



# Algoritmos de Ordenación.

---



## Ordenación

---

- La ordenación de elementos según un orden ascendente o descendente influye notablemente en la velocidad y simplicidad de los algoritmos que los manipulan posteriormente.
- En general, *un conjunto de elementos se almacenan en forma ordenada con el fin de simplificar la recuperación de información manualmente*, o facilitar el acceso mecanizado a los datos de una manera más eficiente.
- Los métodos de ordenación se suelen dividir en:
  - **ordenamiento interno**, si los elementos que han de ser ordenados están en la Memoria Principal.
  - **ordenamiento externo**, si los elementos que han de ser ordenados están en un dispositivo de almacenamiento auxiliar.



## Complejidad algoritmos de ordenación

---

- La complejidad de cualquier algoritmo estima el **tiempo de ejecución** como una **función del número de elementos a ser ordenados**.
- Cada algoritmo estará compuesto de las siguientes operaciones:
  - COMPARACIONES que prueban si  $A_i < A_j$  ó  $A_i < B$  (donde B es una variable auxiliar)
  - INTERCAMBIOS: permutar los contenidos de  $A_i$  y  $A_j$  ó  $A_i$  y  $B$
  - ASIGNACIONES de la forma  $B \leftarrow A_i$ ,  $A_j \leftarrow B$  ó  $A_j \leftarrow A_i$
- Generalmente, la función de complejidad solo computa COMPARACIONES porque el número de las otras operaciones es como mucho constante del número de comparaciones.



## Ordenación por inserción

---

- También conocido como **método de la baraja**.
- Consiste en **tomar elemento a elemento e ir insertando cada elemento en su posición correcta** de manera que se mantiene el orden de los elementos ya ordenados.
- Es el método habitual usado por los jugadores de cartas para ordenar: tomar carta por carta manteniendo la ordenación.

## Ordenación por Inserción

- Inicialmente se toma el primer elemento, a continuación se toma el segundo y se inserta en la posición adecuada para que ambos estén ordenados, se toma el tercero y se vuelve a insertar en la posición adecuada para que los tres estén ordenados, y así sucesivamente.
  1. Suponemos el primer elemento ordenado.
  2. Desde el segundo hasta el último elemento, hacer:
    1. suponer ordenados los **(i - 1)** primeros elementos
    2. tomar el elemento **i**
    3. buscar su posición correcta
    4. insertar dicho elemento, obteniendo **i** elementos ordenados

## Código Ordenación por Inserción

```
public void OrdInsercion()
{
    for (int i=1; i < A.length; i++) // Supone el primer elemento ordenado
    {
        int elem = A[i]; // Elemento a ordenar
        int j = (i-1); // Posición a comparar

        /*Si el elemento a comparar es mayor que el elemento a ordenar
        entonces desplazo el elemento a comparar una posición a la derecha
        para insertar el elemento a ordenar en la posición correcta*/
        while ((j >= 0) && (elem < A[j]))
            A[j+1] = A[j--]; /*Desplazo el elemento una posición a la
            derecha y disminuyo en una unidad la Posición a
            comparar*/
        // Se inserta el elemento a ordenar en su posición correcta
        A[j+1] = elem;
    }
}
```

## Complejidad Ordenación por Inserción

- **CASO MEJOR:** Cuando el array está ordenado. Entonces sólo se hace una comparación en cada paso.
- Ej. 15 20 45 60  $n=4$
- En general, para  $n$  elementos se hacen  $(n-1)$  comparaciones. Por tanto, complejidad  $O(n)$ .

| Pasada | Nº de Intercambios | Nº Comparaciones |
|--------|--------------------|------------------|
| 1      | 0                  | 1                |
| 2      | 0                  | 1                |
| 3      | 0                  | 1                |

## Complejidad Ordenación por Inserción

- **CASO PEOR:** Cuando el array está ordenado inversamente.
- Ej. 86 52 45 20  $n=4$ 
  - 52 86 45 20
  - 45 52 86 20
  - 20 45 52 86
- En general, para  $n$  elementos se realizan  $(n-1)$  intercambios y  $(n-1)$  comparaciones. Por tanto,  $O(n^2)$ .

| Pasada | Nº de Intercambios | Nº Comparaciones |
|--------|--------------------|------------------|
| 1      | 1                  | 1                |
| 2      | 2                  | 2                |
| 3      | 3                  | 3                |

## Complejidad Ordenación por Inserción

- **CASO MEDIO:** Los elementos aparecen de forma aleatoria.
- Se puede calcular como la suma de las comparaciones mínimas y máximas dividida entre dos:  
$$((n-1) + n(n-1)/2)/2 = (n^2 + (n-2))/4$$
, por tanto complejida  $O(n^2)$ .

## Inserción Binaria.

```
public void OrdInsercionBin()
{
    for (int i=1; i < A.length; i++)
    {
        int elem = A[i];
        int bajo = 0;
        int alto = (i-1);

        while (bajo <= alto)//Se busca la posición donde se debe almacenar el
            elemento a ordenar
        {
            int medio = (alto + bajo)/2;
            if (elem < A[medio]) alto = medio -1;
            else bajo = medio + 1;
        }
        for (int j = (i-1); j >= bajo; j--)//Se desplazan todos los elementos
            mayores que el elemento a ordenar una posición a la derecha
            A[j+1] = A[j];
        A[bajo] = elem;
    }
}
```



## Complejidad Inserción Binaria

---

- Con la búsqueda binaria se reduce el número de comparaciones desde  $O(n^2)$  hasta un  $O(n \log n)$ . Sin embargo, el número de sustituciones requiere un tiempo de ejecución de  $O(n^2)$ . Por lo tanto, el orden de complejidad no cambia, además la ordenación por inserción se usa normalmente sólo cuando  $n$  es pequeño, y en este caso la búsqueda lineal es igual de eficiente que la búsqueda binaria.



## Ordenación por Selección.

---


- Este método se basa en que **cada vez que se mueve un elemento, se lleva a su posición correcta**. Se comienza examinando todos los elementos, se localiza el más pequeño y se sitúa en la primera posición. A continuación, se localiza el menor de los restantes y se sitúa en la segunda posición. Se procede de manera similar sucesivamente hasta que quedan dos elementos. Entonces se localiza el menor y se sitúa en la penúltima posición y el último elemento, que será el mayor de todos, ya queda automáticamente colocado en su posición correcta.
- Para  $i$  desde la primera posición hasta la penúltima  
    localizar menor desde  $i$  hasta el final  
    intercambiar ambos elementos

## Código Ordenación por Selección

```
public void OrdSeleccion()
{
    //Para todos los elementos desde el primero hasta el penultimo
    for (int i=0; i < (A.length-1); i++)
    {
        int menor = i;
        /*Se localiza el elemento menor desde la posición
        desde la cual se está ordenando hasta el último elemento*/
        for (int j=(i+1); j < A.length; j++)
            if (A[j] < A[menor]) menor = j;

        /*Si el elemento a ordenar es distinto del de la
        posición que se está ordenando se intercambian los elementos*/
        if (menor != i)
            intercambiar(i,menor); //intercambia los elementos
        de esas posiciones
    }
}
```

## Complejidad Ordenación por Selección

- El tiempo de ejecución de dicho algoritmo viene determinado por el **número de comparaciones**, las cuales son independientes del orden original de los elementos, el tiempo de ejecución es  $O(n^2)$ .
- $f(n) = (n-1) + (n-2) + \dots + 2 + 1 = n(n-1)/2$    
 $O(n^2)$ .
- El número de intercambios es  $O(n)$ .
- Por tanto complejidad  **$O(n^2)$** .

## Desventajas Ordenación por Selección.

- Muy lenta con arrays grandes.
- No detecta si el array está todo ni parcialmente ordenado.
- EJERCICIO: Dado un array de 100 componentes. Realiza un programa que clasifique simultáneamente en orden creciente los componentes pares y en orden decreciente los impares utilizando ordenación por Inserción

## Solución.

```
public void OrdInserParImpar ()
{
    for (int i = 2; i < A.length; i++) /* Variable que recorre el array secuencialmente
                                         desde el primer elemento a ordenar (n/2), hasta el
                                         ultimo elemento. Supone el primer elemento ordenado*/
    {
        int elem=A[i];
        int j = i - 2; // Posicion del elemento a comparar
        if (i % 2 == 0){
            while ((j > -1) && (elem<A[j]))
            {
                A[j+2]=A[j];
                j -= 2;
            }
        }
        else while ((j > 0) && (elem>A[j]))
        {
            A[j+2]=A[j];
            j -= 2;
        }
        A[j+2]=elem;
    }
}
```





## Ordenación por Intercambio o Burbuja

---

- Se basa en el principio de **comparar e intercambiar pares de elementos adyacentes hasta que todos estén ordenados.**
- Desde el primer elemento hasta el penúltimo no ordenado
  - comparar cada elemento con su sucesor
  - intercambiar si no están en orden



## Código Ordenación por Burbuja

---

```
public void OrdBurbuja(){  
    for (int pasada=0; pasada < A.length-1; pasada++)  
        for (int j=0; j < (A.length-pasada-1); j++)  
            if (A[j] > A[j+1]) intercambiar(j,j+1);  
}
```

Si tenemos en cuenta que tras una pasada puede suceder que ya estén todos los elementos ordenados, en este caso no sería necesario seguir realizando comparaciones.

## Mejora Ordenación Burbuja

```
public void OrdBurbuja2()
{
    boolean noOrdenados = true;
    int pasada = 0;
    while ((noOrdenados) && (pasada < A.length - 1))
    {
        noOrdenados = false;
        for (int j=0; j < (A.length-pasada-1); j++)
            if (A[j] > A[j+1])
            {
                intercambiar(j,j+1);
                noOrdenados = true;
            }
        pasada ++;
    }
}
```

## Complejidad Ordenación Burbuja

- El tiempo de ejecución de dicho algoritmo viene determinado por el número de comparaciones, en el peor de los casos  $O(n^2)$ .
- COMPARACIONES:  $(n-1) + (n-2) + \dots + 3 + 2 + 1 = n(n-1)/2 \Rightarrow O(n^2)$
- INTERCAMBIOS:  $(n-1) + (n-2) + \dots + 3 + 2 + 1 = n(n-1)/2 \Rightarrow O(n^2)$



## Ventajas y Desventajas Ordenación Burbuja

---

- Su principal ventaja es la simplicidad del algoritmo.
- El problema de este algoritmo es que solo compara los elementos adyacentes del array. Si el algoritmo comparase primero elementos separados por un amplio intervalo y después se centrare progresivamente en intervalos más pequeños, el proceso sería más eficaz. Esto llevo al desarrollo de ordenación Shell y QuickSort.



## Ejercicio Ordenación Burbuja

---

1. Escribir un método que utilizando el método de la burbuja, alternativamente realice una pasada de izquierda a derecha (coloca el mayor elemento) y a continuación otra de derecha a izquierda (coloca el elemento menor), recortándose los elementos a tratar por ambos lados del array. Demuestra con un ejemplo su funcionamiento. ¿Cuál es la complejidad de este método?

## Solución

```
public void ordenar()
{
    for (int pasada = 0; pasada < A.length; pasada+=2)
    {
        for (int j= pasada; j < (A.length -pasada-1); j++)
            if (A[j] > A[j+1] )
            {
                int temp = A[j];
                A[j] = A [j+1];
                A[j+1] = temp;
            }
        for (int i= A.length - pasada - 2; i > pasada; i--)
            if (A[i] < A[i-1] )
            {
                int temp = A[i];
                A[i] = A [i-1];
                A[i-1] = temp;
            }
    }
}
```

Complejidad  $O(n^2)$

## Ordenación Shell

- Es una mejora de la ordenación por inserción (colocar cada elemento en su posición correcta, moviendo todos los elementos mayores que él, una posición a la derecha), que se utiliza cuando el número de datos a ordenar es grande.
- Para ordenar una secuencia de elementos se procede así: **se selecciona una distancia inicial y se ordenan todos los elementos de acuerdo a esa distancia**, es decir, cada elemento separado de otro a *distancia* estará ordenado con respecto a él. Se disminuye esa distancia progresivamente, hasta que se tenga distancia 1 y todos los elementos estén ordenados.

## Ejemplo Ordenación Shell

| Posición    | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  |
|-------------|---|----|----|----|----|----|----|----|----|
| Original    | 4 | 14 | 21 | 32 | 18 | 17 | 26 | 40 | 6  |
| Distancia 4 | 4 | 14 | 21 | 32 | 18 | 17 | 26 | 40 | 6  |
| Distancia 2 | 4 | 14 | 21 | 32 | 6  | 17 | 26 | 40 | 18 |
| Distancia 1 | 4 | 14 | 6  | 17 | 18 | 32 | 21 | 40 | 26 |
| Final       | 4 | 6  | 14 | 17 | 18 | 21 | 26 | 32 | 40 |

Ordenación  
por  
Insercción

## Código Ordenación Shell

```
public void OrdShell ()
{
    int intervalo = A.length / 2; /* distancia, los elementos estará
                                   ordenados a una distancia igual a intervalo.*/
    while (intervalo > 0) //Se ordenan hasta que la distancia sea igual a 1
    {
        for (int i = intervalo; i < A.length; i++)
        {
            int elem=A[i];
            int j = i - intervalo; // Posicion del elemento a comparar
            while ((j > -1) && (elem<A[j]))
            {
                A[j+intervalo]=A[j];
                j -= intervalo;
            }
            A[j+intervalo]=elem;
        }
        intervalo = intervalo / 2;
    }
}
```



## Complejidad de Shell

---

- No es fácil de calcular.  $O(n^{3/2})$ . Si se divide la distancia por 2.2 optiene una complejidad de  $O(n^{5/4})$ .
- El tiempo de ejecución depende de la secuencia de incrementos que se elige.
- El rendimiento de la ordenación Shell es bastante aceptable en la práctica, aún para  $n$  grande.
- Su simplicidad lo hace adecuado para clasificar entradas moderadamente grandes.



## Ordenación por mezcla (mergesort)

---

- Para ordenar una secuencia de elementos  $S$ , un procedimiento podría consistir en dividir  $S$  en dos subsecuencias disjuntas, ordenarlas de forma independiente y unir los resultados de manera que se genere la secuencia final ordenada. Dentro de esta filosofía es posible hacer una distinción de los algoritmos en dos categorías:
  - Algoritmos de fácil división y difícil unión, el *MÉRGESORT*.
  - Algoritmos de difícil división y fácil unión, el *QUICKSORT*.

## Ordenación por mezcla (mergesort)

- El **MERGESORT** consiste en :
  - Dividir los elementos en dos secuencias de la misma longitud aproximadamente.
  - **Ordenar** de forma independiente cada subsecuencia.
  - **Mezclar** las dos secuencias ordenadas para producir la secuencia final ordenada.

## Código MergeSort

```
public void mergeSort (int [] A, int bajo, int alto){  
    if (bajo < alto) //Si hay más de un elemento  
    {  
        int medio = (alto + bajo)/2;  
        mergeSort (A, bajo, medio);  
        mergeSort (A, medio+1, alto);  
        //Procedimiento que mezcla el resultado de  
        las dos llamadas anteriores  
        merge (A, bajo, medio+1, alto);  
    }  
}
```

## Código Mezcla de MergeSort.

- El proceso de **mezcla** es el siguiente:
  - Repetir mientras haya elementos en una de las dos secuencias:
    - Seleccionar el menor de los elementos de las subsecuencias y añadirlo a la secuencia final ordenada.
    - Eliminar el elemento seleccionado de la secuencia a la que pertenece.
    - Copiar en la secuencia final los elementos de la subsecuencia en la que aún quedan elementos.

## Código Mezcla de MergeSort.

```
public void merge (int [] A, int bajo, int bajo_2, int alto)
{
    int i = bajo; //Variable de primer elemento de la primera subsecuencia
    int finbajo = bajo_2 - 1; //Variable del último elemento de la primera subsecuencia
    int j = bajo_2; //Variable del primer elemento de la segunda subsecuencia
    int k = bajo;
    /* Temp es un array ya definido*/
    while ((i <= finbajo) && (j <= alto))
    {
        if (A[i] <= A[j])
            Temp[k++] = A[i++];
        else
            Temp[k++] = A[j++];
    }

    while (i <= finbajo) //Si se agotaron todos los elementos de la segunda subsecuencia
        Temp[k++] = A[i++];

    while (j <= alto) //Si se agotaron los de la primera subsecuencia
        Temp[k++] = A[j++];
    //Paso todos los elementos del Temporal al array
    for (i = bajo; i <= alto; i++)
        A[i] = Temp[i];
}
```



## Complejidad Mezcla de MergeSort.

- Teniendo en cuenta que la entrada consiste en el total de elementos  $n$  y que cada comparación asigna un elemento a Temp. El número de comparaciones es  $(n-1)$  y el número de asignaciones es  $n$ . Por lo tanto, el algoritmo de mezcla se ejecuta en un tiempo lineal  **$O(n)$** .

## Complejidad MergeSort

- El análisis de eficiencia de la ordenación por mezcla da lugar a una ecuación recurrente para el tiempo de ejecución.
- Suponemos  $n$  potencia de 2,  $n = 2^k$
- Para  $N = 1$ , Tiempo constante.
- Para  $N > 1$ , El tiempo de ordenación para  $n$  números es igual al tiempo para 2 ordenaciones recursivas de tamaño  $n/2$  + el tiempo para mezclar (que es lineal).
- Por tanto,  $T(n) = 2 T(n/2) + n$

$\log_2 n = k$

Tiempo de Mezclar

Tiempo de Ordenación

## Complejidad MergeSort

- Para resolver la ecuación se divide por  $n$ .
- $T(n) / n = (2 (T(n/2)) / n) + (n/n)$
- $T(n) / n = T(n/2) / (n/2) + 1$
- Esta ecuación es válida para cualquier potencia de 2, así que se puede escribir:
- $T(n/2) / (n/2) = (T(n/4) / (n/4)) + 1$
- $T(n/4) / (n/4) = (T(n/8) / (n/8)) + 1$
- Así sucesivamente,  
hasta  $T(2) = (T(1) / 1) + 1$

## Complejidad MergeSort

- Se suman todas las ecuaciones anteriores, como todos los términos se anulan, se obtiene el siguiente resultado:
  - $T(n) / n = (T(1) / 1) + k$ , siendo  $K$  el número de ecuaciones que tenemos, es decir, el número de divisiones a la mitad que realizamos, por tanto  $k = \log_2 n$
  - Para resolver la ecuación se dividió entre  $n$ , por tanto ahora multiplicamos por  $n$ .
  - $T(n) = n + n \log n$ . Por tanto la complejidad del algoritmo MergeSort es de  **$O(n \log n)$**



## Ordenación Rápida (QuickSort)

---

- En la ordenación rápida, **la secuencia inicial de elementos se divide en dos subsecuencias de diferente tamaño**. La obtención de las dos subsecuencias es el proceso que acarrea más tiempo mientras que la combinación de las subsecuencias ordenadas para obtener la secuencia final consume muy poco tiempo.
- Para dividir en dos la secuencia de elementos, se selecciona un elemento sobre el cual efectuar la división, el **PIVOTE**. Se dividen los elementos en dos grupos, los elementos menores que el pivote y aquellos mayores o igual al pivote.



## Ordenación QuickSort.

---

- La elección del elemento Pivote se puede seleccionar de diferentes formas:
  - El mayor de los dos primeros elementos distintos encontrados.
  - El primer elemento.
  - El último elemento.
  - El elemento medio.
  - Un elemento aleatorio.
  - Mediana de tres elementos (El primer elemento, el elemento del medio y el último elemento).

## Pasos a seguir QuickSort.

- El método de ordenación rápida se basa en **ordenar los elementos comprendidos entre  $A_i$  y  $A_j$  conforme a las siguientes cinco etapas:**
  1. Si desde  $A_i$  a  $A_j$  hay al menos dos elementos distintos entonces comenzar la aplicación del algoritmo.
  2. **Seleccionar el PIVOTE** como el elemento mayor de los dos primeros elementos distintos encontrados.
  3. **Insertar PIVOTE en la última posición.**
  4. **Permutar los elementos** desde  $A_i$  hasta  $A_j$  de modo que, para algún  $i \leq k \leq j$  :  
 $A_i, \dots, A_{k-1} < \text{PIVOTE}$   
 $A_k, \dots, A_j \geq \text{PIVOTE}$   
Es decir, en las  $(k-1)$  primeras posiciones queden los elementos menores que pivote, mientras que en la posición  $k$  hacia delante queden los elementos mayores o iguales que el pivote.
  5. Invocar a:      QUICKSORT desde  $i$  hasta  $(k - 1)$   
                     QUICKSORT desde  $k$  hasta  $j$

## Paso 2: Elección del Pivote

- Para la elección del pivote se puede utilizar la siguiente función que localiza el elemento mayor de los dos primeros elementos distintos existentes entre el  $i$  y el  $j$ .

```
int buscaPivote (int i, int j)
{
    int primer = A[i];
    int k = i + 1;

    while (k <= j)
    {
        if (A[k] > primer)
            return k;
        else if (A[k] < primer)
            return i;
        else k++;
    }
    //Si llega al final del array y todos los elementos son iguales, o si sólo hay un elemento
    return -1;
}
```

## Paso 4: Permutación de elementos

- Para el paso 4 de permutación de los elementos se utilizan dos cursores:
  - D : para mover a la derecha mientras el elemento sea menor que el pivote.
  - I : para mover a la izquierda mientras el elemento sea mayor o igual que el pivote
- de acuerdo a tres fases :
  - CAMBIO : Si  $D < I$  se intercambian  $A_D$  y  $A_I$ , con lo cual probablemente:  
 $A_D < \text{PIVOTE}$  y  
 $A_I \geq \text{PIVOTE}$
  - EXPLORACIÓN: Mover  
D hacia la derecha sobre cualquier elemento MENOR que el pivote y  
I hacia la izquierda sobre cualquier elemento MAYOR o IGUAL que el pivote.
  - COMPROBACIÓN: Si  $D > I$  hemos acabado con éxito la reordenación.

## Código, paso 4.

```
int particion (int i, int j, int pivote)
{
    int derecha = i;
    int izquierda = j-1;
    while (derecha <= izquierda)
    {
        intercambiar(derecha, izquierda);
        while (A[derecha] < pivote)
            derecha++;
        while (A[izquierda] >= pivote)
            izquierda--;
    }
    return derecha;
}
```

## Código QuickSort

---

```
void quickSort (int [] A, int i, int j)
{
    int indicePivote = buscaPivote(i, j);
    if (indicePivote != -1)
    {
        int pivote = A[indicePivote];
        intercambiar(indicePivote,j);
        int k = particion(i,j,pivote);
        quickSort(A, i, k-1);
        quickSort(A, k,j);
    }
}
```

## Complejidad QuickSort.

---

- **Caso Mejor:** Cuando el pivote, divide al conjunto en dos subconjuntos de igual tamaño. En este caso hay dos llamadas con un tamaño de la mitad de los elementos, y una sobrecarga adicional lineal, igual que en MergeSort. En consecuencia el tiempo de ejecución es  **$O(n \log n)$** .

## Complejidad QuickSort.

- **Caso Peor:** Se podría esperar que los subconjuntos de tamaño muy distinto proporcionen resultados malos.
- Supongamos que en cada paso de recursión sólo hay un elemento menor a pivote. En tal caso el subconjunto I (elementos menores que pivote) será uno y el subconjunto D (elementos mayores o igual a pivote) serán todos los elementos menos uno. El tiempo de ordenar 1 elemento es sólo 1 unidad, pero cuando  $n > 1$ .
- $T(N) = T(N-1) + N$
- $T(N-1) = T(N-2) + (N-1)$
- $T(N-2) = T(N-3) + (N-2) \dots$
- $T(2) = T(1) + 2$
- $T(N) = T(1) + 2 + 3 + 4 + \dots + N = N(N+1)/2$ ,  $O(n^2)$ .

## Complejidad QuickSort.

- **Caso Medio:** Complejidad  $O(n \log n)$ .



## Ejercicio Algoritmos de Ordenación

---

- Implementa el método de Ordenación por Inserción, pero colocando los elementos mayores. Puedes suponer que todos los elementos que están a la derecha del elemento que quieres ordenar están bien ordenados entre sí (es decir, de menor a mayor). Al final el array queda ordenado crecientemente.