
uao

05

ESTRUCTURA DE DATOS Y ALGORITMOS 1

Algoritmos de Ordenamiento

Profesor
Orlando Arboleda Molina

uao

Algoritmos de Ordenamiento

- El **ordenamiento** (**sort**) consiste en **organizar** de forma **ascendente** o **descendente** los valores u objetos almacenados en **un arreglo** o colección, **según un criterio dado**.

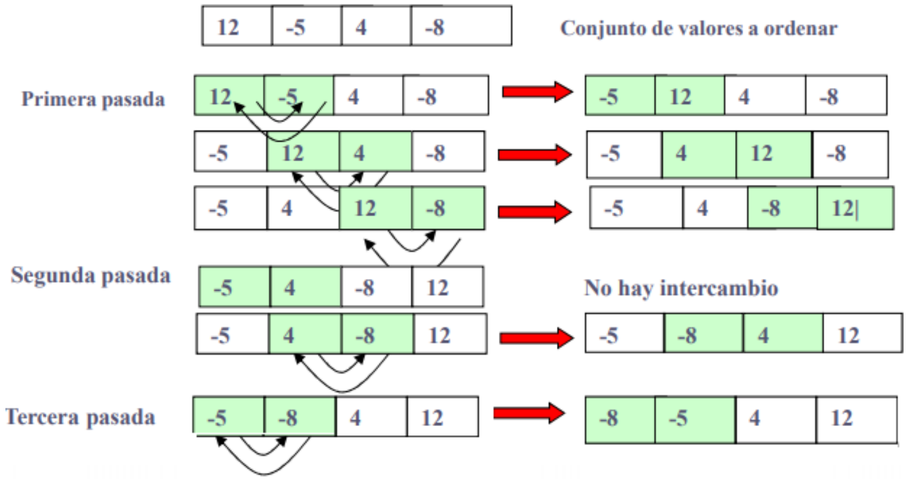
- Existen varios algoritmos de ordenamiento, cada uno con su correspondiente complejidad, entre ellos:
  - **Bubble Sort** (ordenamiento burbuja, que es impartido en los cursos iniciales de programación)
  - **Quick Sort**
  - **Merge Sort** (ordenamiento por mezcla)

- Los **lenguajes de programación** en su API **proporcionan métodos de ordenamiento**. En el caso de JavaScript y Java se proporciona el método ***sort***, al cual hay que **indicarle** el **criterio** por el cual se desea el ordenamiento y ajustar si el ordenamiento es ascendente o descendente.

Bubble Sort

- Ordena a partir de la **comparación** de **pares de elementos adyacentes** y los **intercambia si están en el orden incorrecto**. Este proceso **se repite** varias veces **hasta** que la **lista** está **completamente ordenada**.

Ejemplo:




```
function bubbleSort( arr, n)
{
var i, j;
for (i = 0; i < n-1; i++)
{
    for (j = 0; j < n-i-1; j++)
    {
        if (arr[j] > arr[j+1])
        {
        swap(arr,j,j+1);

        }
    }

}
}
```

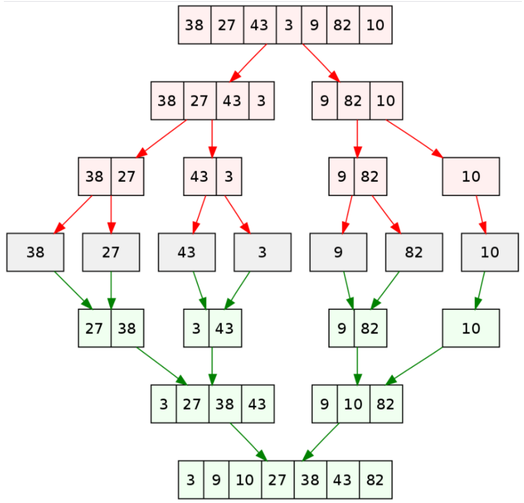
- Requiere ***n-1* pasadas** y en la **pasada *i*** realiza ***n-i* comparaciones** (aunque no deba intercambiar).
- El **numero de comparaciones**, para el **peor** de los **casos** (ordenado descendentemente):
  - $T(n) = (n-1)+(n-2)+\ldots\ldots 2+1 = O(n^2)$

Merge Sort

- **Divide** la **secuencia** original de ***n*** datos en **dos** de tamaño ***n/2***, los cuales **ordena**; y luego obtiene su respuesta de la **mezcla** de estas dos **secuencias ordenadas**.

- El tiempo de **ordenar 2 arreglos** de la ***n/2*** datos es **inferior** a **ordenar** los ***n*** datos. Además **mezclar** 2 **arreglos ordenados se puede hacer eficientemente**.

Ejemplo: se muestra funcionamiento del mergeSort en el arreglo dado. En la parte superior se indica la división, en la parte media el caso base y en la parte inferior la mezcla de las subarreglos ordenados.



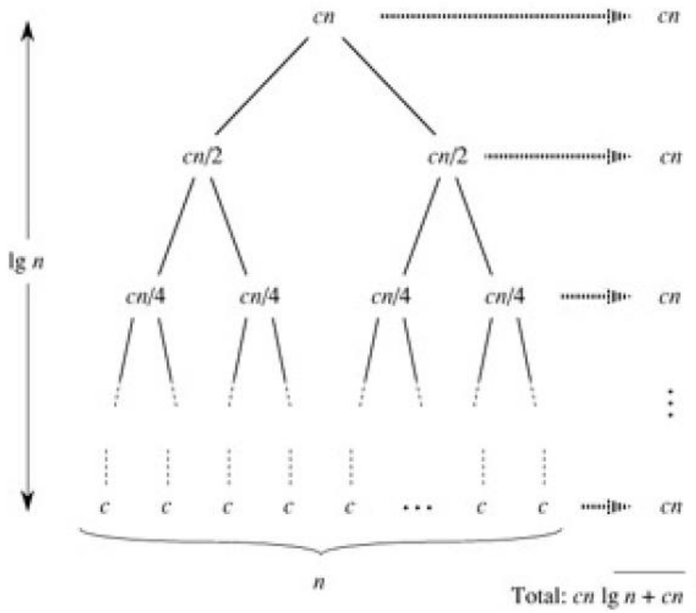

```
MERGE-SORT(A, p, r)
1 if p < r
2    then q ← ⌊(p + r)/2⌋
3         MERGE-SORT(A, p, q)
4         MERGE-SORT(A, q + 1, r)
5         MERGE(A, p, q, r)
```

Merge Sort

```
MERGE-SORT(A, p, r)
1 if p < r
2   then q ← ⌊(p + r)/2⌋
3        MERGE-SORT(A, p, q)
4        MERGE-SORT(A, q + 1, r)
5        MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
 1  n1 ← q - p + 1
 2  n2 ← r - q
 3  create arrays L[1 .. n1+1]  and R[1 ..n2+1]
 4  for i ← 1 to n1
 5       do L[i] ← A[p + i - 1]
 6  for j ← 1 to n2
 7       do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13       do if L[i] ≤ R[j]
14             then A[k] ← L[i]
15                  i ← i + 1
16             else A[k] ← R[j]
17                  j ← j + 1
```

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$



- **Análisis intuitivo**, teniendo en cuenta que dividir y mezclar es de orden Θ(n)

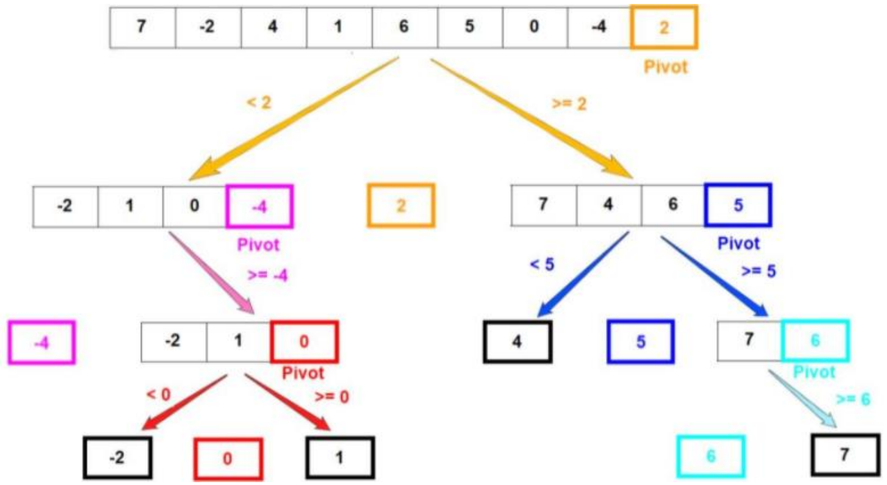
$$T(n) = cn \lg n + cn = \Theta(n \lg n) = O(n \lg n)$$

Quick Sort

- **Selecciona** un valor de la secuencia como **pivote** y lo **ubica** en **su posición final**. Además **reubica** todos los **valores menores** (o iguales) en la **secuencia izquierda** y todos los **valores mayores** en la **secuencia derecha**. Luego ordena estas dos secuencias.

- Generalmente el **pivote** es el **elemento** al **inicio** o **final** de la secuencia.

Ejemplo:




```
QUICKSORT(A, p, r)
1 if p < r
2    then q ← PARTITION(A, p, r)
3         QUICKSORT(A, p, q - 1)
4         QUICKSORT(A, q + 1, r)
```

Quick Sort

```
QUICKSORT(A, p, r)
1 if p < r
2    then q ← PARTITION(A, p, r)
3         QUICKSORT(A, p, q - 1)
4         QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p - 1
3  for j ← p to r - 1
4       do if A[j] ≤ x
5             then i ← i + 1
6                  exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

$$T(n) = \begin{cases} \Theta(1), & si \;\; p \geq r \\ T(n-1) + T(0) + \Theta(n), & si \; p < r \end{cases}$$

T(n) = T(n-1) + Θ(n)
T(n) = (T(n-2)+ Θ(n-1)) + Θ(n)
…….

T(n) = T(1)+ Θ(2) + …. + Θ(n-1) + Θ(n)
T(n) = $\Theta(n^2) = O(n^2)$

- **particionar** requiere **recorrer toda la secuencia**, luego es de orden Θ(n)
- **Análisis del peor caso:** cuando la **secuencia** de entrada está **ordenada**

Quick Sort

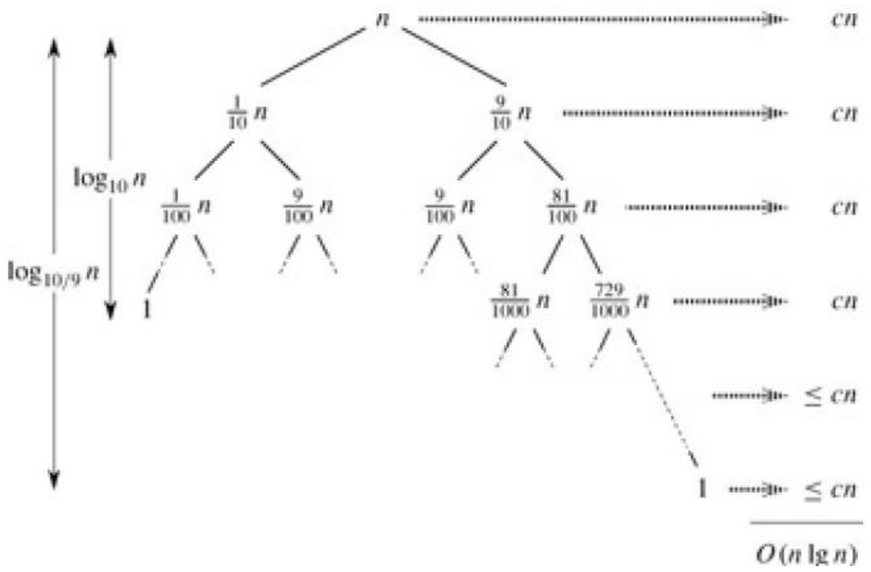
- **Análisis del mejor caso:** cuando se generan **2 secuencias** de tamaño **n/2**. Cual sería la relación de recurrencia respectiva y su complejidad ?

- **Análisis del caso promedio:** suponiendo que hace divisiones proporcionales

```
QUICKSORT(A, p, r)
1 if p < r
2    then q ← PARTITION(A, p, r)
3         QUICKSORT(A, p, q - 1)
4         QUICKSORT(A, q + 1, r)
```

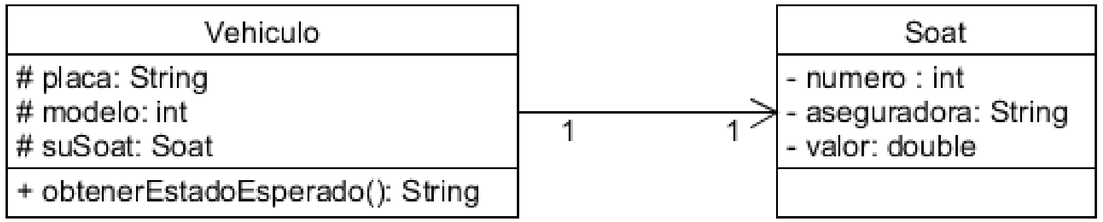
```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p - 1
3  for j ← p to r - 1
4       do if A[j] ≤ x
5             then i ← i + 1
6                  exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

$$T(n) = \begin{cases} \Theta(1), & si \quad p \geq r \\ T(an/b) + T\big((b-a)n/b\big) + \Theta(n), & si\ p < r \end{cases}$$



Ejercicio de Implementación

- Realizar la **practica** correspondiente a **Ordenamiento**, basada en vehículos, como se indica en el siguiente diagrama de clases. Para esta actividad, se debe disponer de:
  - El entorno del desarrollo **Visual Studio Code**.
  - Un **navegador moderno** y actualizado (ej. Chrome, Firefox, etc).



Se solicita:

1. Modificar la practica para que ordene ascendentemente según la placa
2. Realizar las adaptaciones necesarias para realizar el ordenamiento ascendente según el modelo.
3. Realizar las adaptaciones necesarias para realizar el ordenamiento descendente según el valor almacenado en el Soat.

Ordenamiento en JavaScript

- Se realiza con el método ***sort*** y como parámetro una **función de comparación** (si no se proporciona, los elementos se convierten a cadenas y se ordenan en orden lexicográfico).

- La **función de comparación** define cómo se deben comparar dos elementos (a y b):
  - Si retorna un número negativo (<0), a se coloca antes que b.
  - Si retorna 0, a y b se consideran iguales.
  - Si retorna un número positivo (>0), b se coloca antes que a.

Ejemplo: Dado un arreglo denominado ***lasPersonas***, de objetos Estudiantes que tienen los atributos código, nombre y edad, se suministran instrucciones para ordenar de forma ascendente por el código y por el nombre.

```
lasPersonas.sort(function(a,b){return a.codigo-b.codigo;})
```

```
lasPersonas.sort((a,b) => {return a.codigo-b.codigo;})
```

```
lasPersonas.sort(function(a,b){return a.nombre.localeCompare(b.nombre);})
```

```
lasPersonas.sort((a,b) => {return a.nombre.localeCompare(b.nombre);})
```

Ejercicio de Implementación

- En la **practica** que se realizó de **Ordenamiento**, realizar los ordenamientos solicitados previamente, usando el método ***sort***. Para esta actividad, se debe disponer de:
  - El entorno del desarrollo **Visual Studio Code**.
  - Un **navegador moderno** y actualizado (ej. Chrome, Firefox, etc).

(Extra) –Ordenamiento en Java

- En los **objetos preparados** (implementan la **Interface Comparable**) basta con invocar el método ***sort*** de la clase **Collections**.

Ejemplo:

```java
public class SortEmployee implements Comparable<SortEmployee> {
    private int cedula;
    private String name;

    // en este método se indica el atributo usado para el ordenamiento
    @Override
    public int compareTo(SortEmployee employee1) {
        return this.name.compareToIgnoreCase(employee1.name);
    }
```

```java
Collections.sort(lstSortEmployees);
```

- En los **objetos no preparados** (que no implementan la Interface Comparable) se puede realizar con la **interface Comparator**.

Ejemplo:

```java
public class Employee {
    private int cedula;
    private String name;
```

```java
Collections.sort(lstEmployees, new Comparator<Employee>(){
    @Override
    public int compare(Employee o1, Employee o2) {
        return o1.getName().compareToIgnoreCase(o2.getName());
    }
});
```

Ordenamiento en Java (usando Expresiones Lambda)

- Las **expresiones lambda** son **funciones anónimas**, es decir, funciones que **no necesitan una clase**.

- El **ordenamiento** con **expresiones lambda** está permitido desde Java 8, para objetos **no preparados**.

Ejemplo: A continuación se indican las instrucciones Java que permiten realizar el ordenamiento del ejemplo previo, utilizando expresiones lambda:

```
Collections.sort(lstEmployees,
        (x, y) -> x.getName().compareToIgnoreCase(y.getName()));
```

```
lstEmployees.sort(
        (x, y) -> x.getName().compareToIgnoreCase(y.getName()));
```

05

BUEN VIENTO Y BUENA MAR !!!


uao