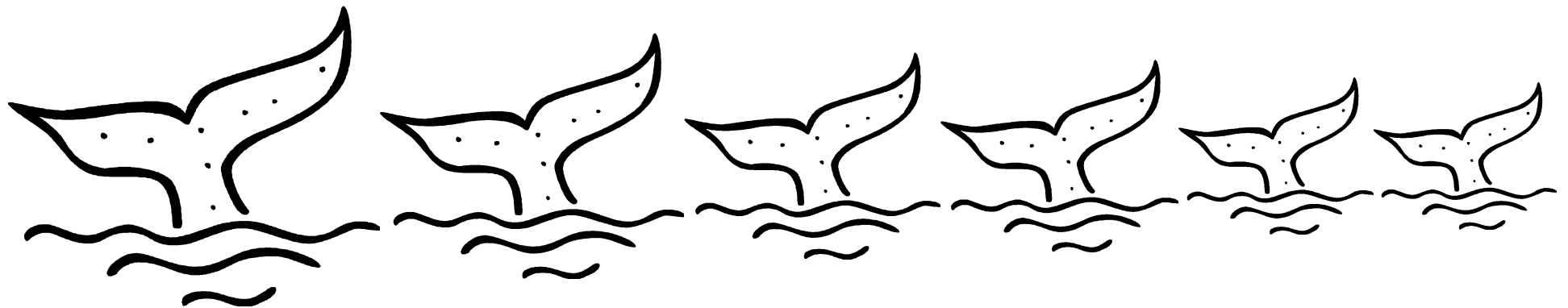

Ordenamiento (Sorting)



El problema del ordenamiento

- Ordenar: $\text{arreglo}[\alpha] \rightarrow \text{arreglo}[\alpha]$, donde α 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
 - $\text{min} \leftarrow \text{seleccionar_min}(i, n-1)$
 - $\text{Intercambiar}(i, \text{min})$

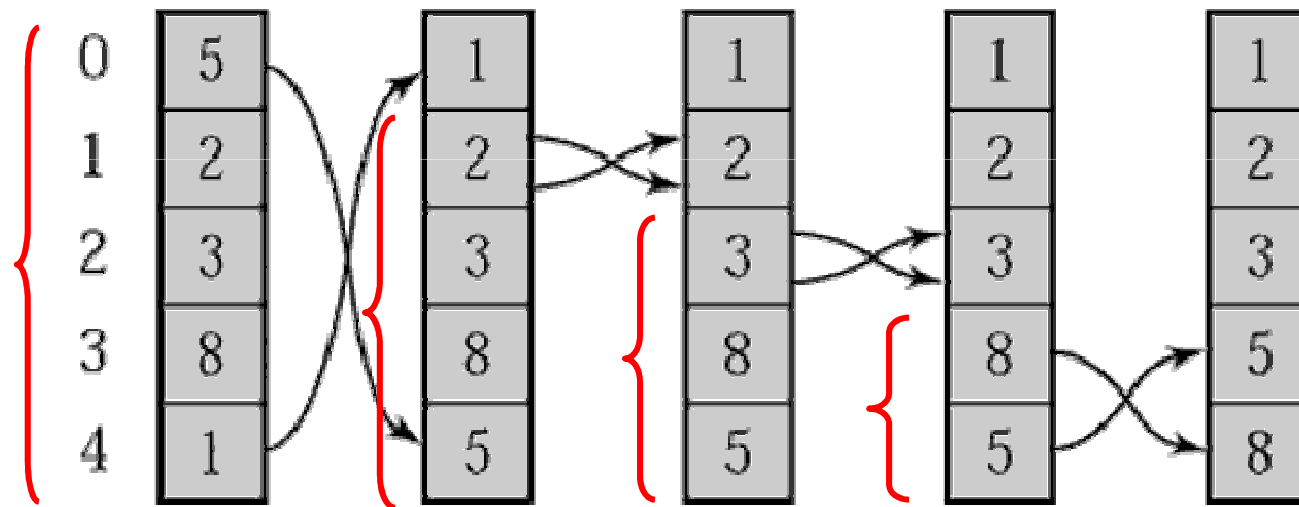
 - Para i desde 0 hasta $n-2$ hacer
 - $\text{min} \leftarrow i$
 - Para j desde $i+1$ hasta $n-1$ hacer
 - si $a[j] < a[\text{min}]$ entonces $\text{min} \leftarrow j$
 - $\text{Intercambiar}(i, \text{min})$
-

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

i	0	1	2	3	4
j	1, 2, 3, 4, 5	2, 3, 4, 5	3, 4, 5	4, 5	



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 $0..i$

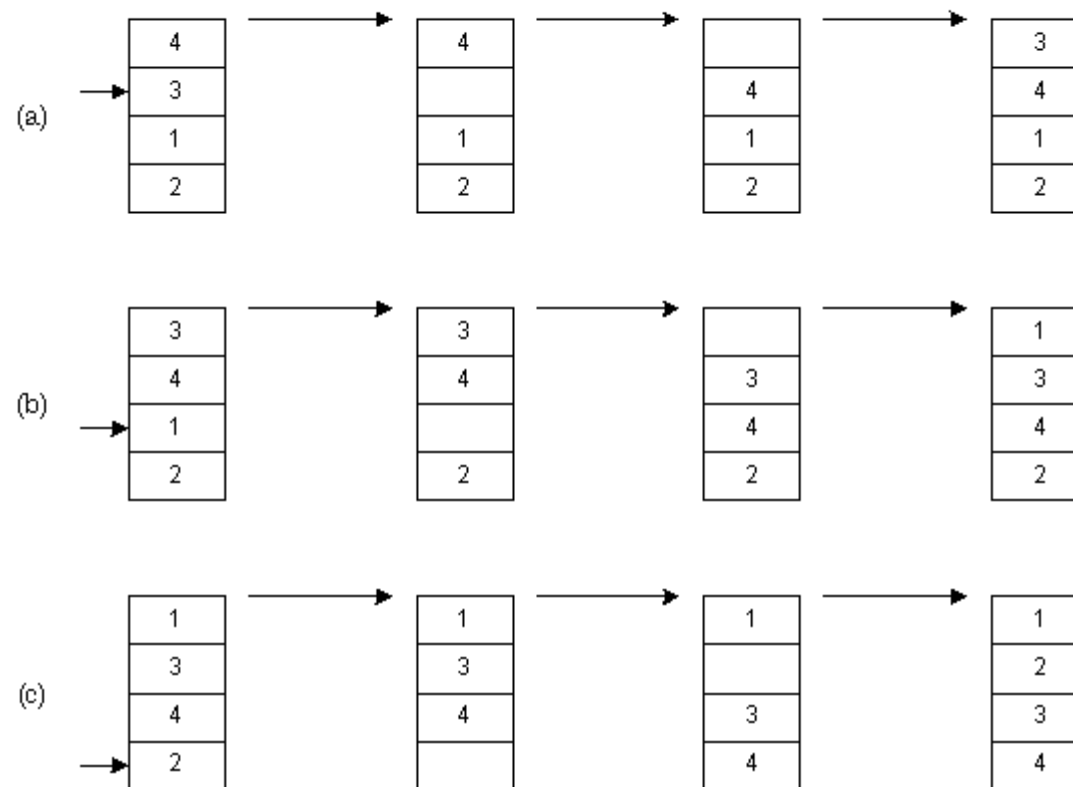


Insertion Sort

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

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

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?
-

HeapSort

- Valor de la 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
 - $\text{heap} \leftarrow \text{array2heap}(A)$
 - para i desde $n-1$ hasta 0 hacer
 - $\text{max} \leftarrow \text{próximo}(\text{heap})$
 - desencolar
 - $A[i] \leftarrow \text{max}$
 - 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 Reinárás”)
 - 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)
-

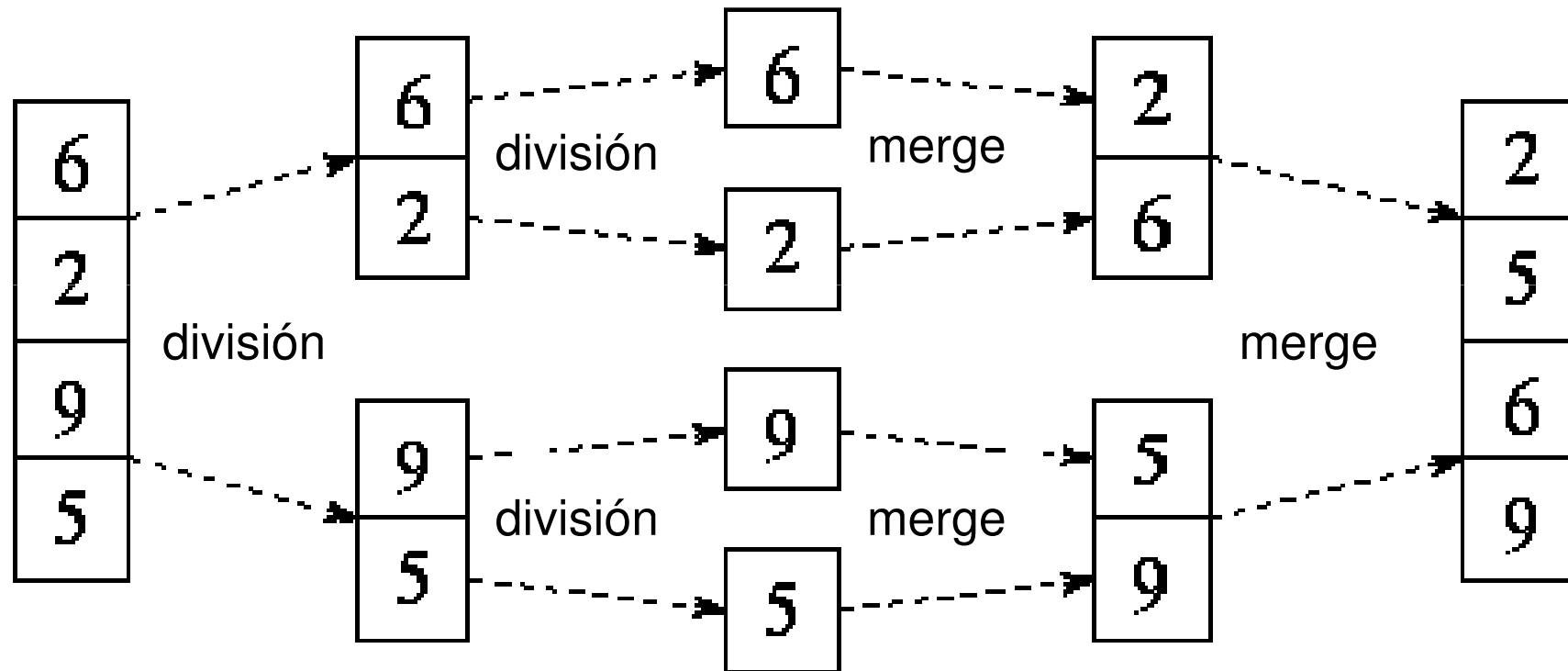
Merge Sort

■ 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



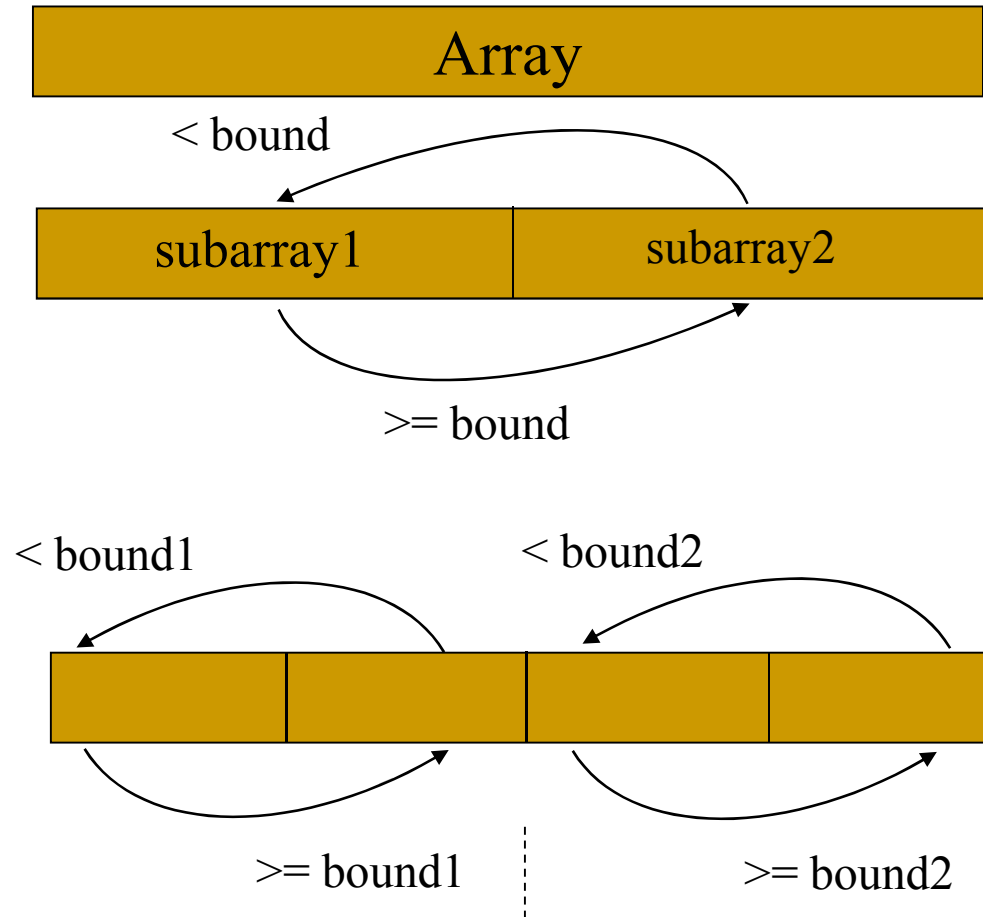
Merge Sort

- Costo (suponiendo $n=2^k$)
 - $T(n)=2T(n/2)+(n-1) =$
 - $= 2(2T(n/4)+(n/2-1))+(n-1) =$
 - $= 4T(n/4)+2*n/2-2+n-1 =$
 - $= 4(2T(n/8)+n/4-1)+2n-2-1=$
 - $= 8T(n/8)+4*n/4-4+2n-2-1=8T(n/8)+3n-4-2-1=$
 - $=.....=$
 - $2^i T(n/2^i)+i*n-2^{i-1}-2^{i-2}-.....-2-1=$
 - $=.....=$ ¿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 elige como pivote el del medio y provisoriamente se lo ubica al inicio
- Cuando se cruzan lower y upper, se acaba la iteración

a) [8 5 4 7 6 1 6 3 8 12 10]

b) [6 5 4 7 8 1 6 3 8 10 12]
↑ lower ↑ upper

c) [6 5 4 7 8 1 6 3 8 10 12]
↑ lower ↑ upper

d) [6 5 4 3 8 1 6 7 8 10 12]
↑ lower ↑ upper

e) [6 5 4 3 8 1 6 7 8 10 12]
↑ lower ↑ upper

f) [6 5 4 3 6 1 8 7 8 10 12]
↑ lower ↑ upper

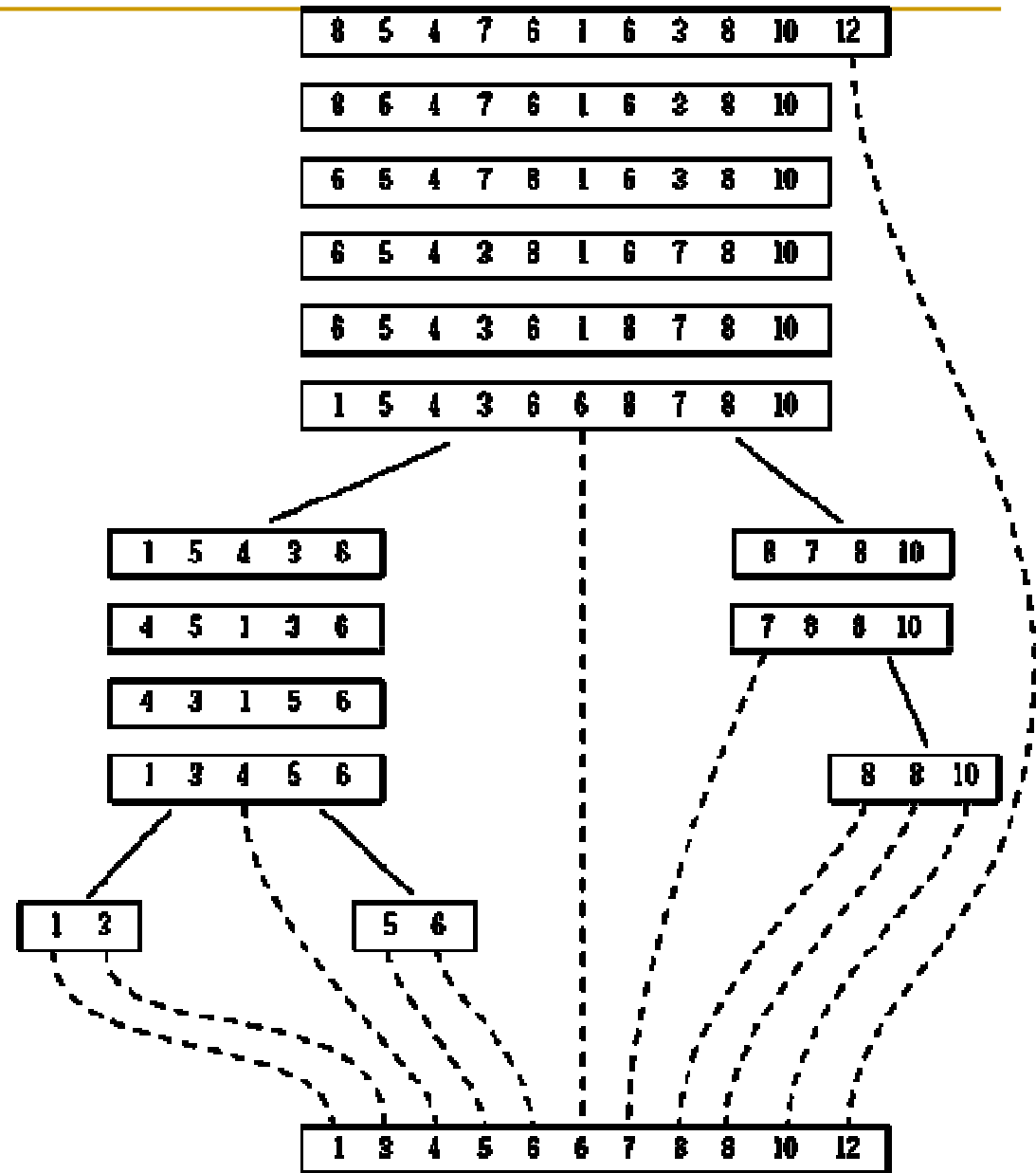
g) [6 5 4 3 6 1 8 7 8 10 12]
↑ lower ↑ upper

h) [6 5 4 3 6 1 8 7 8 10 12]
↑ upper ↑ lower

i) [1 5 4 3 6 | 6 | 8 7 8 10 12]
↑ upper ↑ lower

j) [4 5 1 3 6] [7 8 8 10]
↑ lower ↑ upper ↑ lower ↑ upper

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

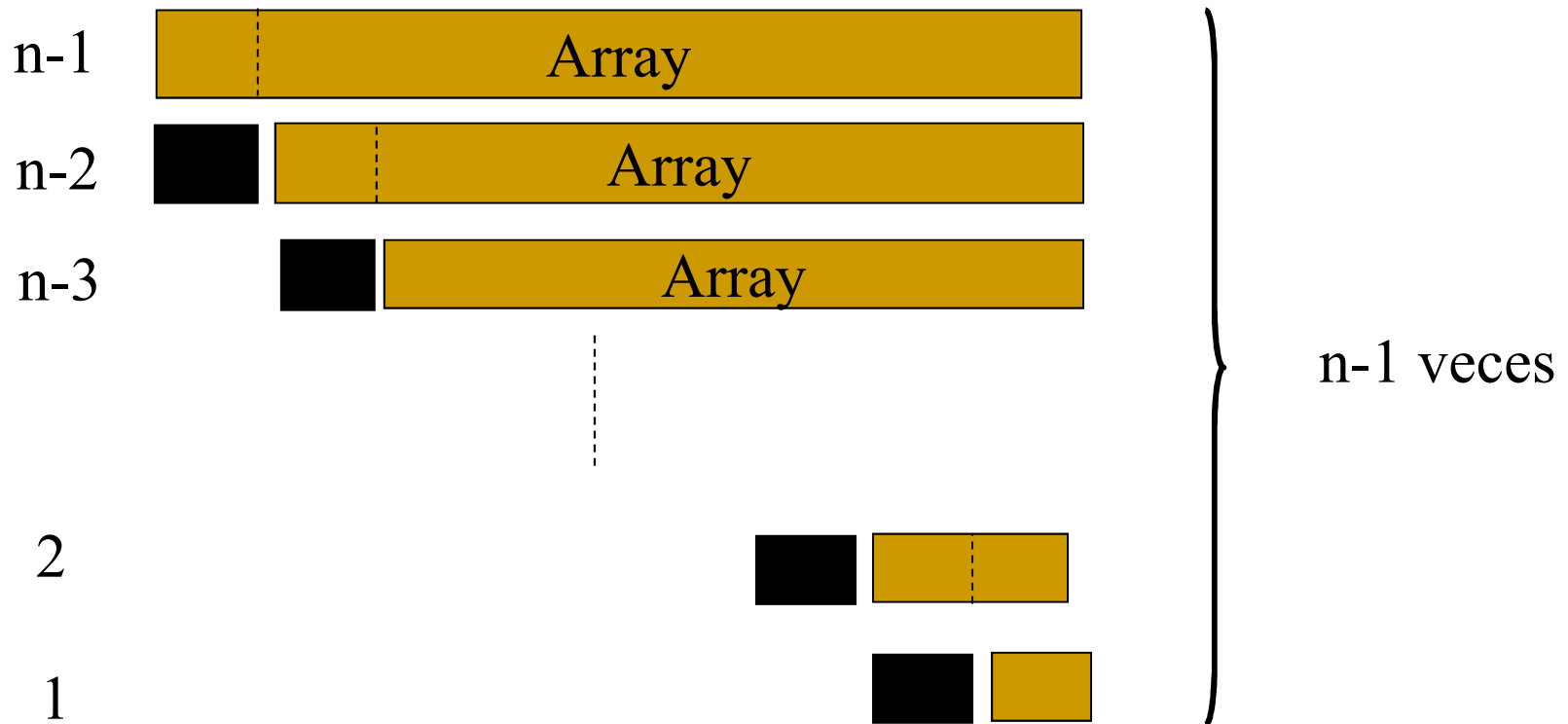


Análisis de Quick Sort

- ❑ 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

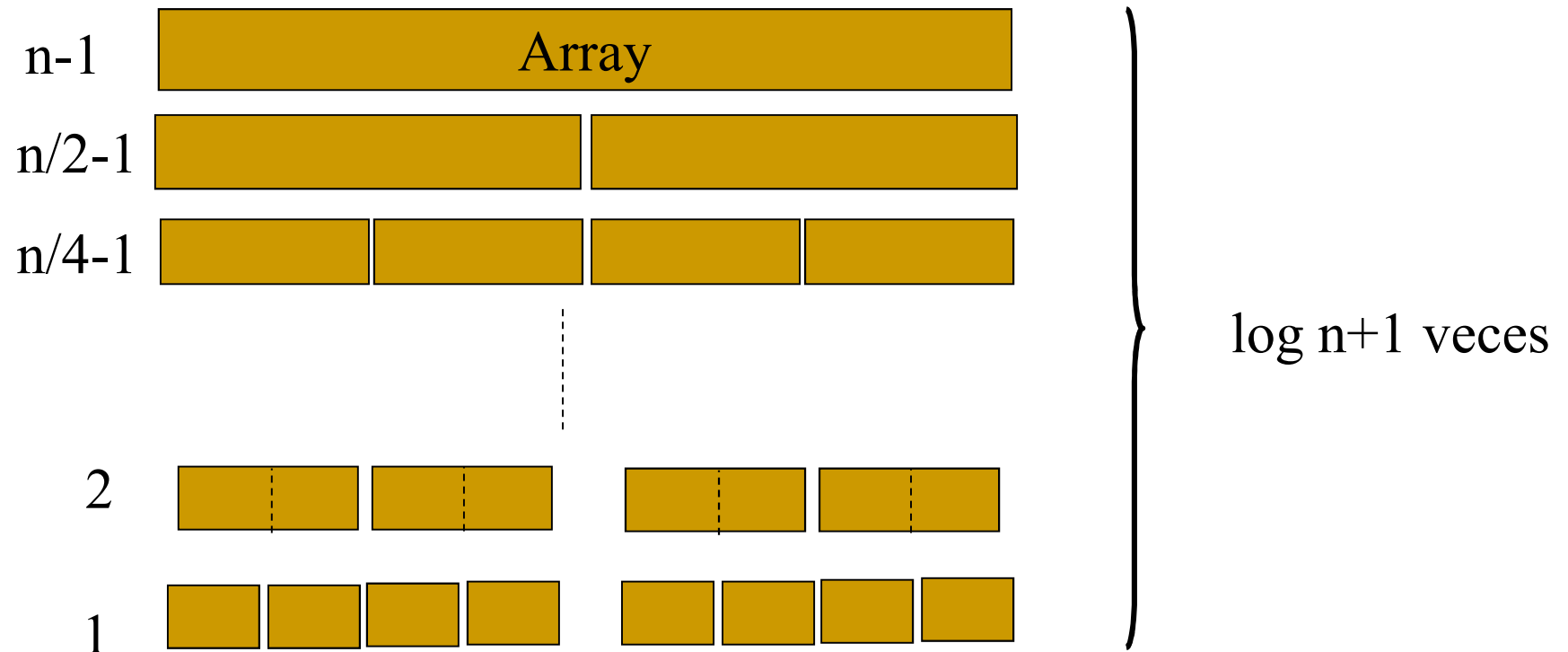


El elemento pivot es siempre el mínimo
Costo = $O(n-1+n-2+\dots+2+1) = O(n^2)$

Quick Sort – Caso mejor

No. comparaciones
por sub-array

n potencia de 2 por simplicidad



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

Complejidad de los algoritmos de ordenamiento

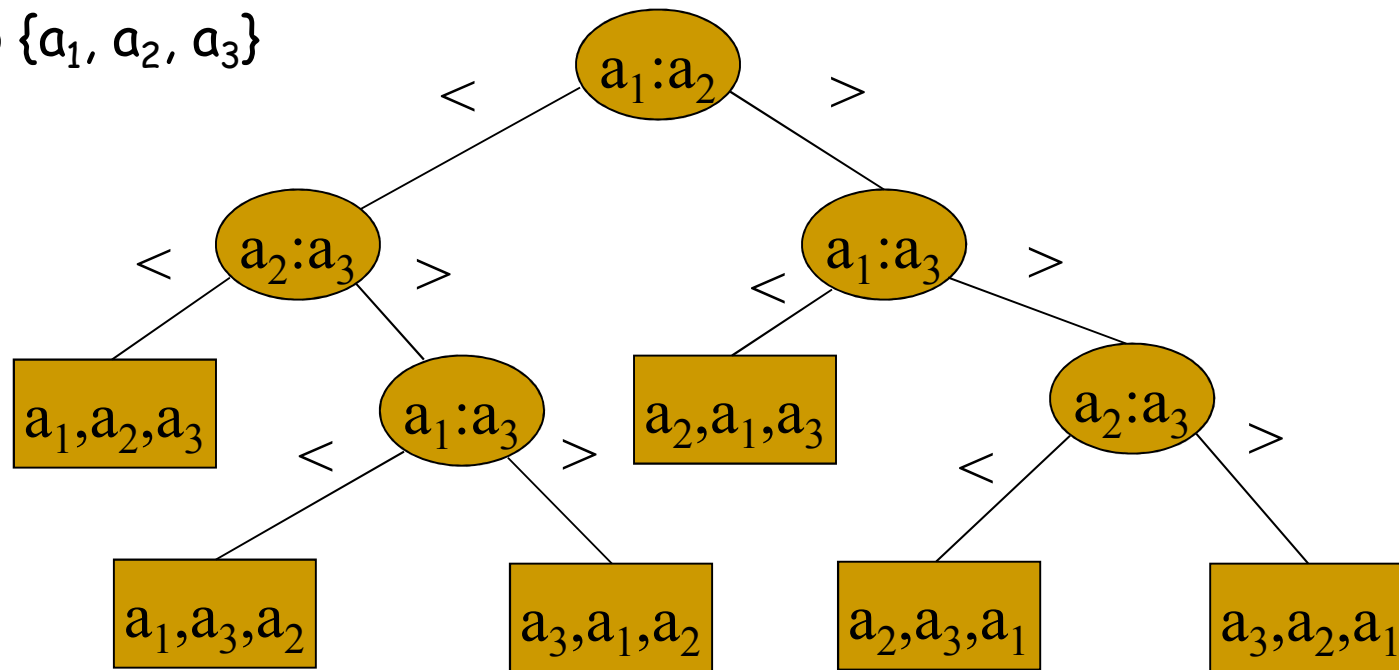
- ❑ Merge Sort (y Heap Sort): $O(n \log n)$
 - ❑ Quick Sort, Selection Sort, Insertion Sort: $O(n^2)$
 - Quick Sort: $O(n \log n)$ en el caso mejor
 - Selection Sort: $O(n^2)$ en todos los casos
 - Insertion Sort: $O(n)$ en el caso mejor
 - ❑ Pregunta: ¿cuál es la eficiencia máxima (complejidad mínima) 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 $a_i \leq a_k$

Árboles de decisión

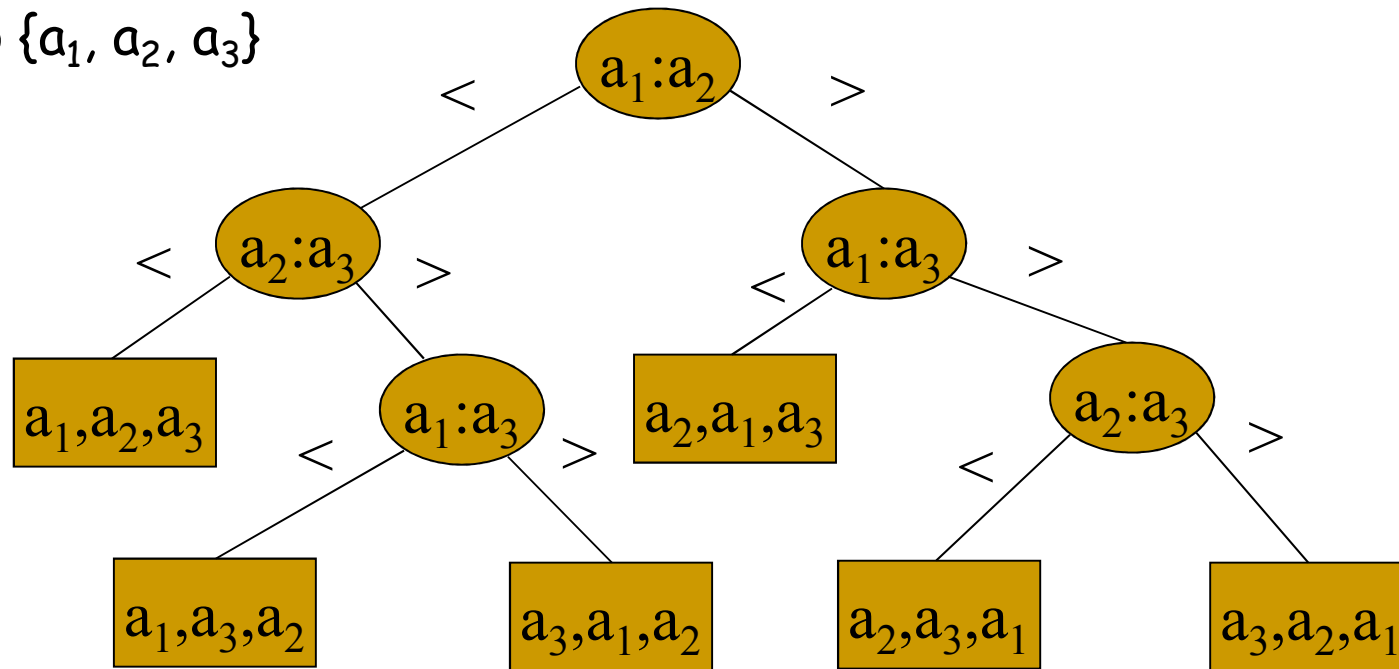
árbol de decisión sobre el conjunto $\{a_1, a_2, a_3\}$



- 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 \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: 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
-

demos

- Todos los métodos

- <http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html>

- Sólo merge sort

- <http://www.cs.toronto.edu/~neto/teaching/238/16/mergesort.html>

- En la página de la materia
