



# Algoritmos de Ordenamiento

# Algoritmos de Ordenamiento

Estos algoritmos nos permiten ordenar datos, vectores o matrices.

Basados en operaciones de *comparación e intercambio*.

Una comparación consiste en tomar dos valores y determinar cual de los dos es mayor o menor.

Un intercambio consiste en tomar dos valores e intercambiar su posición.

Para cada algoritmo se miden la cantidad de comparaciones e intercambios realizados, el tiempo de ejecución, pre-requisitos etc.

A continuación se presentan los algoritmos de ordenamiento más populares.

# Burbuja

Este algoritmo compara 2 elementos consecutivos del arreglo, coloca el mayor en la posición derecha y el menor en la posición izquierda.

Repite este procedimiento por todo el arreglo para ubicar un dato en su posición.

Por lo tanto se repite *n veces* para ubicar los *n elementos*.

# Burbuja

15	12	65	45	1	2
----	----	----	----	---	---

**15-12-65-45-1-2**

**12-15-65-45-1-2**

Primera pasada **12-15-65-45-1-2**

**12-15-45-65-1-2**

**12-15-45-1-65-2**

**12-15-45-1-2-65**

**12-15-45-1-2**

**12-15-45-1-2**

Segunda pasada **12-15-45-1-2**

**12-15-1-45-2**

**12-15-1-2-45**

# Burbuja

15	12	65	45	1	2
----	----	----	----	---	---

**12-15-1-2**

**12-15-1-2**

Tercera pasada **12-1-15-2**

**12-1-2-15**

**12-1-2**

Cuarta pasada **1-12-2**

**1-2-12**

# Complejidad

- **Ventajas:**

- Fácil implementación.
- No requiere memoria adicional.

- **Desventajas:**

- Muy lento.
- Realiza numerosas comparaciones.
- Realiza numerosos intercambios.

- **Total comparaciones-intercambios**

$$n(n-1)/2$$

- **Complejidad:**

$$O(n^2)$$

## ALGORITMO

```
void burbuja (int array[ ], int tam) {  
    int i, j, temp;  
    for (i = tam-1; i > 0; i--) {  
        for (j = 0; j < i; j++) {  
            if (array[j] > array[j+1]) {  
                temp = array[j];  
                array[j] = array[j+1];  
                array[j+1] = temp;  
            }  
        }  
    }  
}
```

# Burbuja Mejorada

## ALGORITMO

```
void burbujaMejorada (int array[ ], int tam) {  
    int i = tam-1 , j , temp, bandera = 0;  
    while ( i > 0 && !bandera ){  
        bandera = 1 ;  
        for (j = 0;j < i;j++) {  
            if (array[j] > array[j+1]) {  
                temp = array[j];  
                array[j] = array[j+1];  
                array[j+1] = temp;  
                bandera = 0 ;  
            }  
        }  
        i-- ;  
    }  
}
```

# Inserción

El ordenamiento por inserción es el método que utilizamos para ordenar **un mazo de cartas** de baraja. Inicialmente se considera la primer carta como ordenada y en base a esta se ordenan todas las cartas de la derecha.

Cada que se elige una nueva carta se compara de derecha a izquierda con todas hasta encontrar una mayor. En este punto se *inserta* la nueva carta.

# Inserción

45	15	12	65	1	2
----	----	----	----	---	---

Primera pasada

**45-15-12-65-1-2**

**aux = 15**

**45-45-12-65-1-2**

**15-45-12-65-1-2**

Segunda pasada

**15-45-12-65-1-2**

**aux = 12**

**15-45-45-65-1-2**

**15-15-45-65-1-2**

**12-15-45-65-1-2**

# Inserción

45	15	12	65	1	2
----	----	----	----	---	---

Tercer pasada

12-15-45-65-1-2

**aux = 65**

Cuarta pasada

12-15-45-65 -1- 2

**aux = 1**

12-15-45-65-65-2

12-15-45-45-65-2

12-15-15-45-65-2

12-12-15-45-65-2

1 -12-15-45-65-2

# Inserción

45	15	12	65	1	2
----	----	----	----	---	---

Quinta pasada

**1-12-15-45-65 -2**

**aux = 2**

**1-12-15-45-65-65**

**1-12-15-45-45-65**

**1-12-15-15-45-65**

**1-12-12-15-45-65**

**1- 2 -12-15-45-65**

# Complejidad

- **Ventajas:**
  - Fácil implementación.
  - No requiere memoria adicional.
- **Desventajas:**
  - Muy lento.
  - Realiza numerosas comparaciones.
  - Realiza numerosos intercambios.
- **Comparaciones:**  $n^2/4$
- **Intercambios:**  $n^2/8$
- **Complejidad:**  
 $O(n^2)$

## ALGORITMO

```
void insercion(int array[], int tam) {  
    int i, j, aux;  
    for (i=1; i < tam; i++){  
        aux = array[i];  
        j = i-1;  
        while ((j >= 0) && (aux< array[j])) {  
            array[j+1] = array[j];  
            j --;  
        }  
        array[j+1] = aux;  
    }  
}
```

# Selección

**La idea básica de este algoritmo consiste en buscar el menor elemento del arreglo y colocarlo en la primera posición.**

**Luego se busca el segundo elemento mas pequeño del arreglo y lo colocamos en la segunda posición. El proceso continua hasta que todos los elementos del arreglo hayan sido ordenados.**

**La ventaja de éste algoritmo es que a pesar de hacer muchas comparaciones, solo hace un intercambio después de recorrer todo el arreglo.**

**Es decir solo en total hace máximo n intercambios**

# Selección

45	15	12	65	1	2
----	----	----	----	---	---

Primera pasada

**45-15-12-65- 1-2**

**45-15-12-65- 1-2**

**1-15-12-65-45-2**

Segunda pasada

**15-12-65-45-2**

**15-12-65-45-2**

**2-12-65-45-15**

# Selección

45	15	12	65	1	2
----	----	----	----	---	---

Tercer pasada

**12-65-45-15**

Cuarta pasada

**65-45-15**

**65-45-15**

**15-45-65**

# Complejidad

- Ventajas:
  - Fácil implementación.
  - No requiere memoria adicional.

- Desventajas:
  - Muy lento.
  - Numerosas comparaciones.
  - Pocos intercambios.

- Total comparaciones

$$C = (n-1) + (n-2) + \dots + 2 + 1 = \frac{n*(n+1)}{2}$$

$$C = \frac{n^2 - n}{2}$$

- Complejidad:  
 $O(n^2)$

## ALGORITMO

```
void seleccion(int array[], int tam) {  
    int menor , aux, i,j;  
    for(i=0 ; i<tam-1 ; i++) {  
        menor = i ;  
        for(j=i+1 ; j<tam ; j++) {  
            if(array[menor] > array [j])  
                menor=j;  
        }  
        aux = array[menor];  
        array [menor] = array [i];  
        array [i] = aux;  
    }  
}
```

# Quick Sort

El ordenamiento rápido es un algoritmo recursivo basado en la técnica de divide y vencerás.

Elige un elemento de la lista al que llama pivote.

Acomodar todos los elementos de la lista al lado del pivote, coloca a su izquierda todos los menores, y a su derecha otros mayores.

Entonces el pivote ya está en su posición (ordenado)

Ahora tenemos dos sublistas, una de menores y una de mayores. Repetir este proceso de forma recursiva para cada sublista.

# Rápido (Quick Sort)

4	7	8	2	6	9	1	3	10	5
---	---	---	---	---	---	---	---	----	---

pivote

4 - 7 - 8 - 2 - 6 - 9 - 1 - 3 - 10 - 5

4 - 7 - 8 - 2 - 6 - 9 - 1 - 3 - 10 - 5

4 - 5 - 3 - 2 - 1 - 9 - 6 - 8 - 10 - 7

4 - 5 - 3 - 2 - 1 - 6 - 9 - 8 - 10 - 7

pivote

4 - 5 - 3 - 2 - 1

4 - 5 - 3 - 2 - 1

1 - 2 - 3 - 5 - 4

pivote

9 - 8 - 10 - 7

7 - 8 - 10 - 9

7 - 8 - 10 - 9

pivote

1 - 2

pivote

5 - 4

pivote

7

pivote

10 - 9

1 - 2

- 3 -

4 - 5

- 6 -

7

- 8 -

9 - 10

# Complejidad

- **Ventajas:**
  - Rápido
  - No requiere memoria adicional.
- **Desventajas:**
  - Implementación poco complicada
  - Recursividad (utiliza muchos recursos)
  - Diferencia entre el mejor y peor caso
- **Total comparaciones**
$$C = n + n + n + \dots + n = kn$$
donde  $k = \log_2(n)$
- **Complejidad:**  
Mejor caso:  $O(n \log_2 n)$   
Peor caso:  $O(n^2)$

```
void quickSort ( int array[], int primero, int ultimo ){
    int pos, i , j , pivote , temp ;
    i = primero;
    j = ultimo;
    pivote = array [ (primero + ultimo) / 2];
    do {
        while ( array [i] < pivote ) {
            i++;
        }
        while ( array [j] > pivote ) {
            j--;
        }
        if ( i <= j ) {
            temp = array [i]
            array [j] = array [i]
            array [ i ] = temp ;
            i++; j--;
        }
    } while(i <= j );
    if(primero < j) {
        quickSort (array , primero,j);
    }
    if(ultimo > i) {
        quickSort (array , i,ultimo);
    }
}
```

# Ejercicio

- Ordenar el siguiente arreglo con: *Burbuja, Inserción y Selección*

8	4	9	7	6	2	1
---	---	---	---	---	---	---

- Ordenar el siguiente arreglo con *QuickSort*

8	13	6	16	2	11	9	5	1	7	15	4	10
---	----	---	----	---	----	---	---	---	---	----	---	----