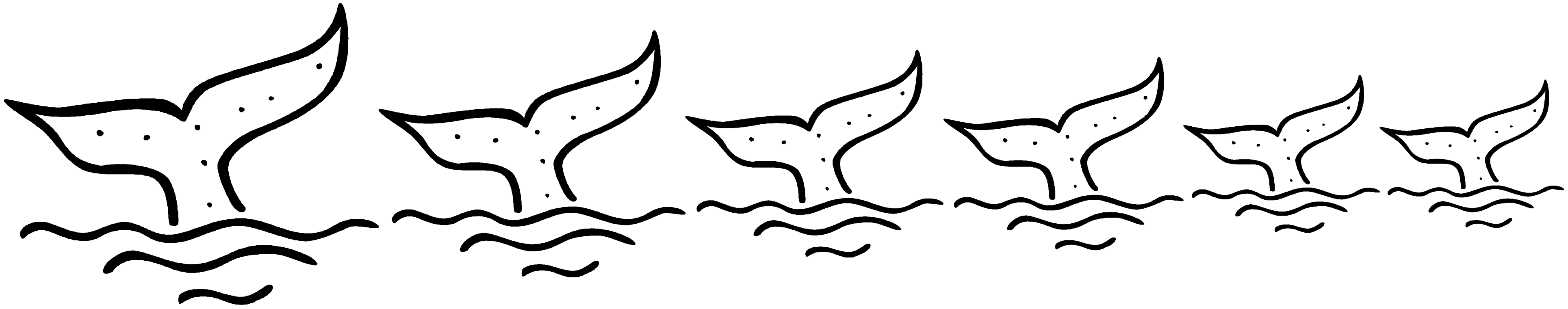
# Ordenamiento (Sorting)



# El problema del ordenamiento

- Ordenar: arreglo[$\alpha$] $\rightarrow$ arreglo[$\alpha$], donde $\alpha$ es un tipo tal que está definida la relación $<_{\alpha}$
- Uno de los problemas más clásicos, útiles y estudiados de la informática
- Variante: ordenamiento en memoria secundaria (por ejemplo grandes archivos)

# Selection Sort

- Algoritmo:
  - Repetir para las posiciones sucesivas i del arreglo:
    - Seleccionar el mínimo elemento que se encuentra entre la posición actual y el final.
    - Ubicarlo en la posición i, intercambiándolo con el ocupante original de esa posición.
- Invariante:
  - los elementos entre la posición 0 y la posición i son los i+1 elementos más pequeños del arreglo original,
  - los elementos entre la posición 0 y la posición i se encuentran ordenados,
  - El arreglo es una permutación del arreglo original (o sea los elementos entre las posiciones i+1 y n-1 son los n-i-1 elementos más grandes del arreglo original).

# Selection Sort

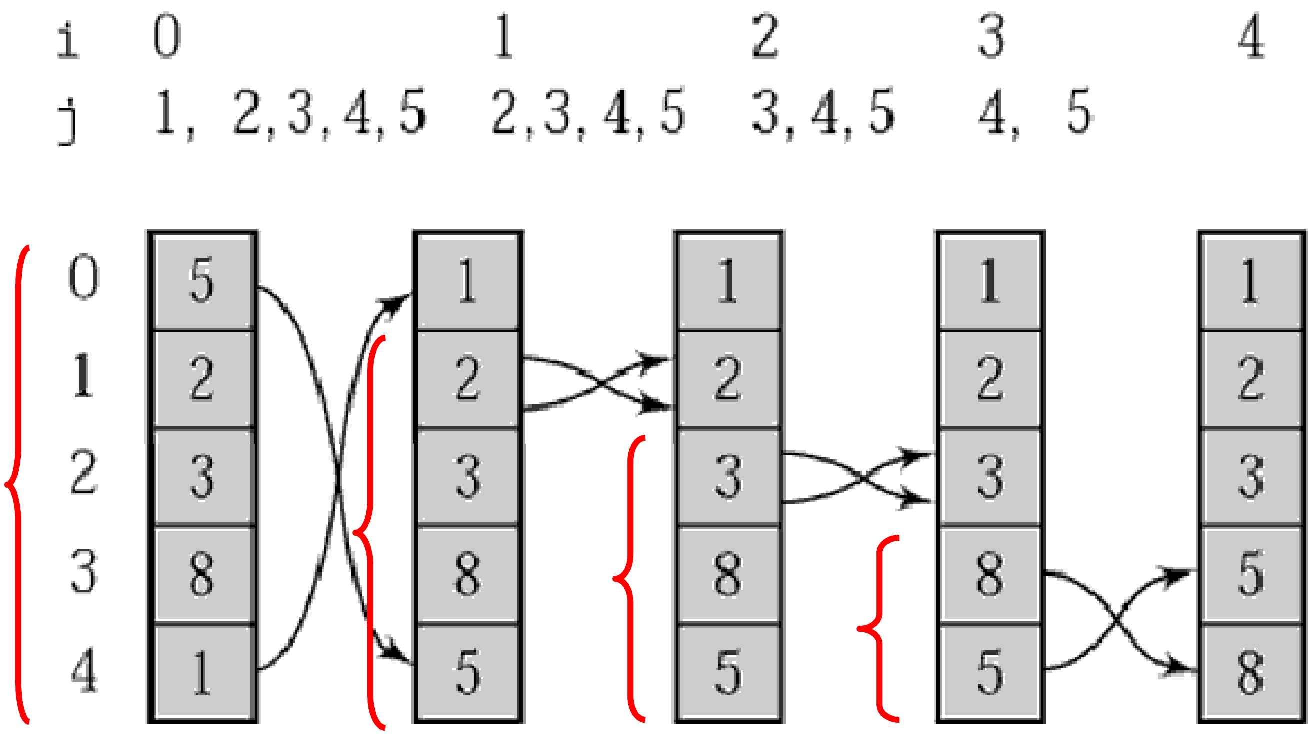
- Para i desde 0 hasta n-2 hacer
  - min ← seleccionar_min(i,n-1)
  - Intercambiar(i,min)

- Para i desde 0 hasta n-2 hacer
  - min ← i
  - Para j desde i+1 hasta n-1 hacer
    - si a[j]<a[min] entonces min ←j
  - Intercambiar(i,min)

# 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

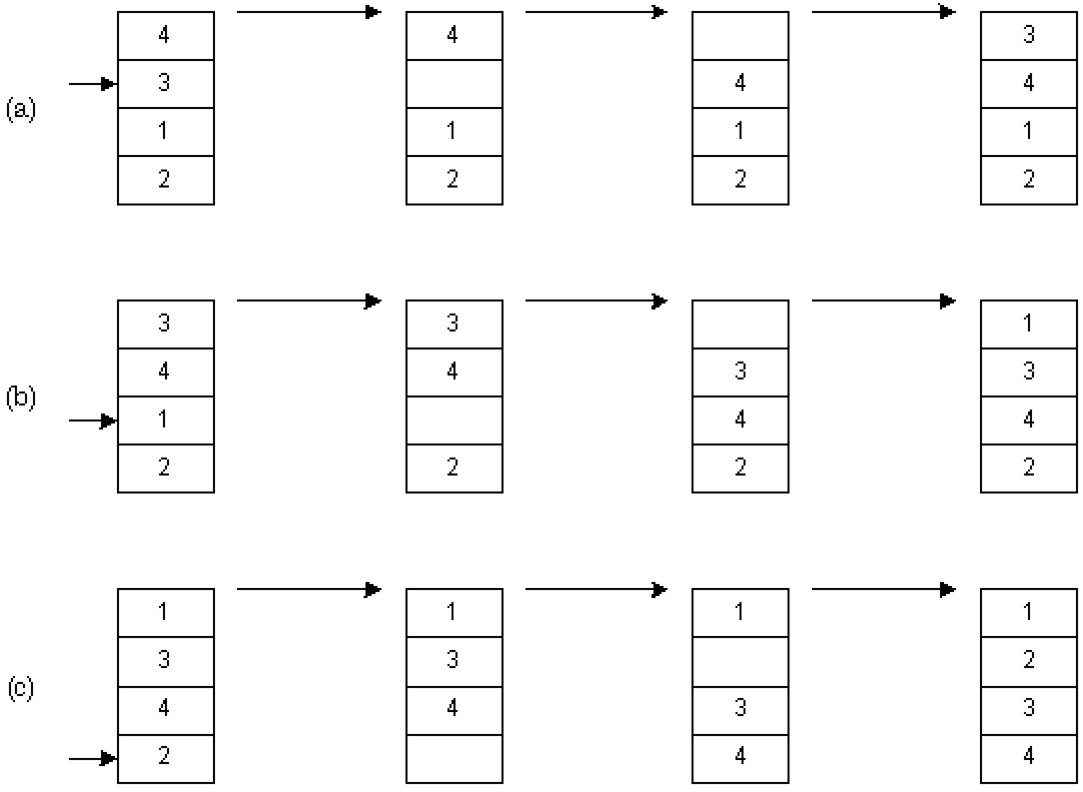
- Invariante:
  - los elementos entre la posición 0 y la posición i son los elementos que ocupaban las posiciones 0 a i del arreglo original,
  - los elementos entre la posición 0 y la posición i se encuentran ordenados,
  - El arreglo es una permutación del arreglo original (o sea que los elementos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

- Para i desde 1 hasta n-1 hacer
  - Insertar(i)

- Para i desde 1 hasta n-1 hacer
  - j $\leftarrow$ i-1, elem $\leftarrow$ a[i]
  - mientras j$\geq$0 $\wedge_L$ a[j] > elem hacer
    - a[j+1] $\leftarrow$ a[j]
    - j $\leftarrow$ j-1
  - a[j+1] $\leftarrow$ elem

# 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

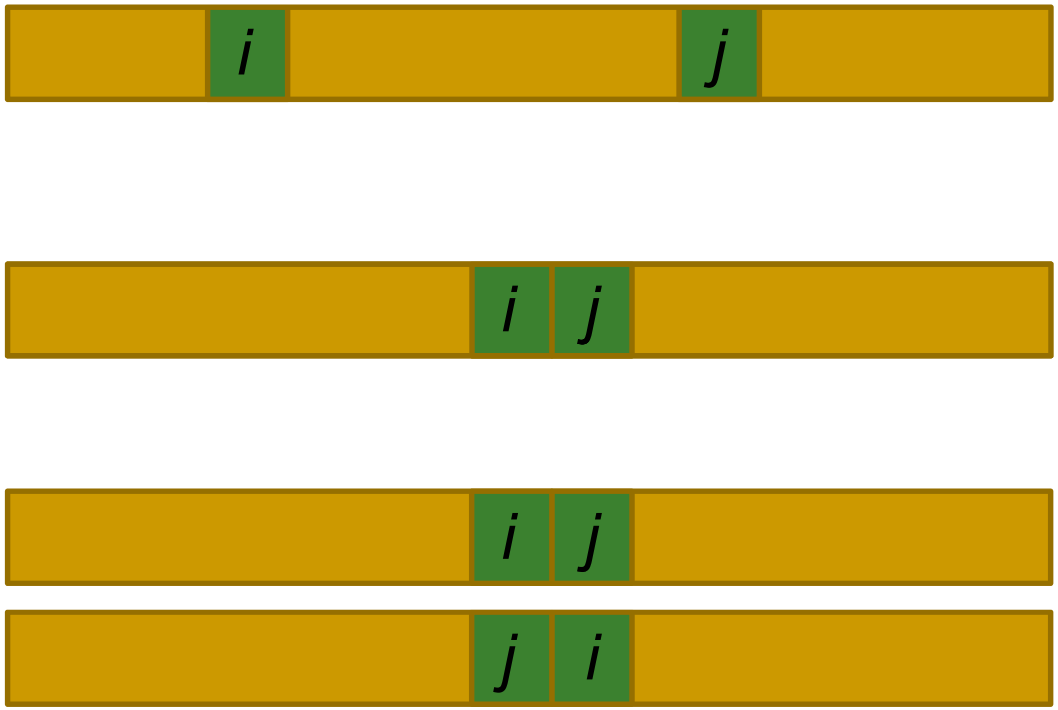
# Estabilidad

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) | (Bruno, TM) | (Valentina, TM) |
| (Federico, TM) | (Federico, TM) | (Bruno, TM) |
| (Florencia, TT) | (Valentina, TM) | (Federico, TM) |
| (Leandro, TT) | (Florencia, TT) | (Martina, TT) |
| (Martina, TT) | (Leandro, TT) | (Leandro, TT) |
| (Valentina, TM) | (Martina, TT) | (Florencia, TT) |
| | Algoritmo estable | Algoritmo inestable |

# Estabilidad - Selection Sort

(6, 1)

(4, 3)

(6, 5)

(1, 7)

# Estabilidad - Selection Sort

| | |
|---|---|
| (6, 1) | (6, 1) |
| (4, 3) | (4, 3) |
| (6, 5) | (6, 5) |
| (1, 7) | (1, 7) |

# Estabilidad - Selection Sort

| | | |
|---|---|---|
| (6, 1) | (6, 1) | (1, 7) |
| (4, 3) | (4, 3) | (4, 3) |
| (6, 5) | (6, 5) | (6, 5) |
| (1, 7) | (1, 7) | (6, 1) |

Algoritmo inestable pero…

# HeapSort

- Valor de las estructura de datos: Selection Sort usa n búsquedas de mínimo. ¿Cómo se hacía eso eficientemente?
- Podemos meter los elementos del arreglo uno a uno en un heap, y luego ir sacándolos.
- Pero: se puede hacer algo todavía mejor.
- ¿Se acuerdan de la operación Array2Heap y del algoritmo de Floyd? Complejidad: O(n)
- Algoritmo de ordenamiento de un array basado en un heap
- Algoritmo
  - heap ← array2heap (A)
  - para i desde n-1 hasta 0 hacer
    - max ← próximo (heap)
    - desencolar
    - A[i] ←max
- Costo: O(n) + O(n log n)
- Notar que no requiere memoria adicional

# Merge Sort

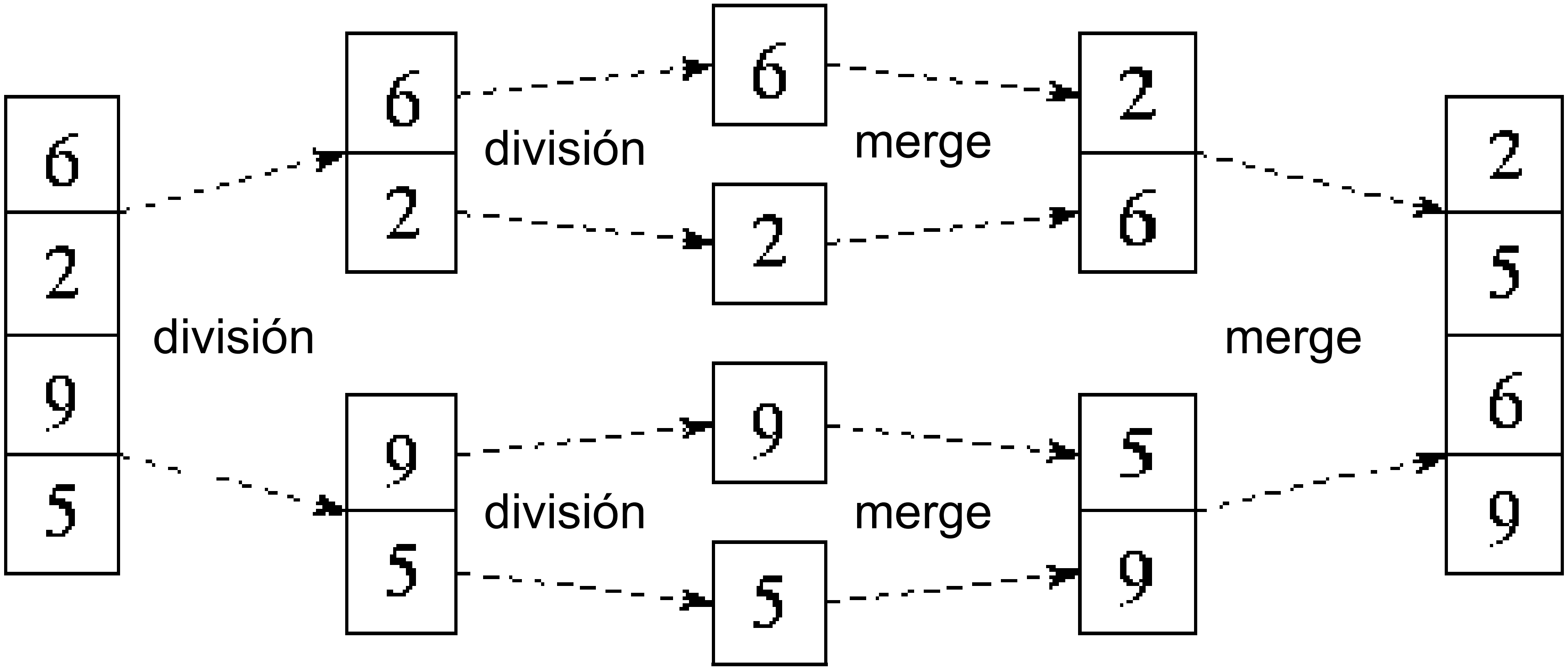
- Clásico ejemplo de la metodología “Divide & Conquer” (o “Divide y Reinarás”)
- La metodología consiste en
  - dividir un problema en problemas similares....pero más chicos
  - resolver los problemas menores
  - Combinar las soluciones de los problemas menores para obtener la solución del problema original.
- El método tiene sentido siempre y cuando la división y la combinación no sean excesivamente caras
- ¿Entonces?
- 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.
  - “Fundir” ambas mitades (ya ordenadas) en un único arreglo

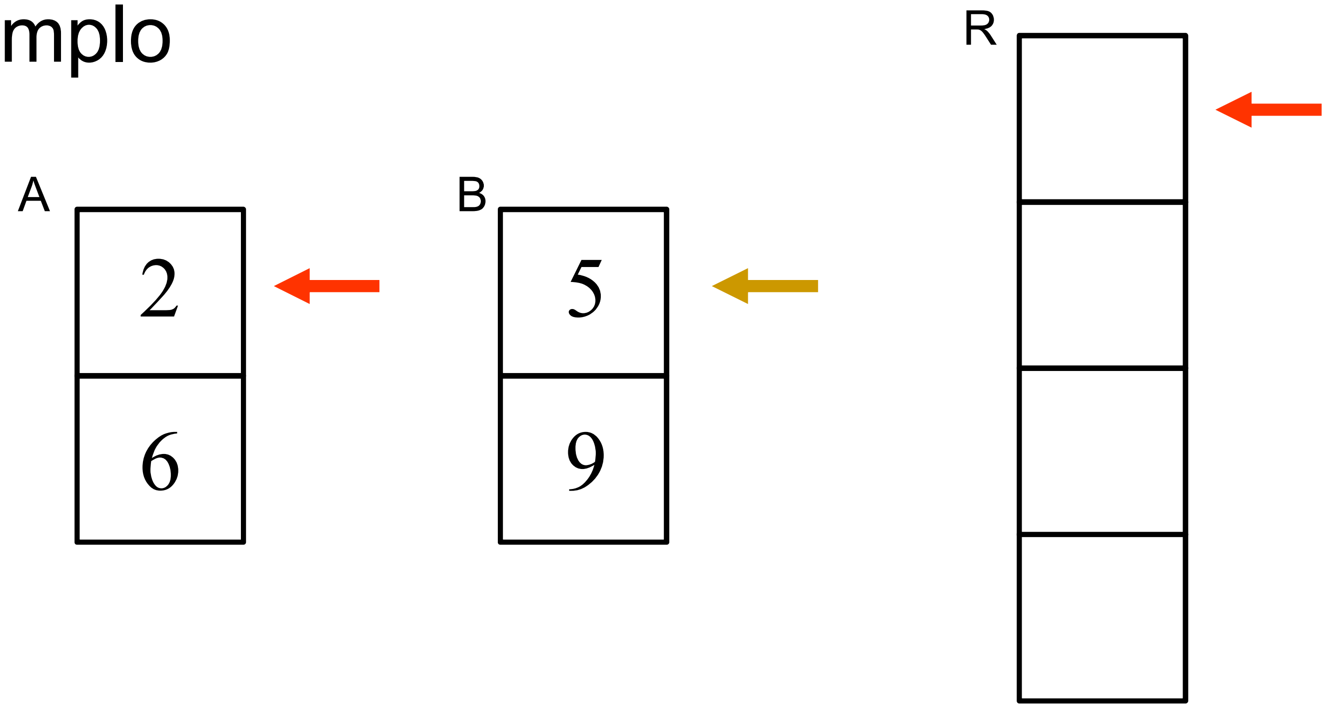
# Merge Sort

- Ejemplo



# Merge Sort

- Ejemplo

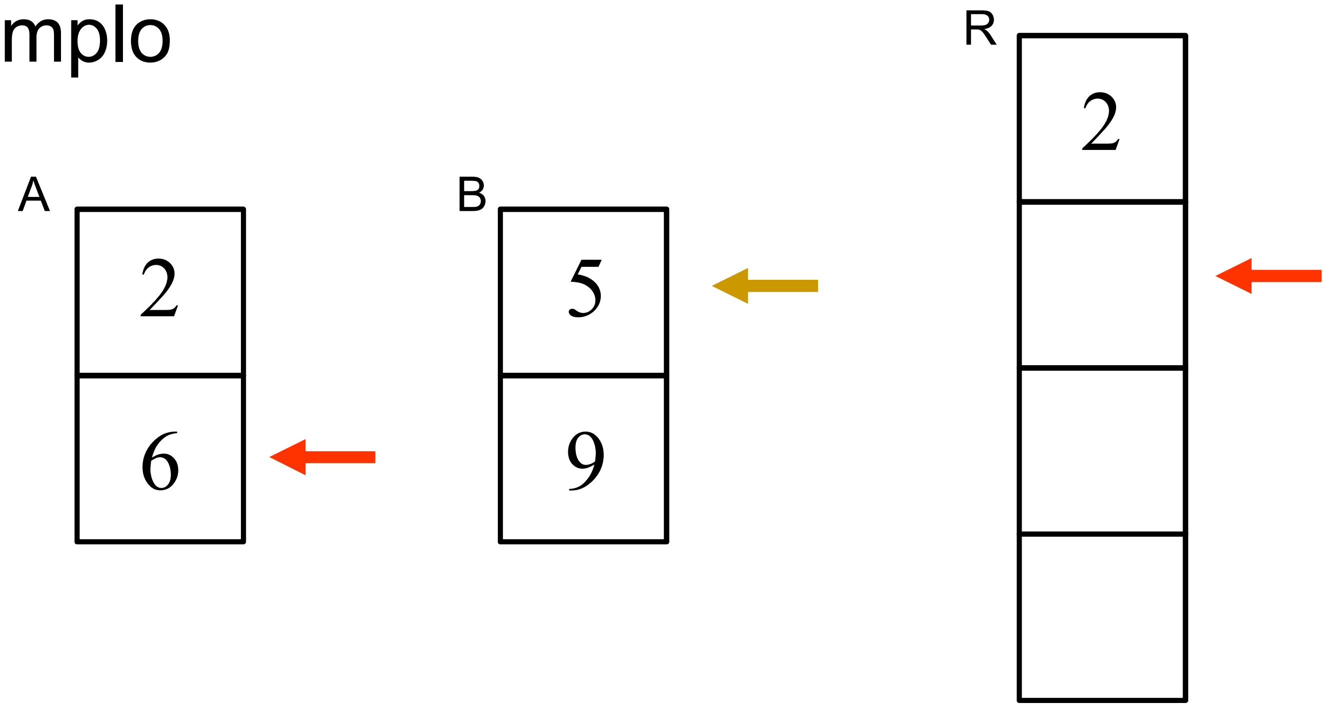



n=4

#pasos=0

# Merge Sort

- Ejemplo

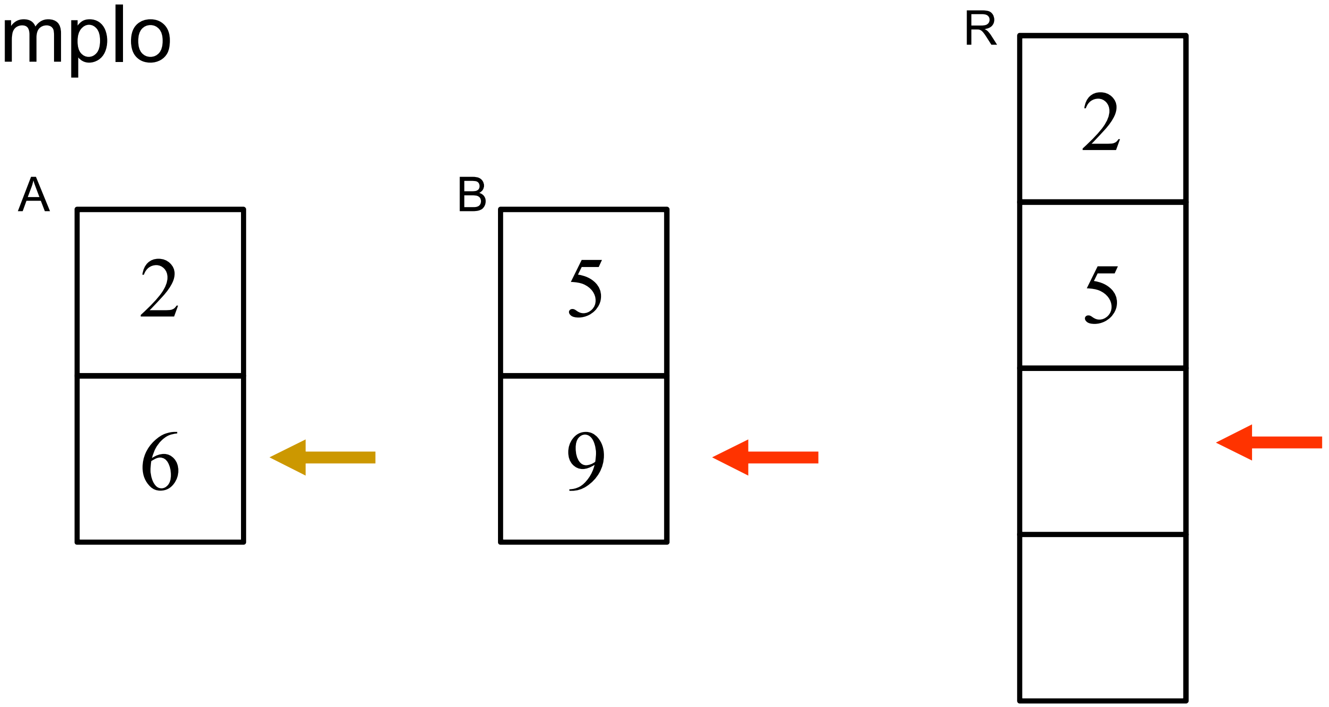



n=4

#pasos=1

# Merge Sort

- Ejemplo

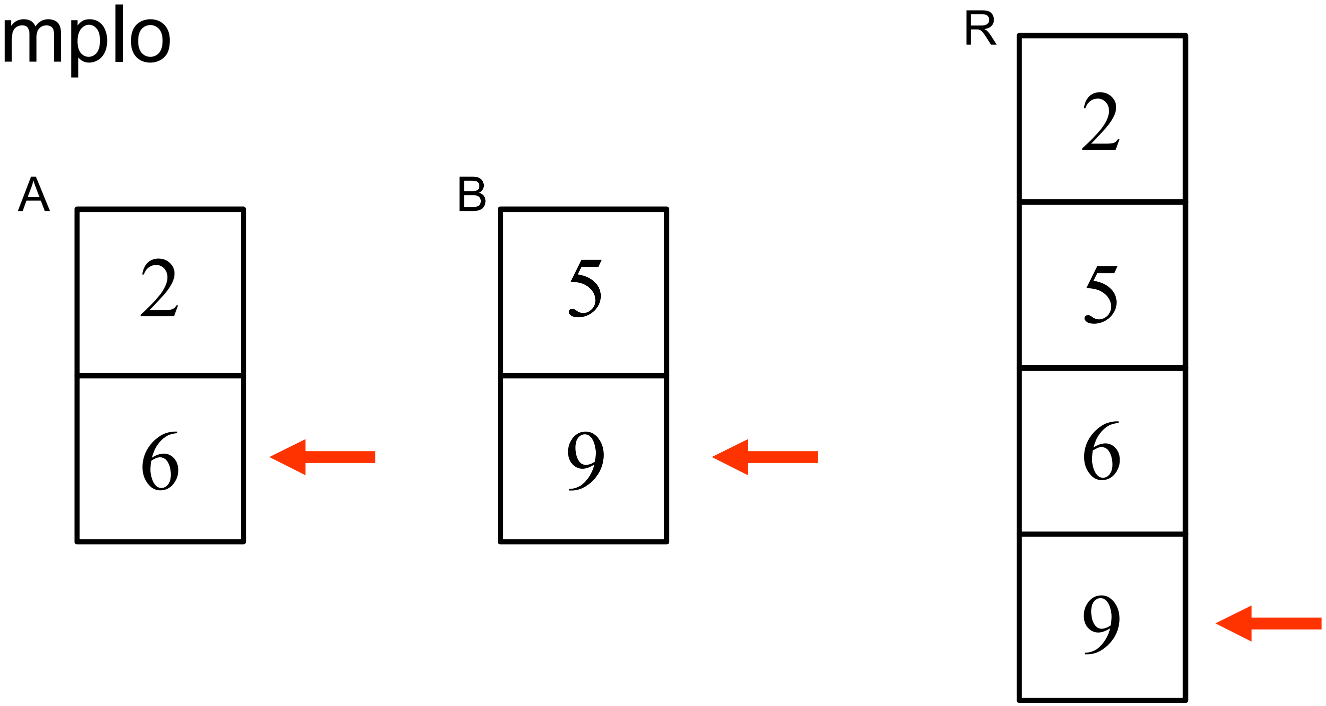



n=4

#pasos=2

# Merge Sort

- Ejemplo




n=4

#pasos=3

# Merge Sort

- Análisis

**MERGE-SORT** $A[0..n-1]$

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

# Merge Sort

- Análisis

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

$O(1)$ 1. Si n=1, ya está ordenado.

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

$O(n\text{-}1)$ 3. *Merge* las dos mitades ordenadas

# Merge Sort

- Análisis

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

$O(1)$ 1. Si $n=1$, ya está ordenado.

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

$O(n\text{-}1)$ 3. *Merge* las dos mitades ordenadas

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

# Merge Sort

- Análisis

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

# Merge Sort

- Análisis

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

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

# Merge Sort

- Análisis

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

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

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

# Merge Sort

- Análisis

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

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

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

# Merge Sort

- Análisis

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

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

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

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

# Merge Sort

- Análisis

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

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

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

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

# Merge Sort

- Análisis

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

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

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

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

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

# Merge Sort

- Análisis

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

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

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

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

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

$$= \cdots =$$

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

$$= \cdots =$$

# Merge Sort

- Análisis

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

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

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

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

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

$$= \cdots =$$

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

$$= \cdots =$$

# Merge Sort

- Análisis

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

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

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

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

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

$$= \cdots =$$

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

$$= \cdots =$$

# Merge Sort

- Análisis

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

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

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

# Merge Sort

- Análisis

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

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

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

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

# Merge Sort

- Análisis

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

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

$$= 2^{\log n}T\left(\frac{n}{2^{\mathrm{lo}}}\right) + \log n\, n - 2^{\log n-1} - 2^{\log n-2} - \cdots - 2 - 1 =$$

$$= 2^{\log n}T(1) + \log n\, n - 2^{\log n-} \quad - 2^{\log n-} \quad - \cdots - 2 - 1 =$$

$$= 2^{\log n} + \log n\, n - 2^{\log n-1} - 2^{\log n-} \quad - \cdots - 2 - 1 =$$

# Merge Sort

- Análisis

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

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

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

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

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

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

# Merge Sort

- Análisis

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

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

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

$$= 2^{\log n} T(1) + \log n\, n - 2^{\log n} \quad - 2^{\log n - 2} - \cdots - 2 - 1 =$$

$$= 2^{\log n} + \log n\, n - 2^{\log n} \quad - 2^{\log n - 2} - \cdots - 2 - 1 =$$

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

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

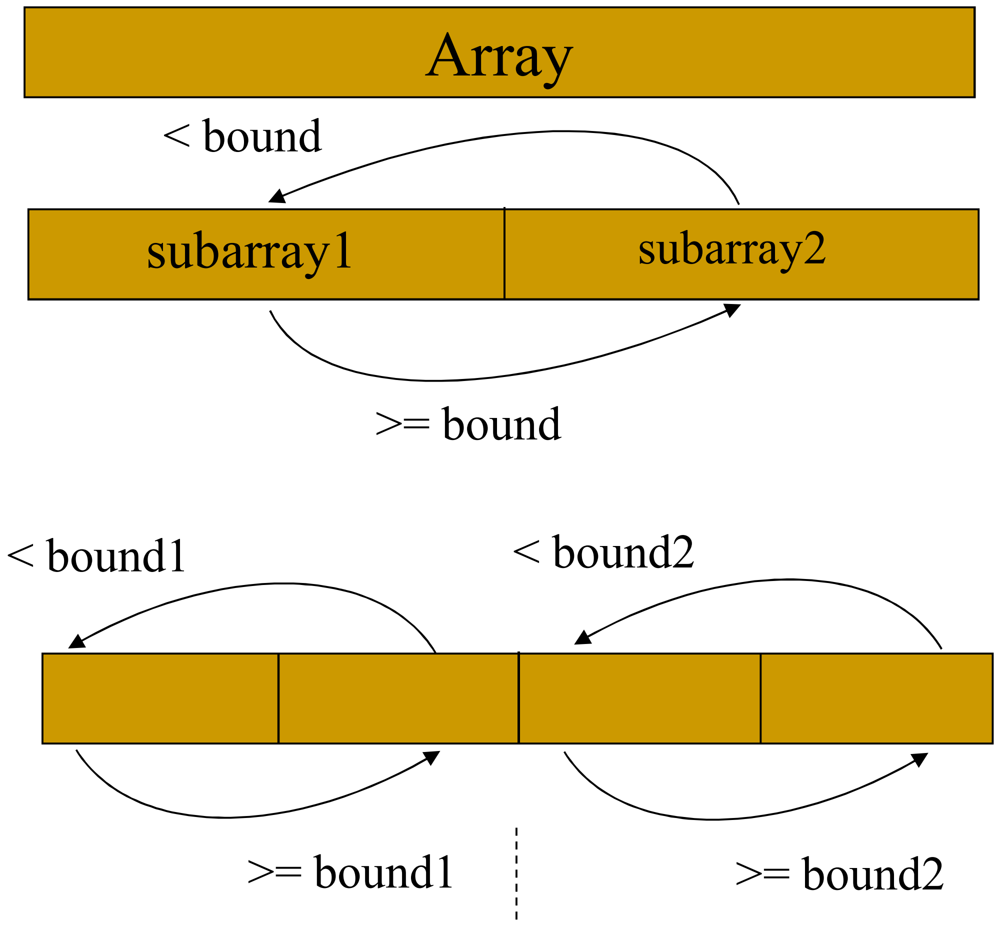
# Merge Sort

- Costo (suponiendo $n=2^k$)
  - $T(n)=2T(n/2)+(n-1) =$
  - $= 2(2T(n/4)+(n/2-1))+(n-1) =$
  - $= 4T(n/4)+2*n/2-2+n-1 =$
  - $= 4(2T(n/8)+n/4-1)+2n-2-1=$
  - $= 8T(n/8)+4*n/4-4+2n-2-1=8T(n/8)+3n-4-2-1=$
  - =..................................=
  - $2^iT(n/2^i)+i*n-2^{i-1}-2^{i-2}-........-2-1=$
  - =..................................= ¿hasta cuándo? Hasta $2^i=n$ o sea $i=\log n$
  - $= 2^{\log n}T(n/2^{\log n})+\log n*n-2^{\log n-1}-2^{\log n-2}-........-2-1=$
  - $= 2^{\log n}T(1)+\log n*n-2^{\log n-1}-2^{\log n-2}-........-2-1=$
  - $= 2^{\log n}+\log n*n-2^{\log n-1}-2^{\log n-2}-........-2-1=$
  - $= O(n \log n)$
- ¿Y si n no fuera exactamente igual a $2^k$?

# Quick Sort

- Idea en cierto modo parecida….(D&Q)
- Debido a C.A.R. Hoare
- 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



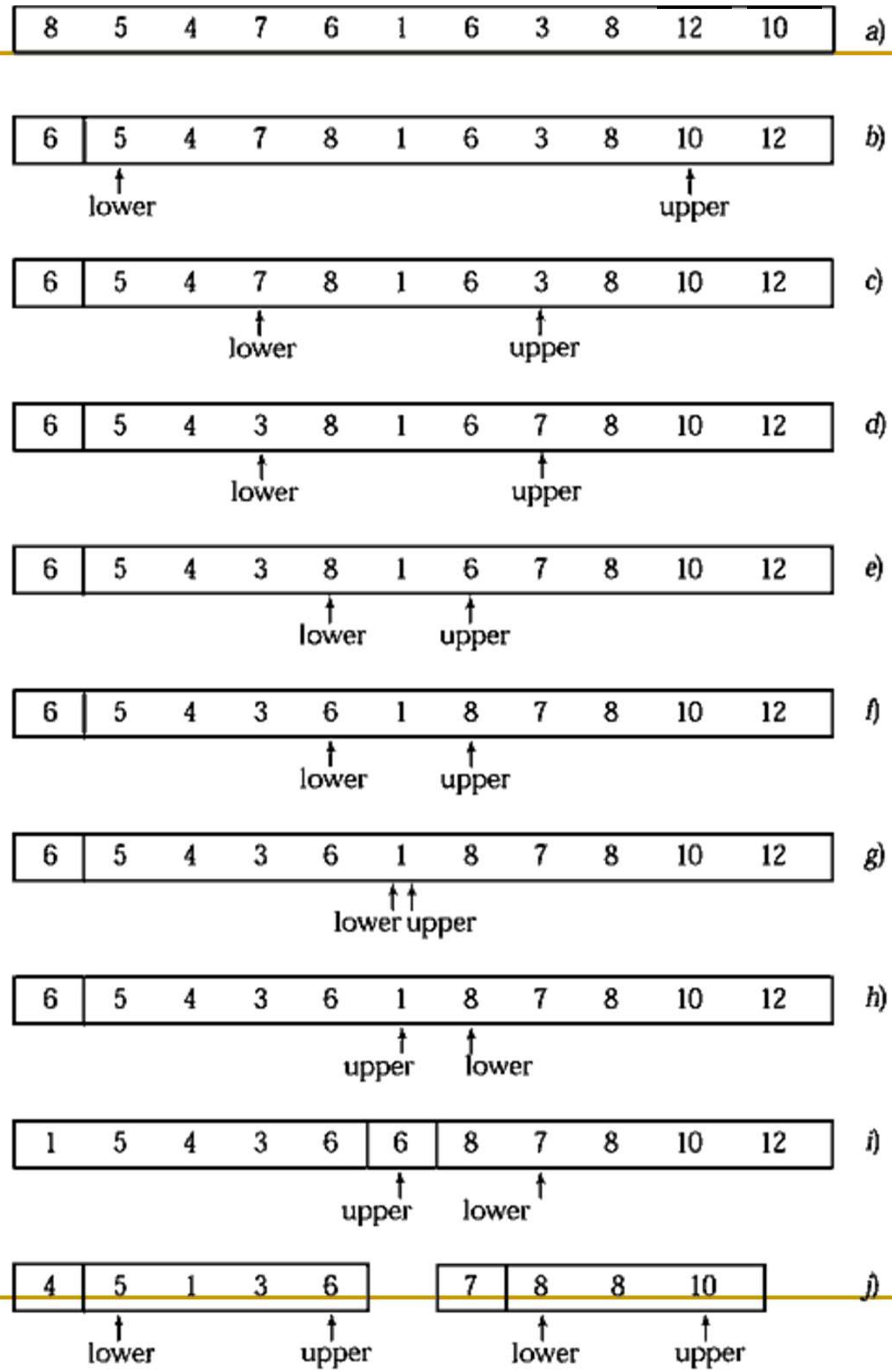

- ¿Y si no conocemos el elemento mediano?

# Quick Sort (en algún lenguaje)

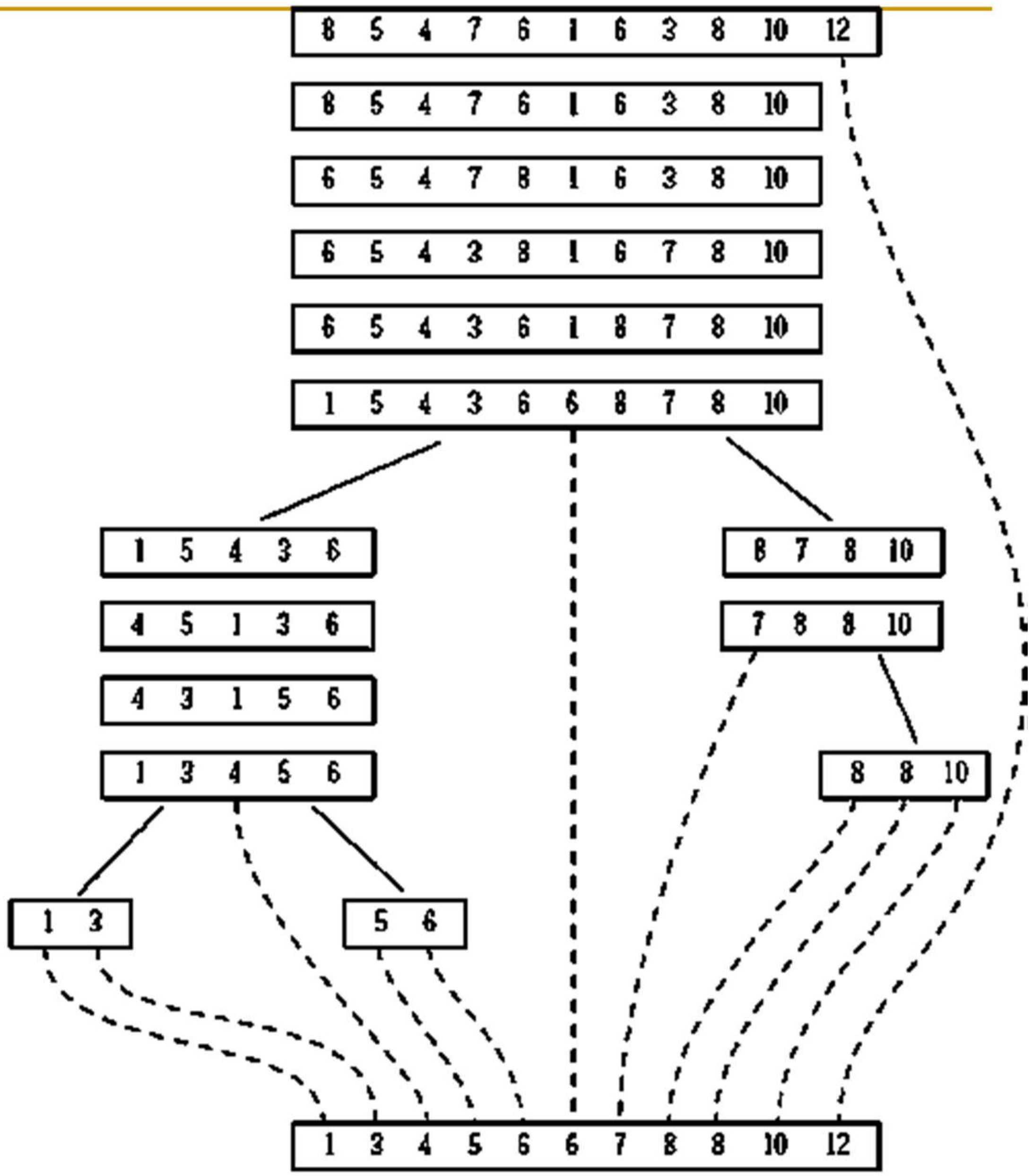
```
quicksort(array[]) {
    if (array.length>1) {
        Elegir bound; /* subarray1 y subarray2 */
        while (haya elementos en el array)
            if (elemento generico < bound)
                insertar elemento en subarray1;
            else insertar elemento en subarray2;
        quicksort(subarray1);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

particionamiento
del array
[8 5 4 7 6 1 6 3 8 12 10]
con quicksort

- Se busca el máximo y se lo ubica al final. Sirve para que lower no se pueda ir de rango
- Se elije como pivote el del medio y provisoriamente se lo ubica al inicio
- Cuando se cruzan lower y upper, se acaba la iteración



particionamiento
del array
[8 5 4 7 6 1 6 3 8 12 10]
con quicksort

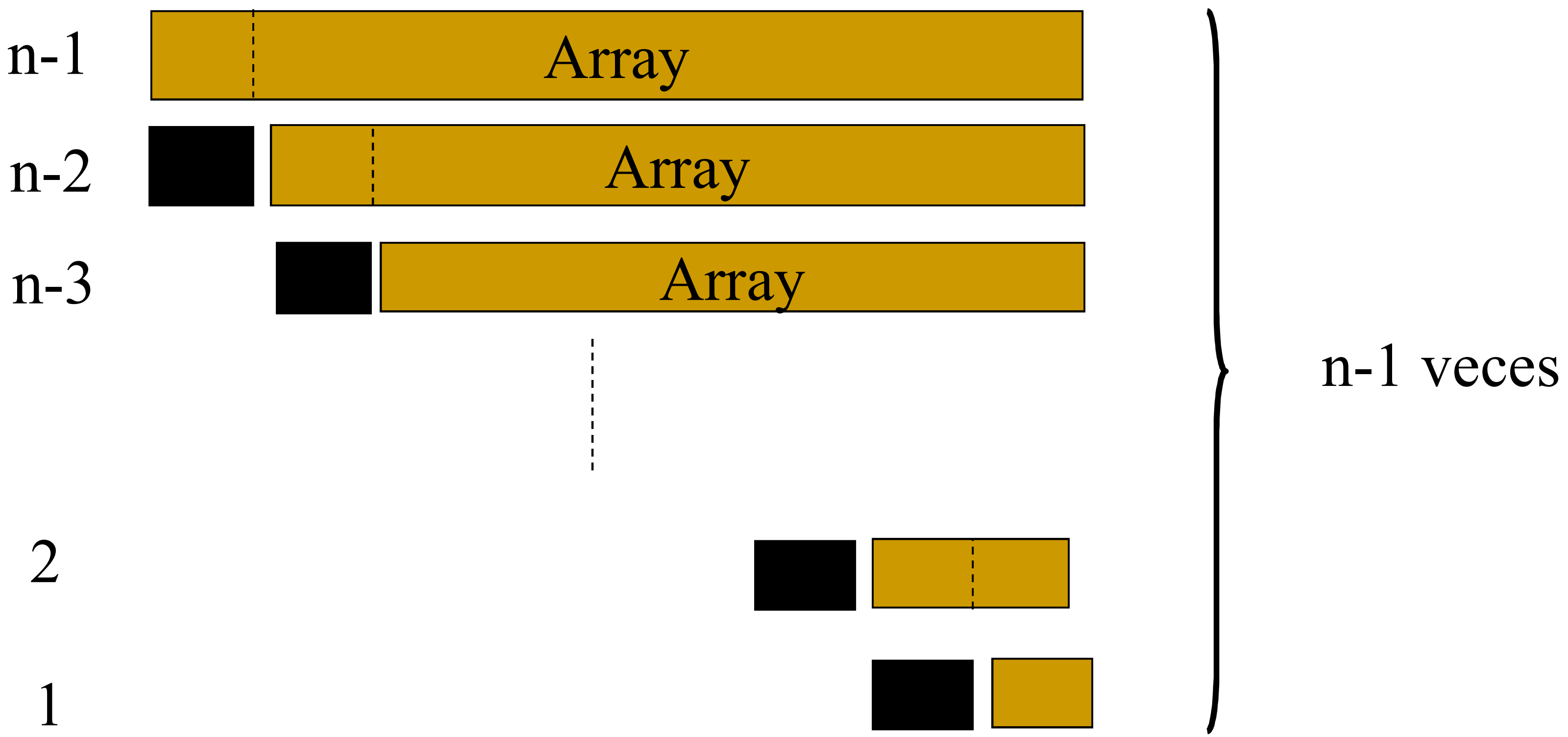


# Análisis de Quick Sort

- Costo = O(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



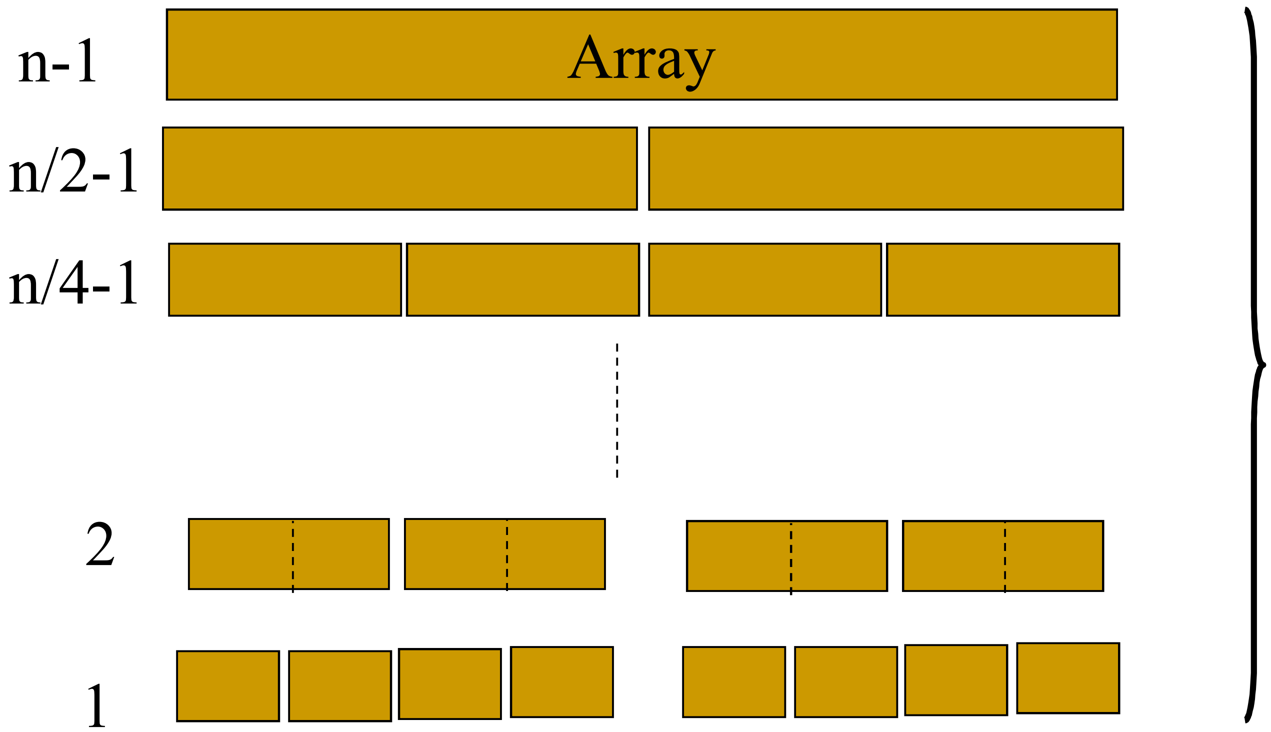

<u>El elemento pivot es siempre el mínimo</u>

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

# Quick Sort – Caso mejor

No. comparaciones
por sub-array

n potencia de 2 por simplicidad




log n+1 veces

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
- $T(n)=n+1+1/n\ \Sigma_{1\leq k\leq n}\ (T(k-1)+T(n-k))=$

  $= n+1+2/n\ \Sigma_{1\leq k\leq n}\ T(k-1) = O(n \log n)$
- Ojo, que no siempre podemos suponer que el input proviene de una distribución uniforme.
- Pero......¿y si lo "forzamos"?
- Permutando el input o bien...
- ¡Eligiendo el pivote al azar! (Algoritmos probabilísticos)

# Complejidad de los algoritmos de ordenamiento

- Merge Sort (y Heap 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

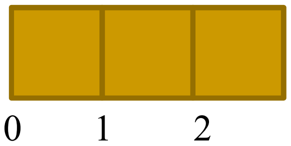
- Observación fundamental: todos los algoritmos deben comparar elementos (o sea, ese es nuestro modelo de cómputo)
- Dados $a_i$, $a_k$, tres casos posibles: $a_i < a_k$, $a_i > a_k$, o $a_i = a_k$
- Se asume por simplicidad que todos los elementos son distintos
- Se asume entonces que todas las comparaciones 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 $a_i <= a_k$

# Árboles de decisión

Búsqueda binaria de elemento *x* en un arreglo:

$n = 3$

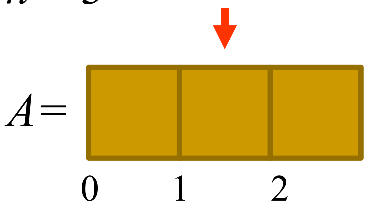
$A =$



# Árboles de decisión

Búsqueda binaria de elemento *x* en un arreglo:

$n = 3$



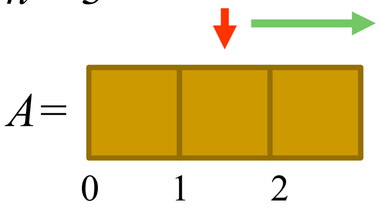


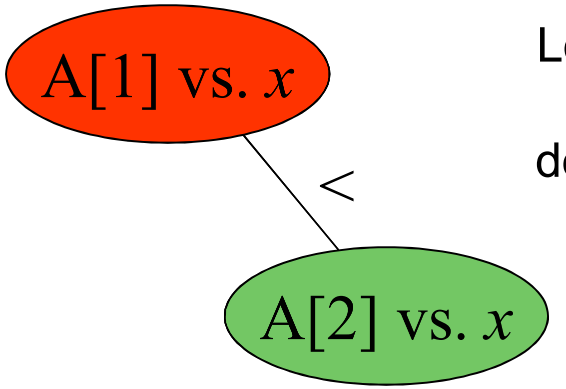
Los nodos internos representan decisiones binarias

# Árboles de decisión

Búsqueda binaria de elemento *x* en un arreglo:

$n = 3$

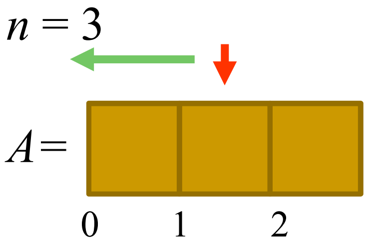


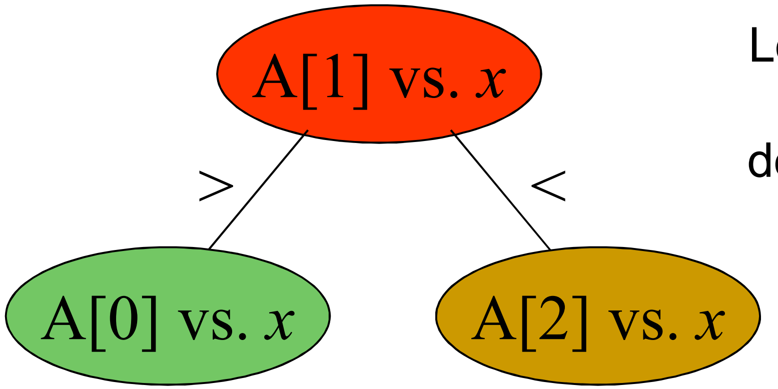




Los nodos internos representan decisiones binarias

# Árboles de decisión

Búsqueda binaria de elemento *x* en un arreglo:



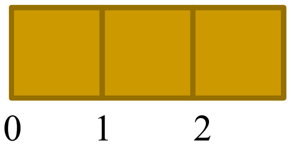



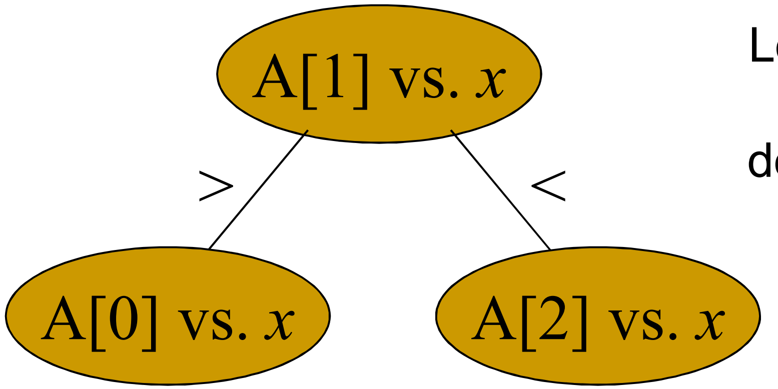

Los nodos internos representan decisiones binarias

# Árboles de decisión

Búsqueda binaria de elemento *x* en un arreglo:

$n = 3$

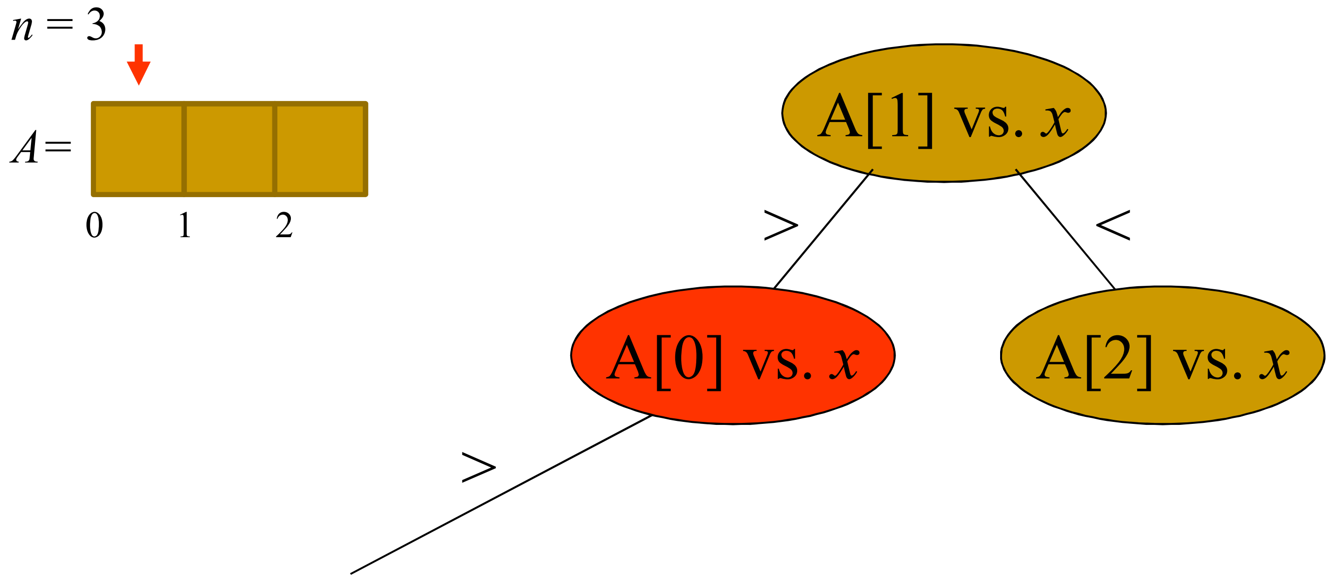
$A =$ 


Los nodos internos representan decisiones binarias

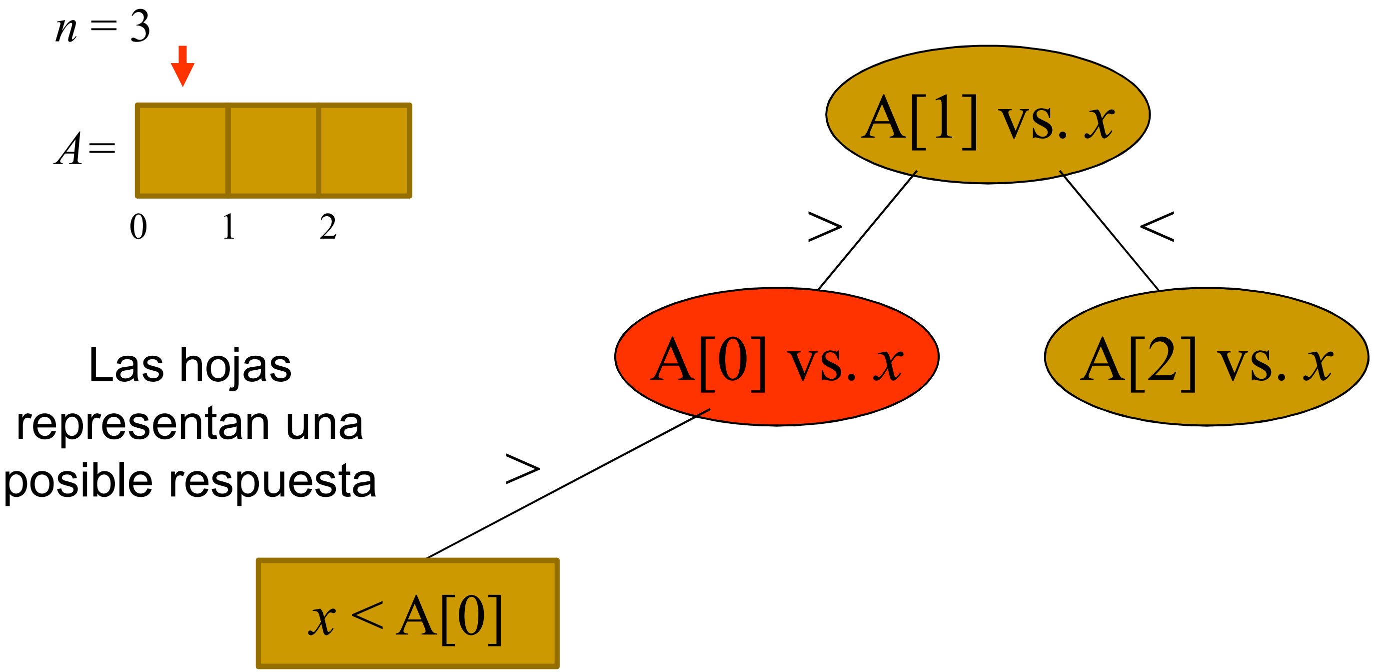
# Árboles de decisión

Búsqueda binaria de elemento *x* en un arreglo:



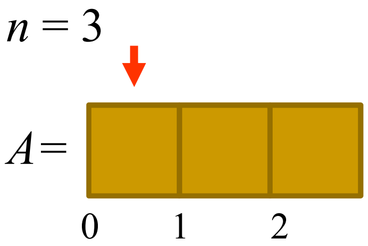
# Árboles de decisión

Búsqueda binaria de elemento *x* en un arreglo:

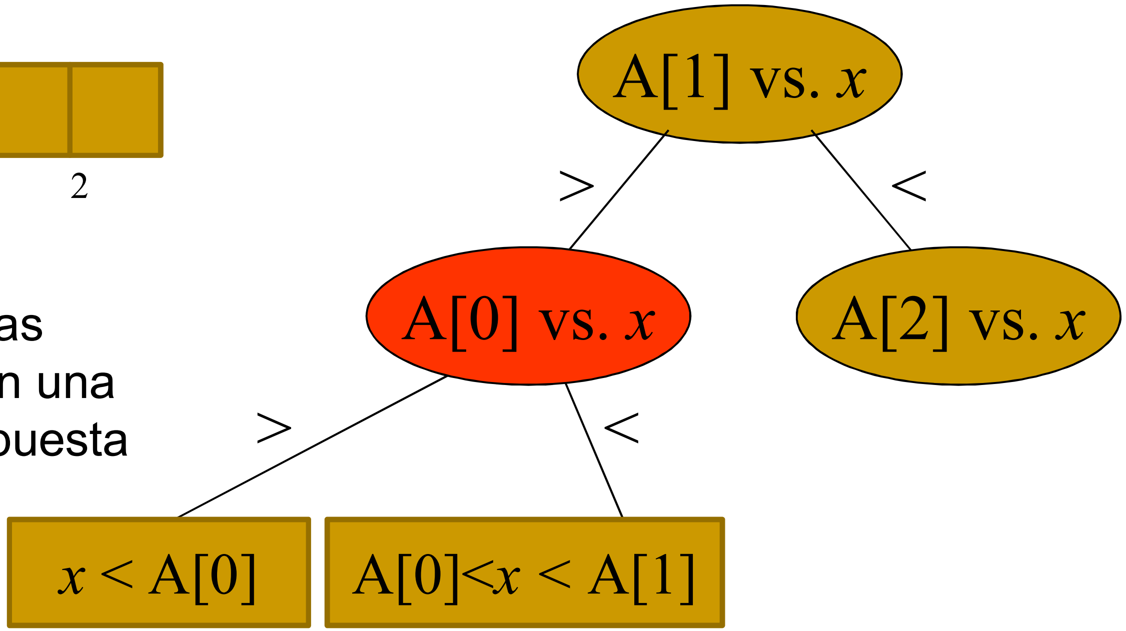


# Árboles de decisión

Búsqueda binaria de elemento *x* en un arreglo:

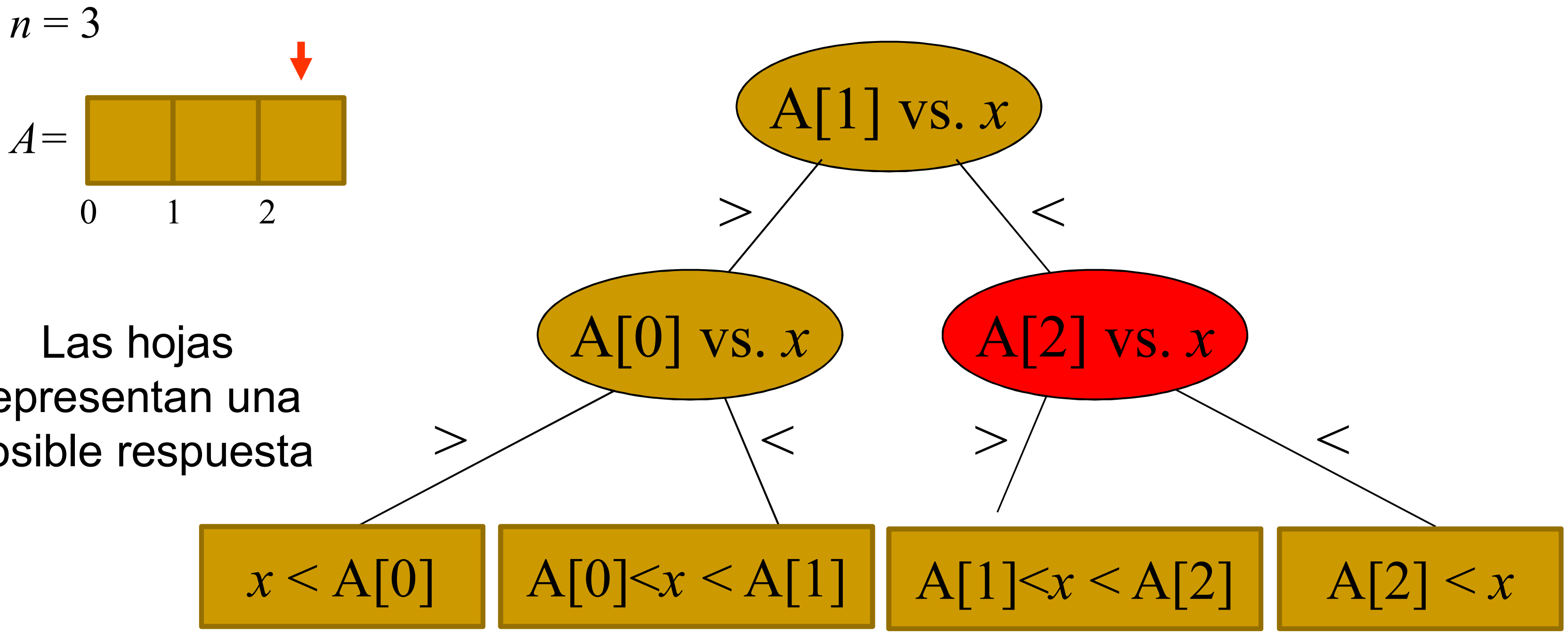



Las hojas representan una posible respuesta



# Árboles de decisión

Búsqueda binaria de elemento *x* en un arreglo:

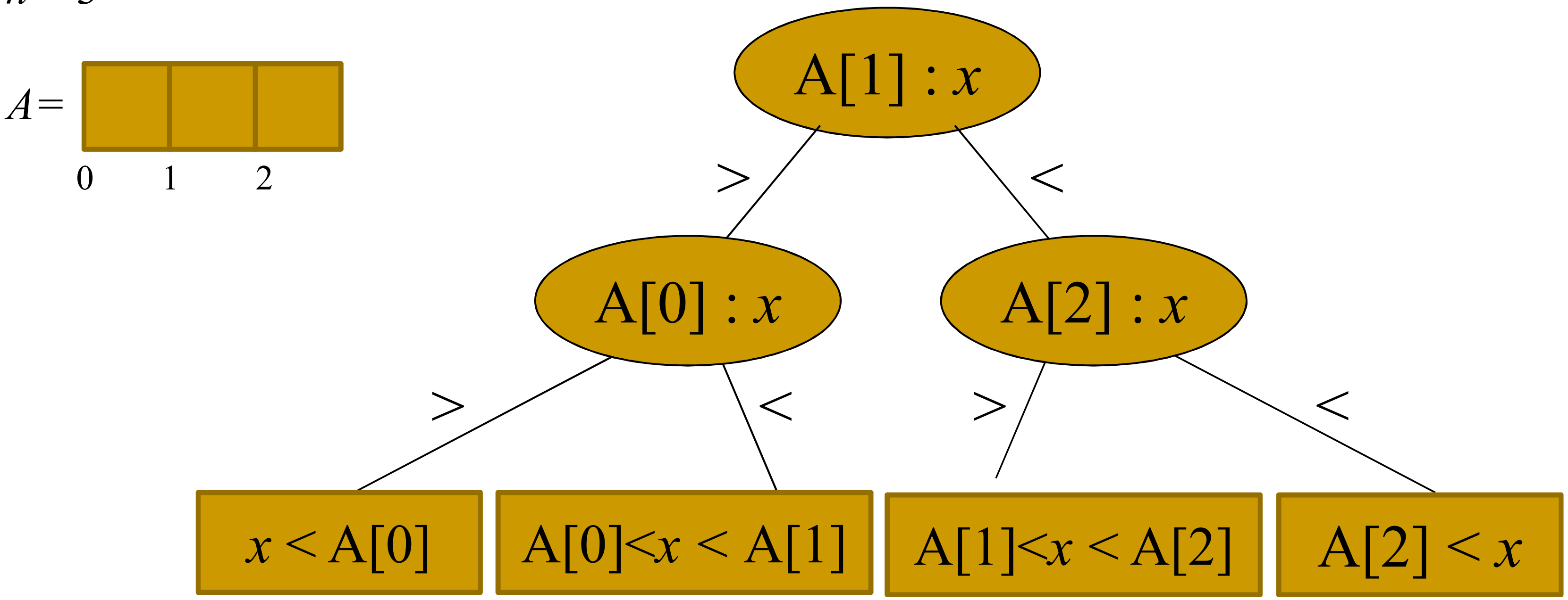



Las hojas representan una posible respuesta

# Árboles de decisión

Búsqueda binaria de elemento *x* en un arreglo:

$n = 3$

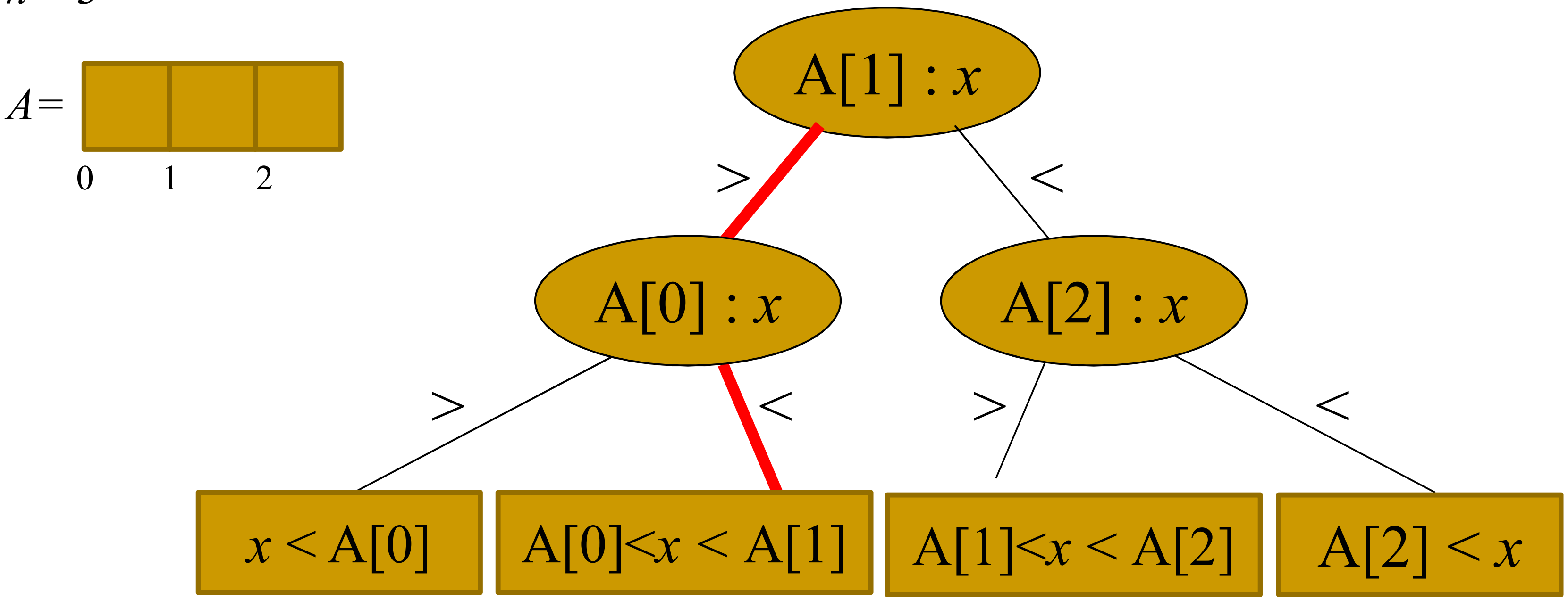



- Un árbol de decisión representa las comparaciones ejecutadas por un algoritmo sobre un input dado
- Cada hoja corresponde a una de las posibles outputs del algoritmo.

# Árboles de decisión

Búsqueda binaria de elemento *x* en un arreglo:

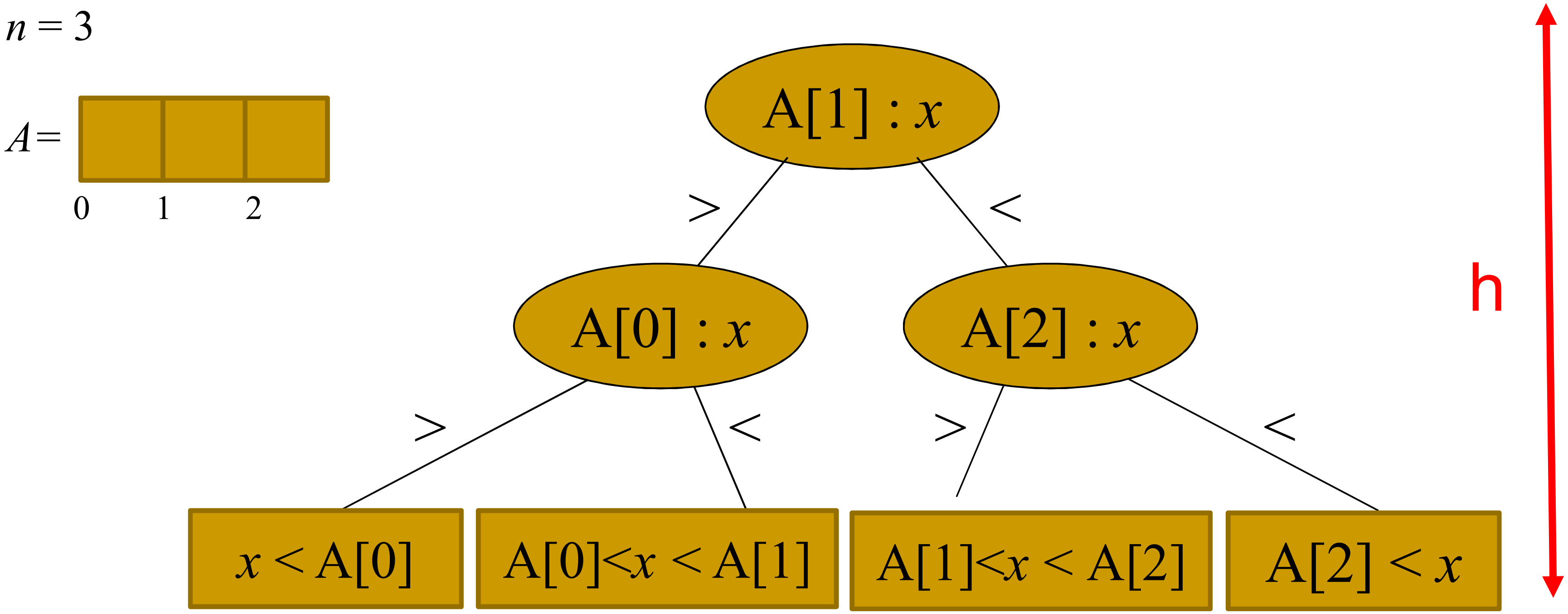
$n = 3$




- 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

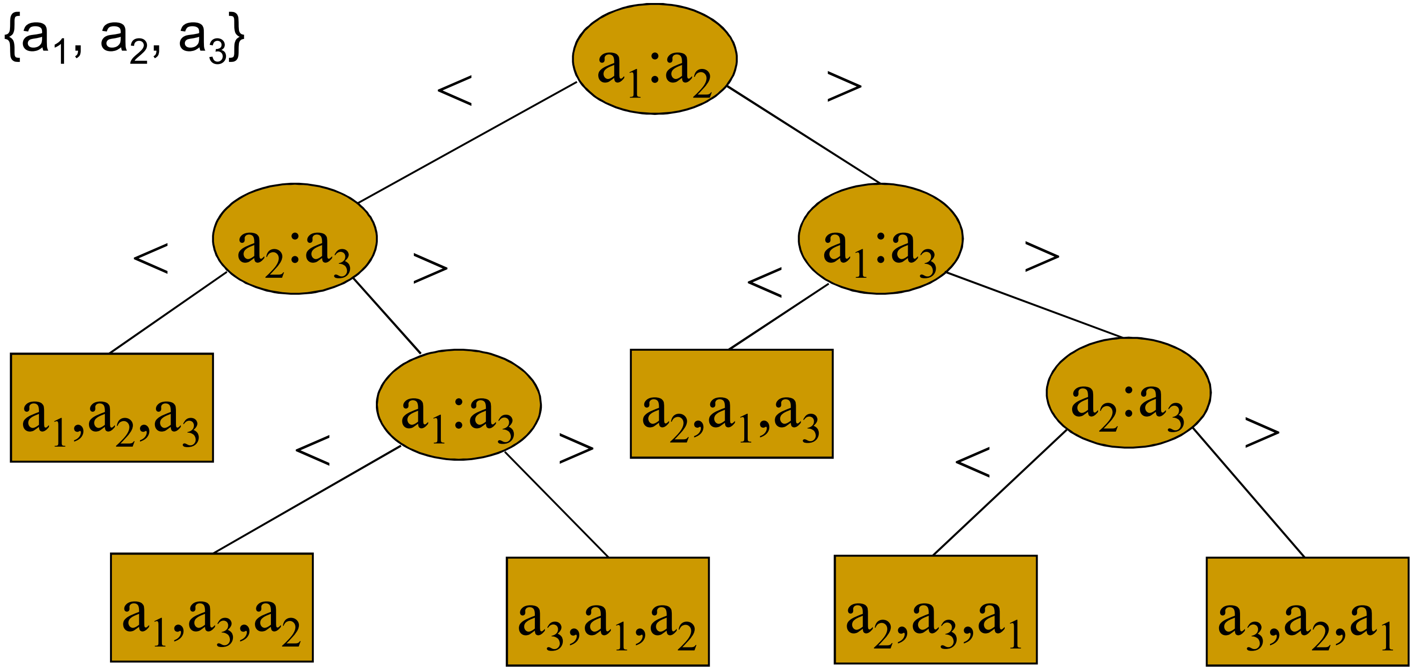
Búsqueda binaria de elemento *x* en un arreglo:




- 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

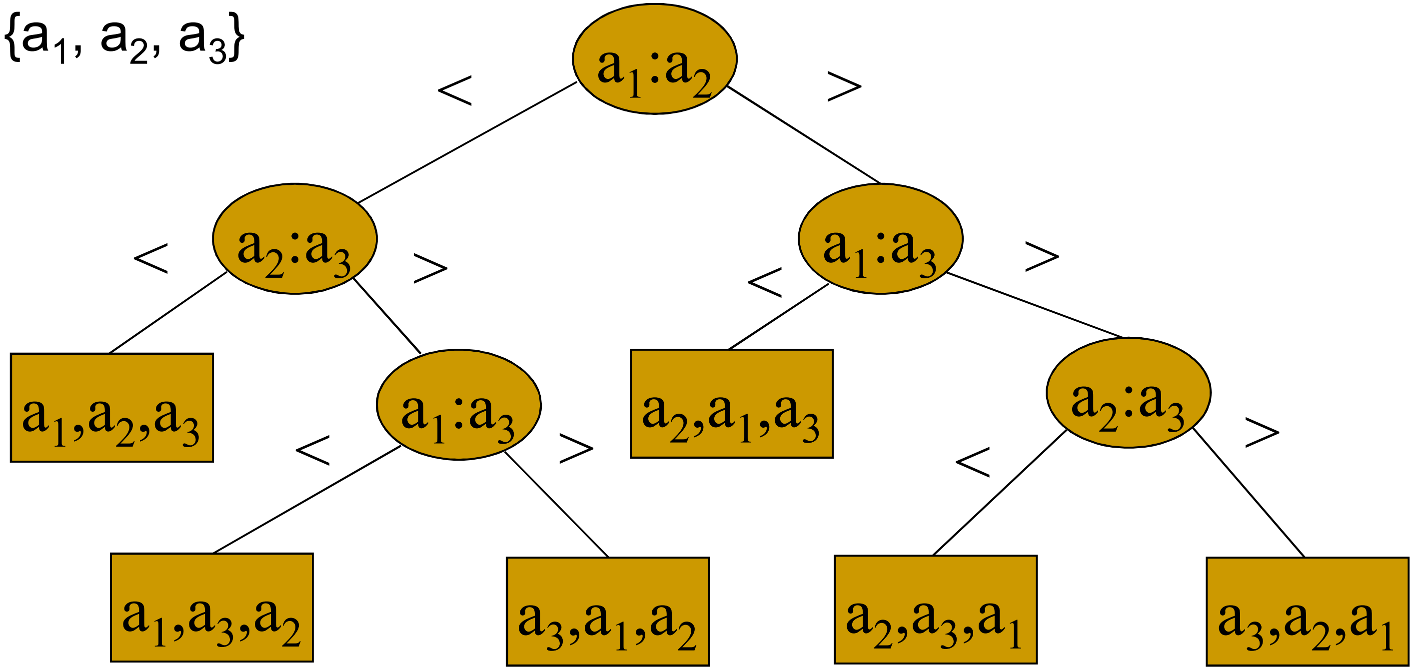
árbol de decisión sobre el conjunto $\{a_1, a_2, a_3\}$




- 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

árbol de decisión sobre el conjunto $\{a_1, a_2, a_3\}$




- Hay n! posibles permutaciones -> 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: los algoritmos Merge Sort y Heap 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

- Co
  cor
- Co
  tier
- No
  cor
  ext
  esis

**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/