Algoritmos: **Sorting**

Algoritmos y Estructuras de Datos II

Motivación

Ya sabemos que, dado un criterio, ordenar un conjunto de datos según ese criterio puede permitirnos realizar tareas en forma muy eficiente.

El ejemplo clásico: la comparación entre la **búsqueda binaria** (posible sobre un arreglo ordenado) y la **búsqueda lineal**.

Selection sort (Idea)

Idea del algoritmo:

1.- Se asume que la parte inicial del arreglo está ordenada hasta la posición i

2.- Se busca el mínimo entre la posición i y la posición n-1 del arreglo A

3.- Se intercambia el mínimo con A[i]

4.- Se vuelve al paso **1.-** pero asumiendo que ahora el arreglo está ordenado hasta la posición i+1

A =

0	1	2	…	i	…	n-2	n-1
5	9	12	…	75	…	54	23

Ordenado [0..i)

j = Mínimo [i..n)

Intercambiar A[i] A[j]

Selection sort

Algoritmo (en pseudo C)**:**

```
void ordenar (int A[n]){
    for (int i = 0; i < n; i++){
        min = minimo (A, i);
        aux = A[i]; A[i] = A[min]; A[min] = aux;
    }
}
int minimo (int A[n], i){
    int min = 0;
    for (int j = i+1; i < n; j++)
        if (A[min] > A[j]) then min = j;
    return min;
}
```

Selection sort (complejidad)

$$\sum_{i=1}^{n} O(n-i) =$$

$$= O\left(\sum_{i=1}^{n} n-i\right) =$$

$$= O\left(\sum_{i=1}^{n} i\right) = O\left(\frac{n^2-n}{2}\right)$$

$$= O(n^2)$$

Insertion sort (Idea)

Idea del algoritmo:

1.- Se asume que la parte inicial del arreglo está ordenada hasta la posición i

2.- Se inserta el elemento A[i+1] ordenadamente en el subarreglo A[0..i]

3.- Se vuelve al paso **1.-** pero asumiendo que ahora el arreglo está ordenado hasta la posición i+1

A =

0	1	2	…	i+1	…	n-2	n-1
5	9	12	…	75	…	54	23

Ordenado [0..i)

Insertar A[i+1] A[0..i]

Insertion sort

Algoritmo (en pseudo C)**:**

```
void ordenar (int A[n]){
    for (int i = 0; i < n; i++)
        insertar (A[0..i+1], A[i+1]);
}

void insertar (int A[n], a){
    int aux = a, i;
    for (i = 0; i < n; i++)
        if (A[i] > a) then {
            aux = A[i]; A[i] = a;
        }
    A[i] = aux;
}
```

Insertion sort (complejidad)

$$O\left(\sum_{i=1}^{n} i\right) = O\left(\frac{n^2 - n}{2}\right)$$

$$= O(n^2)$$

Bubble sort (Idea)

Idea del algoritmo:

1.- Se asume que la parte final del arreglo está ordenada desde la posición i

2.- Se mueve el elemento más grande de A[0..i-1] hasta A[i-1]

3.- Se vuelve al paso **1.-** pero asumiendo que ahora el arreglo está ordenado desde la posición i-1

A =

0	1	2	…	i	…	n-2	n-1
5	9	12	…	23	…	54	75

Ordenado [i..n)

Mover mayor A[0..i-1]

Bubble sort

Algoritmo (en pseudo C)**:**

```
void ordenar (int A[n]){
    for (int i = n; i < n; i--)
        for (int j = 0; j < i; j++){
            if (A[j] > A[j+1]) then
                aux = A[j]; A[j] = A[j+1]; A[j+1] = aux;
}
```

Bubble sort (complejidad)

$$\sum_{i=1}^{n} O(n-i) =$$

$$= O\left(\sum_{i=1}^{n} n-i\right) =$$

$$= O\left(\sum_{i=1}^{n} i\right) = O\left(\frac{n^2-n}{2}\right)$$

$$= O(n^2)$$

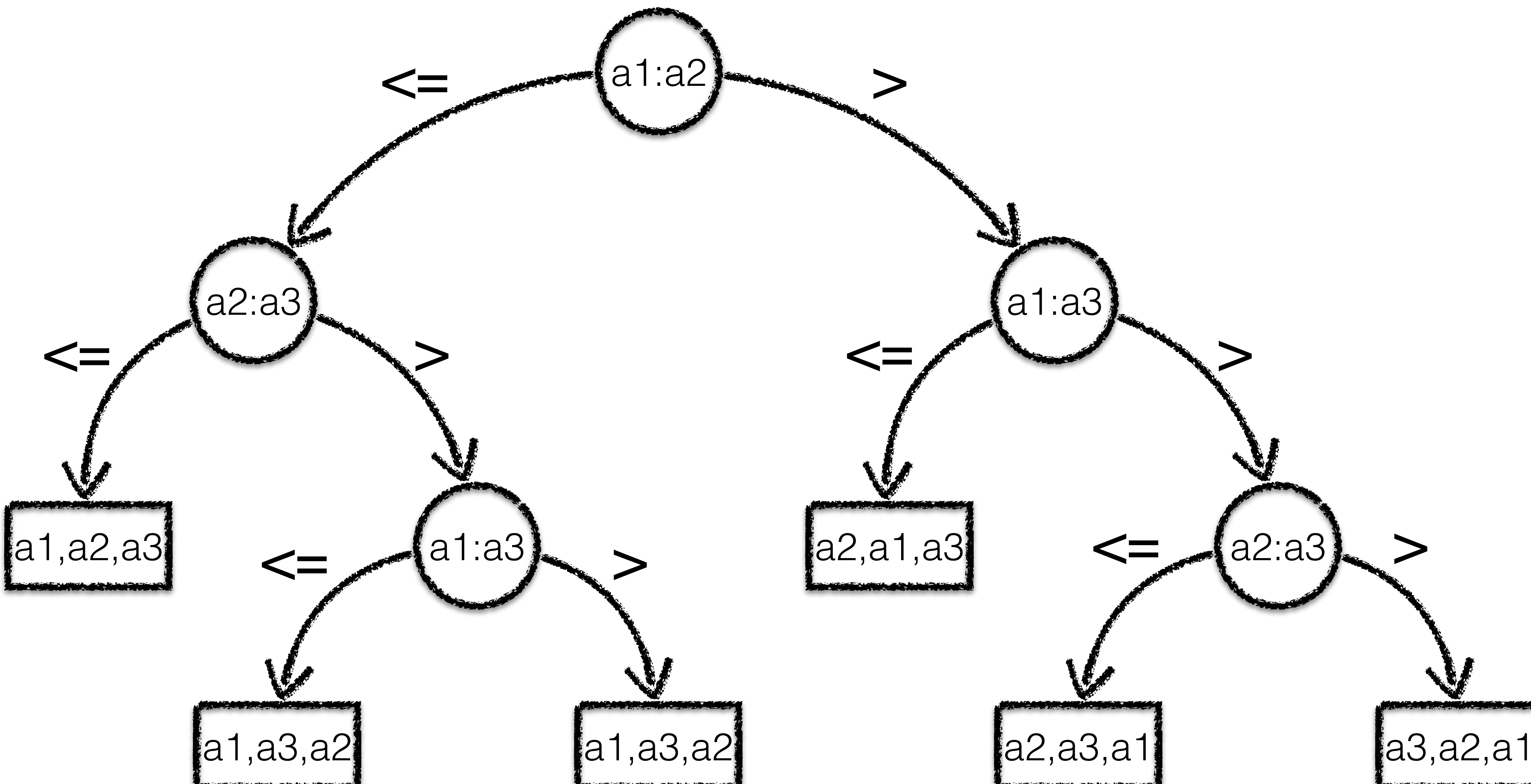
Propiedades

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?
- ¿Cómo pueden ser adaptados para que sean estables?¿De qué elementos depende esta adaptación?

Arboles de decisión
Cota para el problema de sorting

Ordenar el arreglo [a1, a2, a3] según la relación <=



Arboles de decisión
Cota para el problema de sorting

Ordenar el arreglo [a1, …, an] según la relación <=

* El árbol representa todas las comparaciones necesarias entre elementos del arreglo en forma ordenada de forma que su altura es mínima (ver ejemplo anterior)
* Cada hoja del árbol es una permutación del arreglo original según un orden particular. Hay n! permutaciones
* La ejecución de un algoritmo de ordenamiento describe una forma de recorrer un camino en el árbol desde la raíz a un hoja
* La altura del árbol representa la cota inferior para la cantidad de comparaciones necesaria en peor caso (i.e. Ω)

Arboles de decisión
Cota para el problema de sorting

Ordenar el arreglo [a1, …, an] según la relación <=

Teorema: El árbol de decición para ordenar un arreglo de n elementos tiene altura $\Omega(n * \log n)$.

Demostración: El árbol de decisión es binario, balanceado y tiene n! hojas.

$$\log n! = \sum_{i=1}^{n} \log i$$
$$\leq n \log n \in \Omega(n \log n)$$

Corolario: No hay ningún algoritmo con orden de complejidad temporal menor a $\Omega(n * \log n)$.

Quick sort

- El algoritmo Quick Sort es un clásico ejemplo de la aplicación de la técnica “Divide & Conquer” (del latín Divide et impera atribuida a “Felipe” II de Macedonia, padre de Alejandro Magno)
- Introducido por C.A.R. Hoare en 1959 mientras visitaba como estudiante la URSS.
- Se trata de partir el problema de ordenar un arreglo de n elementos en el problema de pegar dos arreglos ordenados de n/2 elementos (tal que uno tiene los elementos más chicos y el otro los más grandes).

Quick sort

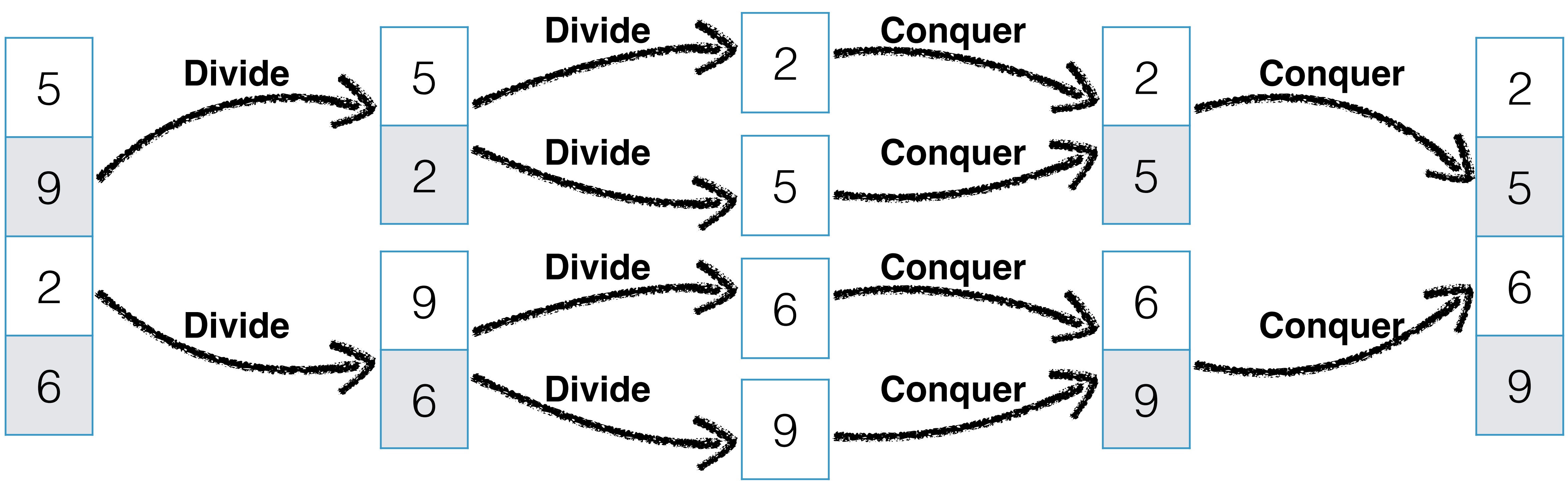
Idea del algoritmo:

1.- Si el arreglo tiene longitud menor o igual a 1, está ordenado y por lo tanto se retorna sin hacer nada

2.- [Divide] En caso contrario, se toma el k-ésimo elemento de A y se divide A[1..n] en dos A1[1..pk] conteniendo todos los elementos de A menores o iguales a A[k] y A2[pk+1..n] conteniendo todos los elementos de A mayores a A[k]

3.- [Conquer] A[1..n] = A1[1..n/2] ++ A2[n/2+1..n]

k=1



Quick sort

Algoritmo (en pseudo C)**:**

```
void ordenar (int A[n]){
    if (n < 2) then return;
    else {
        int pk = posicion (A[k]);
        int A1[1..pk], A2[p/k+1..n];
         A1 = menoresOIguales (A[1..n], A[k]);
         A2 = mayores (A[1..n], A[k]);
         ordenar (A1); ordenar (A2);
         A[1..pk] = A1; A[pk+1..n] = A2;
    }
}
```

Quick sort (complejidad)

- En **peor** caso el algoritmo Quick sort tiene orden de complejidad temporal O(n^2). La razón es que podríamos tener la mala suerte de que el pivote es siembre el elemento más chico o más grande.

$$O\left(\sum_{i=1}^{n} i\right) = O\left(\frac{n^2-n}{2}\right)$$

$$= O(n^2)$$

Quick sort (complejidad)

- En **mejor** caso el algoritmo Quick sort tiene orden de complejidad temporal O(n * log n). La razón es que podríamos tener la buena suerte de que el pivote es siembre el elemento mediano.

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

$$f(n) = n$$

Caso 2) Teorema Maestro

$$f(n) \in \Theta(n^c \log^k n)$$

$$c = \log_2 2$$

$$k = 0$$

$$T(n) \in \Theta(n \log n)$$

Quick sort (complejidad)

- ¿Qué análisis podemos hacer para el caso general dado que no podemos elegir el pivote eficientemente?

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

$$= \frac{2}{n}\sum_{i=1}^{n} T(i) + n$$

$$T(n) \in \Theta(2n \log n)$$

$$\in \Theta(n \log n)$$

Ver texto adjunto

Quick sort (propiedades)

- El argumento anterior no siempre puede usarse; solo cuando la muestra es uniforme

- Una solución es “randomizar” el arreglo antes de partirlo; esto prácticamente garantiza el orden O(n * log n) pero aumenta la constante

- No es necesario usar arreglos auxiliares ya que es posible implementar Quick sort in place

Merge sort

- El algoritmo Merge Sort también es un ejemplo de la aplicación de la técnica “Divide & Conquer”; si se quiere más claro.
- Atribuído por Donald Knuth a John von Newman
- Se trata de partir el problema de ordenar un arreglo de n elementos en el problema de combinar dos arreglos ordenados de n/2 elementos.

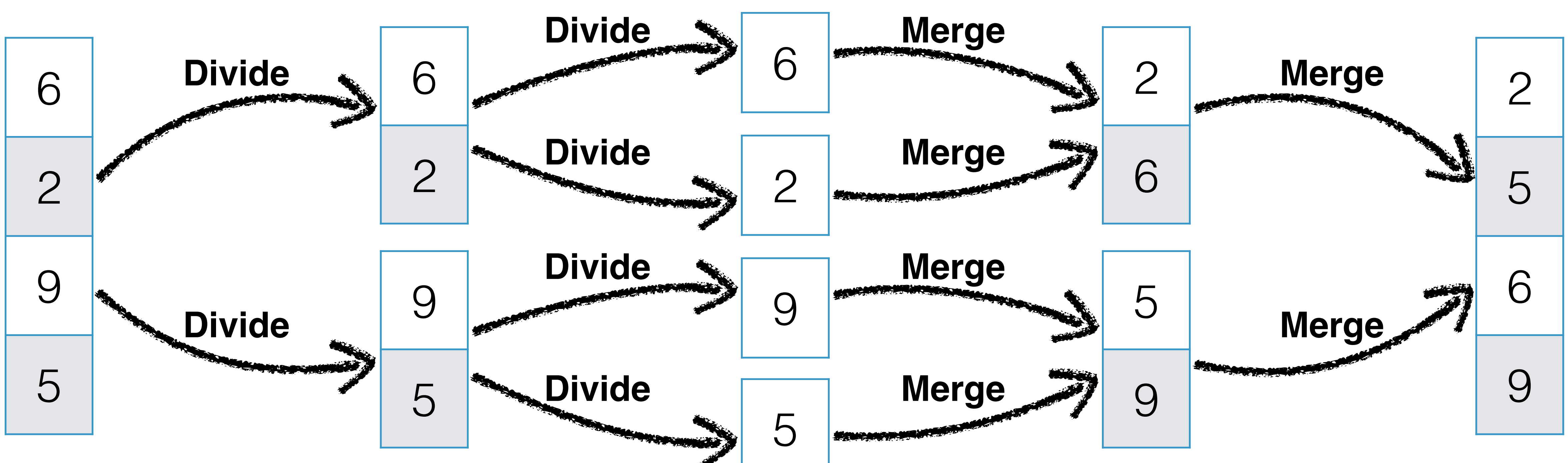
Merge sort

Idea del algoritmo:

1.- Si el arreglo tiene longitud menor o igual a1, está ordenado y por lo tanto se retorna sin hacer nada

2.- [Divide] En caso contrario, se parte el arreglo A[1..n] en A1[1..n/2] y A2[n/2+1..n] y se procede recursivamente sobre ellos

3.- [Merge] Se hace una intercalación ordenada de los elementos de A1[1..n/2] y A2[n/2+1..n] en A[1..n]



Merge sort

Algoritmo (en pseudo C)**:**

```
void ordenar (int A[n]){
    if (n < 2) then return;
    else {
        int A1[n/2], A2[n/2];
        A1 = A[1..n/2]; A2 = A[n/2+1..n];
        ordenar (A1); ordenar (A2);
        merge (A, A1, A2);
    }
}

void merge (int A[n], B[n/2], C[n/2]){
    int i = 0, j = 0, k = 0;
    while (i + j < n)
        if (B[i] <= C[j]) then {
            A[k] = B[i]; k++; i++;
        } else {
            A[k] = C[i]; k++; j++;
        }
}
```

Merge sort (complejidad)

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

$$f(n) = n$$

Caso 2
Teorema
Maestro

$$f(n) \in \Theta(n^c \log^k n)$$

$$c = \log_2 2$$

$$K = 0$$

$$T(n) \in \Theta(n \log n)$$

Heap sort

Idea del algoritmo:

1.- Convertimos el arreglo en un Heap tal usando el algoritmo Heapify de Floyd en O(n)

2.- Desencolamos los n elementos en O(n * log n) y los colocamos en un nuevo arreglo de n posiciones

Repaso

- Vimos algoritmos de ordenamiento (los clásicos de la literatura: Selection, Insertion, Bubble, Heap, Quick y Merge)

- Demostramos cuál es la cota inferior para el problema del ordenamiento para el caso en que no se tiene información adicional sobre los elementos

- Existen algoritmos con cotas en peor caso menores que $\Omega(n * \log n)$ pero requieren hipótesis sobre los elementos (p.e. bucket sort)

¡Es todo por hoy!