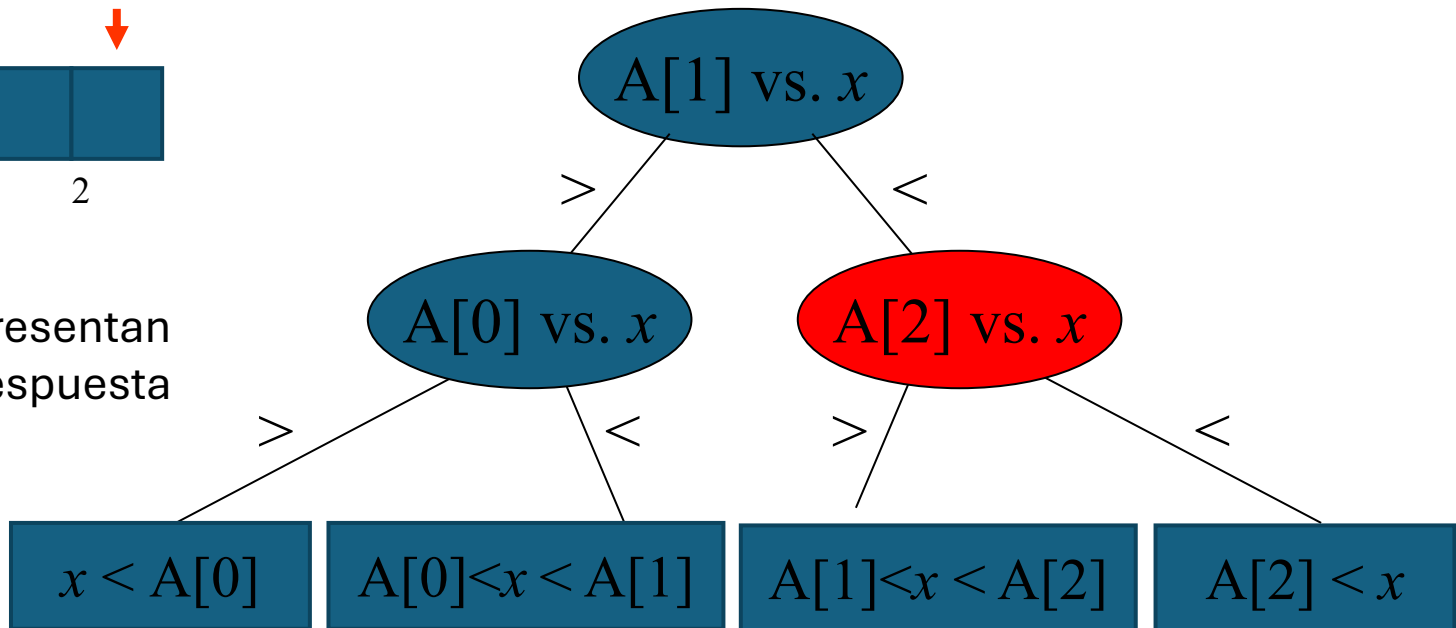
Árboles de decisión

Búsqueda binaria de elemento x en un arreglo:

$n = 3$



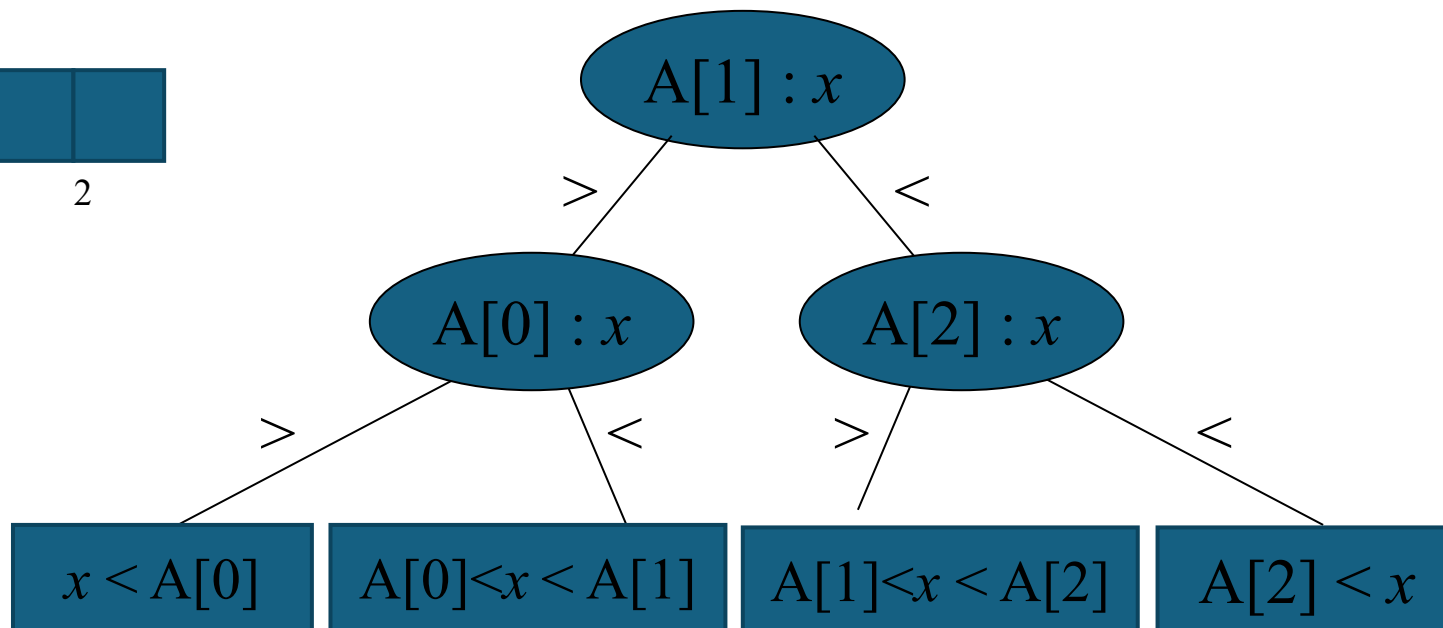
Las hojas representan una posible respuesta



Árboles de decisión

Búsqueda binaria de elemento x en un arreglo:

$n = 3$

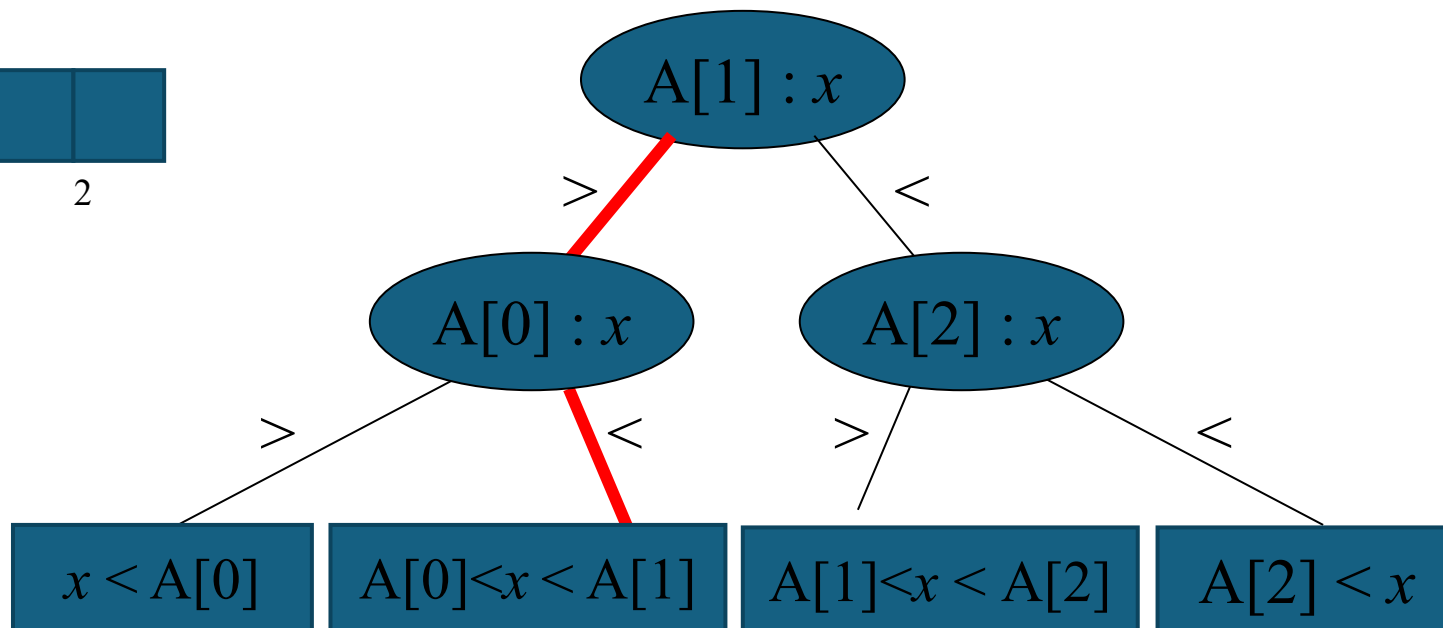
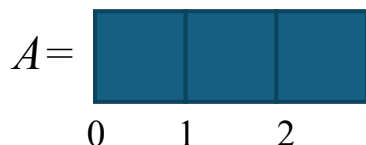


- Un árbol de decisión representa las comparaciones ejecutadas por un algoritmo sobre un input dado
- Cada hoja corresponde a una de las posibles outputs del algoritmo.

Árboles de decisión

Búsqueda binaria de elemento x en un arreglo:

$n = 3$

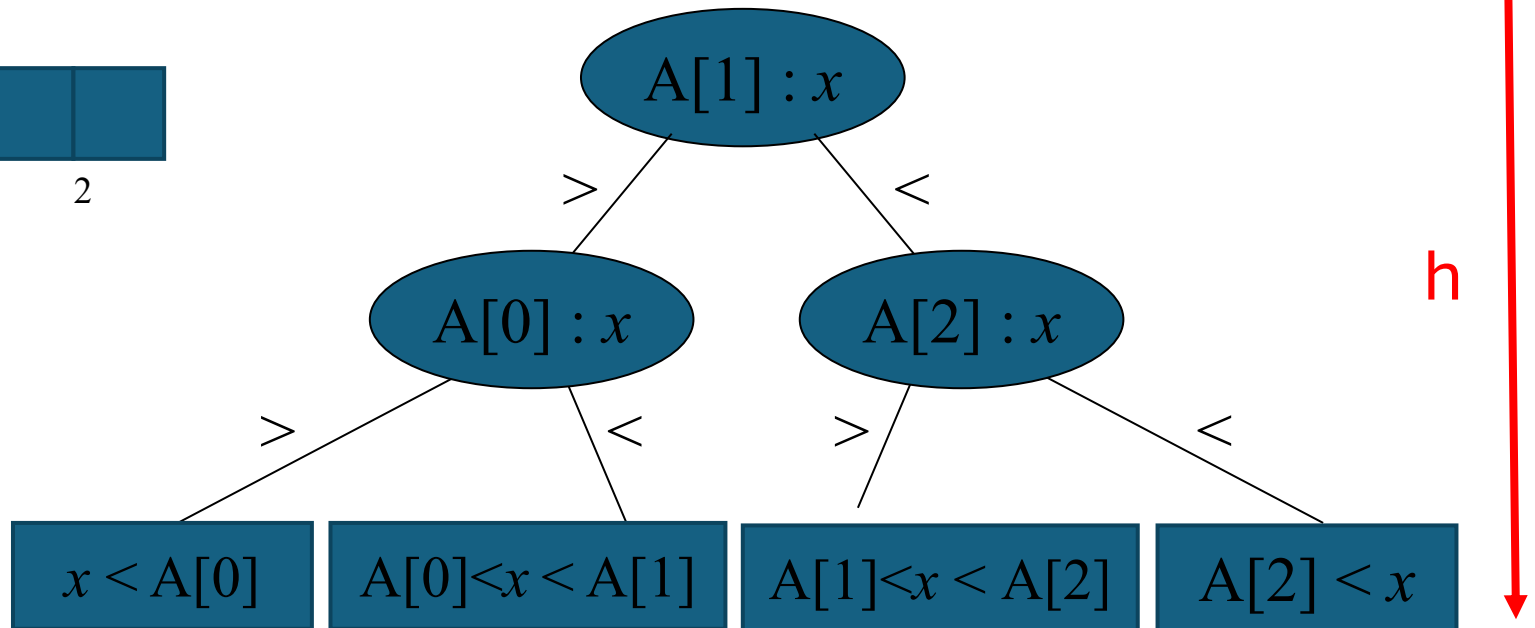


- Un árbol de decisión representa las comparaciones ejecutadas por un algoritmo sobre un input dado
- Cada hoja corresponde a una de las posibles outputs del algoritmo.

Árboles de decisión

Búsqueda binaria de elemento x en un arreglo:

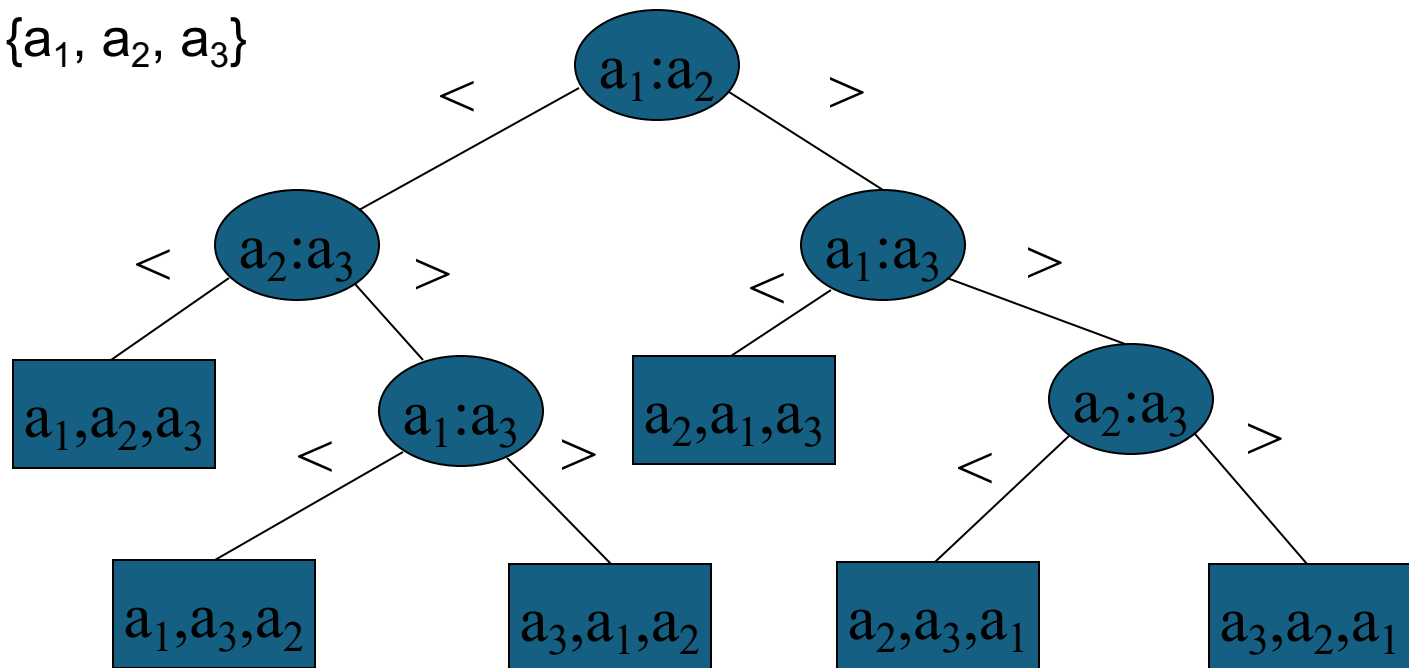
$n = 3$



- Un árbol de decisión representa las comparaciones ejecutadas por un algoritmo sobre un input dado
- Cada hoja corresponde a una de las posibles outputs del algoritmo.

Árboles de decisión

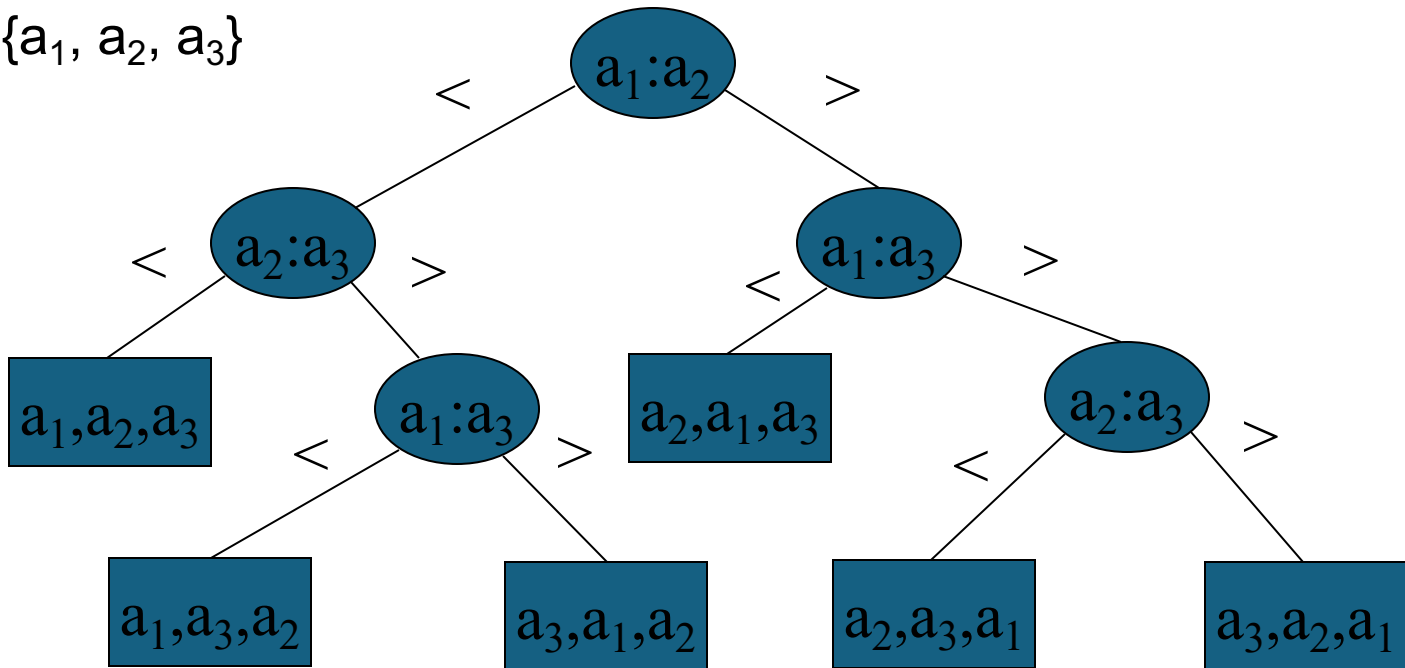
Para ordenar un
conjunto $\{a_1, a_2, a_3\}$



- Un árbol de decisión representa las comparaciones ejecutadas por un algoritmo sobre un input dado
- Cada hoja corresponde a una de las posibles permutaciones

Árboles de decisión/2

Para ordenar un conjunto $\{a_1, a_2, a_3\}$



- Hay $n!$ posibles permutaciones \rightarrow el árbol debe contener $n!$ hojas
- La ejecución de un algoritmo corresponde a un camino en el árbol de decisión correspondiente al input considerado

Árboles de decisión/3

- El camino más largo de la raíz a una hoja (altura) representa el número de comparaciones que el algoritmo tiene que realizar en el caso peor
- Teorema: cualquier árbol de decisión que ordena n elementos tiene altura $\Omega(n \log n)$
- Demostración:
 - Árbol de decisión es binario
 - Con $n!$ Hojas
 - Altura mínima $\rightarrow \Omega(\log(n!)) = \Omega(n \log n)$

Árboles de decisión/4

Corolario: ningún algoritmo de ordenamiento tiene complejidad mejor que $\Omega(n \log n)$

Corolario: el algoritmo Merge Sort tienen complejidad asintótica óptima

Nota: existen algoritmos de ordenamiento con complejidad más baja, pero requieren ciertas hipótesis extra sobre el input

Árboles de decisión/4

- Coro
com
- Coro
com
- Nota
com
extra

Integer Sorting in $O(n\sqrt{\log \log n})$ Expected Time and Linear Space

Yijie Han*

School of Interdisciplinary Computing and Engineering
University of Missouri at Kansas City
5100 Rockhill Road
Kansas City, MO 64110
hanyij@umkc.edu
<http://welcome.to/yijiehan>

Mikkel Thorup

AT&T Labs— Research
Shannon Laboratory
180 Park Avenue
Florham Park, NJ 07932
mthorup@research.att.com

Abstract

We present a randomized algorithm sorting n integers in $O(n\sqrt{\log \log n})$ expected time and linear space. This improves the previous $O(n \log \log n)$ bound by Anderson et al. from STOC'95.

of the input integers. The assumption that each integer fits in a machine word implies that integers can be operated on with single instructions. A similar assumption is made for comparison based sorting in that an $O(n \log n)$ time bound requires constant time comparisons. However, for integer sorting, besides comparisons, we can use all the other in-

Demos de Ordenamiento

- Sorting Algorithms Animations:
<https://www.toptal.com/developers/sorting-algorithms>
- Algoritmos de ordenamiento con baile húngaro, rumano y gitano:
<https://www.fayerwayer.com/2011/04/algoritmos-de-ordenamiento-con-baile-hungaro-rumano-y-gitano/>