**Algoritmos de ordenación**

### Como ordenar arrays en Java. Método Arrays.sort()

## ARRAYS.SORT() - COLLECTIONS.REVERSEORDER()

Para ordenar arrays de cualquier tipo Java dispone del método sort de la clase Arrays.

Para utilizarlo es necesario incluir el import:

import java.util.Arrays;

Por ejemplo, dado el siguiente array de Strings:

String [] nombres = {"juan", "pedro", "ana", "maria", "felipe", "luis", "eduardo"};

para ordenarlo de forma ascendente escribiremos la instrucción:

Arrays.sort(nombres);

Si mostramos el array por pantalla, comprobaremos que está ordenado de forma ascendente:

for(String s : nombres)

System.out.println(s);

Arrays.sort ordena de forma ascendente (de menor a mayor). Para ordenar un array de forma descendente (de mayor a menor) hay que indicarlo utilizando el método reverseOrder() de la clase Collections.

Para utilizar reverseOrder es necesario incluir el import:

import java.util.Collections;

Por ejemplo, para ordenar el array nombres de forma descendente escribimos la instrucción Arrays.sort de la siguiente forma:

Arrays.sort(nombres, Collections.reverseOrder());

También tenemos la opción de ordenar solo una parte del array, indicando la posición del elemento inicial y la del elemento final (que no se incluye en la ordenación).

Por ejemplo, para ordenar solo los elementos 1, 2 y 3 ("pedro", "ana", "maria") del array nombres escribimos la instrucción de esta forma:

Arrays.sort(nombres, 1, 4);

El 1 indica la posición del elemento donde comienza la ordenación y el 4 indica la posición del primer elemento que no entra en la ordenación.

El contenido del array después de esta ordenación es el siguiente:

juan

ana

maria

pedro

felipe

luis

eduardo

vemos que solo se han ordenado los elementos 1, 2 y 3. El resto queda igual:

También podemos ordenar solo una parte del array en orden inverso. Por ejemplo, para ordenar solo los elementos 1, 2 y 3 en orden inverso:

Arrays.sort(nombres, 1,4, Collections.reverseOrder());

El contenido del array es ahora:

juan

pedro

maria

ana

felipe

luis

eduardo

Con Arrays.sort podemos ordenar arrays de cualquier tipo de datos. Por ejemplo, para ordenar un array de enteros:

int [] numeros = {3, 5, 1, 2, 1, 7, 0, -1};

Arrays.sort(numeros);

//mostrarlo ordenado

for (int n : numeros) {

System.out.println(n);

}

Collections.reverOrder() solo funciona para arrays de objetos. Por este motivo si queremos ordenar de forma descendente arrays de tipos de datos simples debemos utilizar la clase envolvente equivalente al tipo de dato básico.

Por ejemplo, para ordenar un array de enteros de forma descendente hay que declararlo de tipo Integer en lugar de int.

Integer [] numeros = {3, 5, 1, 2, 1, 7, 0, -1};

Arrays.sort(numeros, Collections.reverseOrder());

for (int n : numeros) {

System.out.println(n);

}

### Algoritmos de ordenación. Método de la Burbuja

## ALGORITMO DE LA BURBUJA

El **algoritmo de la burbuja** es uno de los métodos de ordenación más conocidos y uno de los primeros que aprenden los programadores.  
  
Consiste en comparar pares de elementos adyacentes en un array y si están desordenados intercambiarlos hasta que estén todos ordenados.

Si A es el array a ordenar, se realizan A.length-1 pasadas. Si la variable i es la que cuenta el número de pasadas, en cada pasada i se comprueban los elementos adyacentes desde el primero hasta A.length-i-1 ya que el resto hasta el final del array están ya ordenados. Si los elementos adyacentes están desordenados se intercambian.

El método de ordenación de la burbuja en java para ordenar un array A es el siguiente:

public static void burbuja(int[] A) {

int i, j, aux;

for (i = 0; i < A.length - 1; i++) {

for (j = 0; j < A.length - i - 1; j++) {

if (A[j + 1] < A[j]) {

aux = A[j + 1];

A[j + 1] = A[j];

A[j] = aux;

}

}

}

}

Es uno de los peores algoritmos de ordenación en cuanto a tiempo de ejecución, solamente es recomendable su uso para ordenar listas con un número pequeño de elementos.

**Algoritmos de ordenación en Java. Método de Ordenación por Selección.**

El método de ordenación por selección consiste en repetir los siguientes pasos:

Se busca el elemento más pequeño del array y se coloca en la primera posición.

Entre los restantes, se busca el elemento más pequeño y se coloca en la segunda posición.

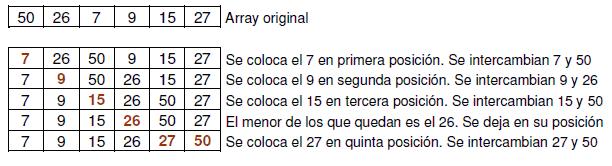
Entre los restantes se busca el elemento más pequeño y se coloca en la tercera posición.

……..

Este proceso se repite hasta colocar el último elemento.

De forma gráfica el proceso sería el siguiente:

Array original a ordenar: [50, 26, 7, 9, 15, 27]



El método de ordenación por selección en java para ordenar un array de enteros A es el siguiente:

//método java de ordenación por selección

public static void seleccion(int A[]) {

int i, j, menor, pos, tmp;

for (i = 0; i < A.length - 1; i++) { // tomamos como menor el primero

menor = A[i]; // de los elementos que quedan por ordenar

pos = i; // y guardamos su posición

for (j = i + 1; j < A.length; j++){ // buscamos en el resto

if (A[j] < menor) { // del array algún elemento

menor = A[j]; // menor que el actual

pos = j;

}

}

if (pos != i){ // si hay alguno menor se intercambia

tmp = A[i];

A[i] = A[pos];

A[pos] = tmp;

}

}

}

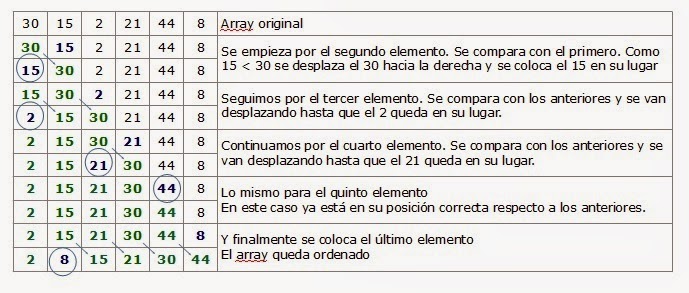
El tiempo de ejecución del algoritmo de ordenación por selección es del orden O(n2)

**Ordenamiento por Inserción directa**

El método de ordenación por inserción directa consiste en recorrer todo el array comenzando desde el segundo elemento hasta el final. Para cada elemento, se trata de colocarlo en el lugar correcto entre todos los elementos anteriores a él o sea entre los elementos a su izquierda en el array.

Dada una posición actual p, el algoritmo se basa en que los elementos A[0], A[1], ..., A[p-1] ya están ordenados.

De forma gráfica el proceso que sigue el método de inserción directa es el siguiente:



 El método de Ordenamiento por inserción directa en Java es el siguiente:

public static void insercionDirecta(int A[]){

int p, j;

int aux;

for (p = 1; p < A.length; p++){ // desde el segundo elemento hasta

aux = A[p]; // el final, guardamos el elemento y

j = p - 1; // empezamos a comprobar con el anterior

while ((j >= 0) && (aux < A[j])){ // mientras queden posiciones y el

// valor de aux sea menor que los

A[j + 1] = A[j]; // de la izquierda, se desplaza a

j--; // la derecha

}

A[j + 1] = aux; // colocamos aux en su sitio

}

}

**En el peor de los casos, el tiempo de ejecución en O(n2).**

**En el mejor caso** (cuando el array ya estaba ordenado), el tiempo de ejecución de este método de ordenamiento es **O(n).**

El caso medio dependerá de cómo están inicialmente distribuidos los elementos. Cuanto más ordenada esté inicialmente más se acerca a O(n) y cuanto más desordenada, más se acerca a O(n2).

El peor caso el método de inserción directa es igual que en los métodos de burbuja y selección, pero el mejor caso podemos tener ahorros en tiempo de ejecución.

**Java Quicksort**

El método de ordenación Quicksort fue desarrollado por Hoare en el año 1960.

Es el algoritmo de ordenación más rápido.

Se basa en la técnica **divide y vencerás**, que consiste en ir subdividiendo el array en arrays más pequeños, y ordenar éstos. Para hacer esta división, se toma un valor del array como **pivote**, y se mueven todos los elementos menores que este pivote a su izquierda, y los mayores a su derecha. A continuación, se aplica el mismo método a cada una de las dos partes en las que queda dividido el array.

Después de elegir el pivote se realizan dos búsquedas:

Una de izquierda a derecha, buscando un elemento mayor que el pivote

Otra de derecha a izquierda, buscando un elemento menor que el pivote.

Cuando se han encontrado los dos elementos anteriores, se intercambian, y se sigue realizando la búsqueda hasta que las dos búsquedas se encuentran.

La implementación del método de ordenación Quicksort es claramente recursiva.

Suponiendo que tomamos como pivote el primer elemento, el método Java Quicksort que implementa este algoritmo de ordenación para ordenar un array de enteros se presenta a continuación. Los parámetros izq y der son el primer y último elemento del array a tratar en cada momento.

El método ordena un array A d enteros desde la posición *izq* hasta la posición *der.* En la primera llamada recibirá los valores izq = 0, der = ELEMENTOS-1.

public static void quicksort(int A[], int izq, int der) {

int pivote=A[izq]; // tomamos primer elemento como pivote

int i=izq; // i realiza la búsqueda de izquierda a derecha

int j=der; // j realiza la búsqueda de derecha a izquierda

int aux;

while(i < j){ // mientras no se crucen las búsquedas

while(A[i] <= pivote && i < j) i++; // busca elemento mayor que pivote

while(A[j] > pivote) j--; // busca elemento menor que pivote

if (i < j) { // si no se han cruzado

aux= A[i]; // los intercambia

A[i]=A[j];

A[j]=aux;

}

}

A[izq]=A[j]; // se coloca el pivote en su lugar de forma que tendremos

A[j]=pivote; // los menores a su izquierda y los mayores a su derecha

if(izq < j-1)

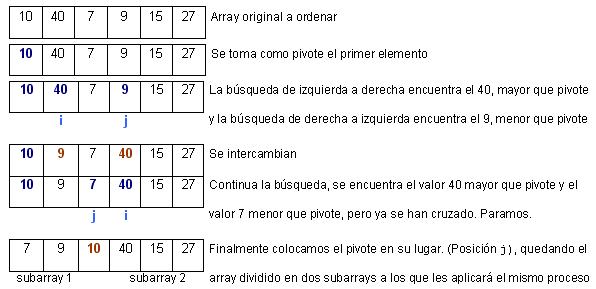
quicksort(A,izq,j-1); // ordenamos subarray izquierdo

if(j+1 < der)

quicksort(A,j+1,der); // ordenamos subarray derecho

}

De forma gráfica el proceso sería el siguiente:



La elección del pivote determinará la eficiencia de este algoritmo ya que determina la partición del array. Si consideramos que el array está desordenado, podemos elegir el primer elemento y el algoritmo funcionaría de forma eficiente. Pero si el array está casi ordenado, elegir el primer elemento como pivote sería una mala solución ya que obtendríamos un subarray muy pequeño y otro muy grande. Por la misma razón, elegir el último elemento del array como pivote también es una mala idea. Pretendemos conseguir que el tamaño de los subarrays sea lo más parecido posible.

Una alternativa a elegir el primer elemento es elegir como pivote un elemento al azar de entre todos los del array.

Otra estrategia es calcular la mediana de los valores de la izquierda, centro y derecha del vector.

Por ejemplo, para el vector: 9 8 1 6 10 2 3, se calcula la mediana de los elementos que ocupan el primer lugar, el último y el centro o sea 9 3 6. La mediana es 6 que determinaría las particiones {1 3 2} {6} {8 10 9}.

**En el peor caso, cuando el pivote es el elemento menor del array el tiempo de ejecución del método Quicksort es O(n2).**

**En general el tiempo medio de ejecución del Quicksort es O(n log n).**

**Método Shell de Ordenación**

El método de ordenación Shell debe su nombre a su inventor, Donald Shell, y fue uno de los primeros algoritmos de ordenamiento en romper la barrera del tiempo cuadrático.

Es una mejora del método de inserción directa, utilizado cuando el array tiene un gran número de elementos.

Cualquier algoritmo de ordenación que intercambia elementos adyacentes (como los algoritmos burbuja, selección o inserción) tiene un tiempo promedio de ejecución de orden cuadrático (n2). El método Shell mejora este tiempo comparando cada elemento con el que está a un cierto número de posiciones llamado salto, en lugar de compararlo con el que está justo a su lado. Este salto es constante, y su valor inicial es N/2 (siendo N el número de elementos, y siendo división entera).

Se van dando pasadas con el mismo salto hasta que en una pasada no se intercambie ningún elemento de sitio. Entonces el salto se reduce a la mitad, y se vuelven a dar pasadas hasta que no se intercambie ningún elemento, y así sucesivamente hasta que el salto vale 1.

El método Shell de ordenación en Java para ordenar un array A de enteros es el siguiente:

public static void shell(int A[]) {

int salto, aux, i;

boolean cambios;

for (salto = A.length / 2; salto != 0; salto /= 2) {

cambios = true;

while (cambios) { // Mientras se intercambie algún elemento

cambios = false;

for (i = salto; i < A.length; i++) // se da una pasada

{

if (A[i - salto] > A[i]) { // y si están desordenados

aux = A[i]; // se reordenan

A[i] = A[i - salto];

A[i - salto] = aux;

cambios = true; // y se marca como cambio.

}

}

}

}

}

Ejemplo de ejecución:



Con sólo 6 intercambios se ha ordenado el array, cuando por inserción se necesitaban muchos más. El rendimiento del método Shell de ordenación es bastante aceptable, aún para el caso de un número de elementos muy grande. Se ha comprobado que el **tiempo de ejecución promedio es de O(n2/3)** para la mayoría de las secuencias de salto.

**Método de ordenación MergeSort**

El método MergeSort es un algoritmo de ordenación recursivo con un número de comparaciones entre elementos del array mínimo.

Su funcionamiento es similar al Quicksort, y está basado en la técnica divide y vencerás.

De forma resumida el funcionamiento del método MergeSort es el siguiente:

- Si la longitud del array es menor o igual a 1 entonces ya está ordenado.

- El array a ordenar se divide en dos mitades de tamaño similar.

- Cada mitad se ordena de forma recursiva aplicando el método MergeSort.

- A continuación, las dos mitades ya ordenadas se mezclan formando una secuencia ordenada.

La ordenación mergesort se puede implementar en Java de la siguiente forma:

public static void mergesort(int A[],int izq, int der){

if (izq < der){

int m=(izq+der)/2;

mergesort(A,izq, m);

mergesort(A,m+1, der);

merge(A,izq, m, der);

}

}

El método ordena un array A de enteros desde la posición *izq* hasta la posición *der.* En la primera llamada al método recibirá los valores izq = 0, der = ELEMENTOS-1.

Primero se calcula el elemento central *m*. A continuación la primera parte del array, desde izq hasta m y la segunda parte del array, desde m+1 hasta der, se mezclan mediante llamadas recursivas al método mergesort.

La recursión termina cuando izq == der, es decir, cuando un subarray contiene solamente un elemento.

La operación principal de mezcla la realiza el método *merge.*Este método se puede implementar de varias formas, la más usual es la siguiente:

public static void merge(int A[],int izq, int m, int der){

int i, j, k;

int [] B = new int[A.length]; //array auxiliar

for (i=izq; i<=der; i++) //copia ambas mitades en el array auxiliar

B[i]=A[i];

i=izq; j=m+1; k=izq;

while (i<=m && j<=der) //copia el siguiente elemento más grande

if (B[i]<=B[j])

A[k++]=B[i++];

else

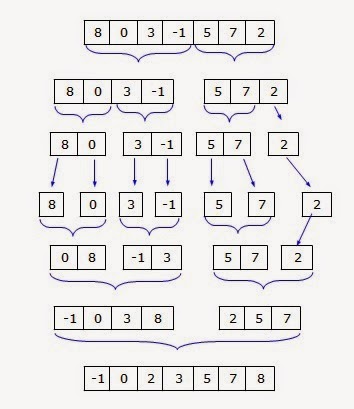
A[k++]=B[j++];

while (i<=m) //copia los elementos que quedan de la

A[k++]=B[i++]; //primera mitad (si los hay)

}

De forma gráfica podemos representar el funcionamiento del algoritmo de la siguiente forma:

[](https://4.bp.blogspot.com/-G9vbYIZoTwY/VEeVj_EPeYI/AAAAAAAAAaM/ev9tZdrnyTU/s1600/java-mergesort.jpg)

**El tiempo de ejecución promedio del método MergeSort es O(n log n).**

\*\*Enlace que nos muestra el funcionamiento de estos métodos de ordenación.

<https://www.toptal.com/developers/sorting-algorithms>

Ejercicios

1. Implementar una aplicación que registre 10 números y que muestre posteriormente los números impares ordenados en forma ascendente. Utilizar métodos de ordenamiento.
2. Insertar nombres de equipos de futbol con su respectiva puntuación y visualizarlo ordenados por la puntación utilizando métodos quicksort y burbuja.