# Métodos de Ordenación Elementales

En el contexto de los métodos de ordenación, cada elemento de dato tiene su clave, y los métodos de ordenación trabajan ordenando los elementos de dato según sus claves. Por lo regular, los métodos comparan las claves e intercambian los elementos de dato. En lugar de desplazar físicamente los elementos de datos, con frecuencia, sólo se intercambian índices, punteros o referencias. Esto se denomina ordenación indirecta.

Un método de ordenación que trabaja sobre un conjunto de datos que se encuentra en memoria (e.g., un arreglo, una lista) se dice que es un método de ordenación interna. Por el contrario, si el conjunto de datos almacenados en archivos no pueden ser cargado en memoria (por ejemplo, por razones de tamaño) y el método de ordenación opera sobre los archivos, se dice que es de ordenación externa. Evidentemente, en la ordenación interna se accede a los elementos de dato más fácilmente, mientras que en la ordenación externa se accede a los elementos de dato de forma secuencial o al menos en grades bloques.

Los métodos de ordenación se pueden clasificar de acuerdo a sus requerimientos de memoria. Los métodos in situ son aquellos que requieren ninguna o muy poca memoria extra. En el otro extremo, existen métodos que requieren mucha memoria extra.

Una característica que puede ser importante es la estabilidad del método de ordenación. Un algoritmo de ordenación es estable si elementos de dato con la misma clave conservan su orden relativo luego de su aplicación. Típicamente, los métodos elementales son estables mientras que la mayoría de los algoritmos sofisticados no lo son.

Entre los métodos de ordenación elementales están Selección e Inserción, los cuales son descritos a continuación.

Típicamente, los métodos de ordenación elementales tienen peor desempeño que los sofisticados, pero existen muchas aplicaciones en las que es mejor utilizar un método de ordenación elemental. Por ejemplo, cuando el algoritmo se utiliza pocas veces y/o se ordenan pocos elementos.

Como regla general se tiene que los métodos elementales necesitan cerca de N2 pasos para ordenar N elementos organizados al azar.

En general, no se recomienda su uso para ordenar:

archivos grandes

archivos clasificados aleatoriamente.

Por su parte, métodos más avanzados pueden lograr desempeños de orden: N( log N) , N(3/2), N. Más aún, se puede demostrar que ningún método de ordenación puede utilizar menos de N( log N) comparaciones entre claves cuando éstas están organizadas al azar.

**MERGE SORT**

Este algoritmo también es llamado de intercambio o combinación, debido que combina (intercala) dos estructuras previamente ordenadas. Este algoritmo consiste en:

Dividir el grupo de datos en dos y ordenar por separado cada mitad.

Cuando se tengan las mitades ordenadas, pueden irse mezclando para obtener fácilmente una secuencia ordenada.

El algoritmo MergeSort Utiliza los siguientes tres pasos:

DIVIDIR: divide la secuencia de "n" elementos a ordenar en dos subsecuencias de "n/2" elementos cada una.

VENCER: ordena las dos subsecuencias de manera recursiva mediante el algoritmo MERGESORT.

COMBINAR: combina las dos subsecuencias ordenadas para generar la solución.

De ahí su comparación con el paradigma algorítmico: "Divide y Vencerás".

Este algoritmo fue desarrollado por el matemático húngaro John Von Neumann en 1945.

[Thomas Cormen, 2001].

**Codigo:**

public int[] mergeSort(int[] vec) {

return mergeSort( vec, 0, vec.length-1, new int[vec.length]);

}

public int[] mergeSort(int[] vec, int izquierda, int derecha, int[] aux) {

if (derecha > izquierda) {

int mitad = (derecha + izquierda) / 2;

vec = mergeSort(vec, izquierda, mitad, aux);

vec = mergeSort(vec, mitad + 1, derecha, aux);

merge(vec, izquierda, mitad+1, derecha, aux);

}

return vec;

}

public void merge(int vec[], int izquierda, int medio, int derecha,int[] aux) {

int izqEnd = medio - 1;

int tmpPos = izquierda;

int numElements = derecha - izquierda + 1;

while (izquierda <= izqEnd && medio <= derecha)

if (this.vec[izquierda] <(this.vec[medio]))

aux [tmpPos++] = this.vec[izquierda++];

else

aux[tmpPos++] = this.vec[medio++];

while (izquierda <= izqEnd)// Copy rest of first half

aux[tmpPos++] = this.vec[izquierda++];

while (medio <= derecha)// Copy rest of der half

aux[tmpPos++] = this.vec[medio++];

// Copio aux a vec.

for (int i = 0; i < numElements; i++, derecha--)

this.vec[derecha] = aux[derecha];

}

### HEAP SORT

El ordenamiento por montículos (Heap sort en inglés) es un algoritmo de ordenación no recursivo, no estable, con complejidad computacional O(n log n).

Este algoritmo consiste en almacenar todos los elementos del vector a ordenar en un montículo (heap), y luego extraer el nodo que queda como nodo raíz del montículo (cima) en sucesivas iteraciones obteniendo el conjunto ordenado. Basa su funcionamiento en una propiedad de los montículos, por la cual, la cima contiene siempre el menor elemento (o el mayor, según se haya definido el montículo) de todos los almacenados en él.

### QUICK SORT

Esta es probablemente la técnica más rápida conocida. Fue desarrollada por C.A.R. Hoare en 1960. El algoritmo original es recursivo, pero se utilizan versiones iterativas para mejorar su rendimiento (los algoritmos recursivos son en general más lentos que los iterativos, y consumen más recursos).

Entre sus ventajas están:

* Trabaja in situ.
* Emplea N(log N) operacionales en promedio para ordenar N elementos.
* Su bucle interno extremadamente corto.

Y sus desventajas:

* Es recursivo y su implementación no recursiva es complicada.
* El desempeño es N2en su peor caso.

El algoritmo fundamental es el siguiente:

\* Eliges un elemento de la lista. Puede ser cualquiera (en Optimizando veremos una forma más efectiva). Lo llamaremos elemento de división.

\* Buscas la posición que le corresponde en la lista ordenada (explicado más abajo).

\* Acomodas los elementos de la lista a cada lado del elemento de división, de manera que a un lado queden todos los menores que él y al otro los mayores (explicado más abajo también). En este momento el elemento de división separa la lista en dos sublistas (de ahí su nombre).

\* Realizas esto de forma recursiva para cada sublista mientras éstas tengan un largo mayor que 1. Una vez terminado este proceso todos los elementos estarán ordenados.

Una idea preliminar para ubicar el elemento de división en su posición final sería contar la cantidad de elementos menores y colocarlo un lugar más arriba. Pero luego habría que mover todos estos elementos a la izquierda del elemento, para que se cumpla la condición y pueda aplicarse la recursividad. Reflexionando un poco más se obtiene un procedimiento mucho más efectivo. Se utilizan dos índices: i, al que llamaremos contador por la izquierda, y j, al que llamaremos contador por la derecha. El algoritmo es éste:

\* Recorres la lista simultáneamente con i y j: por la izquierda con i (desde el primer elemento), y por la derecha con j (desde el último elemento).

\* Cuando lista[i] sea mayor que el elemento de división y lista[j] sea menor los intercambias.

\* Repites esto hasta que se crucen los índices.

\* El punto en que se cruzan los índices es la posición adecuada para colocar el elemento de división, porque sabemos que a un lado los elementos son todos menores y al otro son todos mayores (o habrían sido intercambiados).

**Codigo:**

public void quicksort(Vector vec, int primero, int ultimo) {

int i = primero, j = ultimo;

double pivote = this.vec[(primero + ultimo) / 2];

double auxiliar;

do {

while (this.vec[i] < pivote)

i++;

while (this.vec[j] > pivote)

j--;

if (i <= j) {

auxiliar = this.vec[j];

this.vec[j] = this.vec[i];

this.vec[i] = auxiliar;

i++;

j--;

}

} while (i <= j);

if (primero < j)

quicksort(vec, primero, j);

if (ultimo > i)

quicksort(vec, i, ultimo);

}

### SHELL SORT

Debe su nombre al ingeniero y matemático estadounidense Donald Shell , que lo publicó en la revista Communications of the ACM en 1959.

Es un algoritmo de ordenación interna muy sencillo pero muy ingenioso, basado en comparaciones e intercambios, y con unos resultados radicalmente mejores que los que se pueden obtener con el método de la burbuja, el de selección directa o el de inserción directa.

CARACTERISTICAS

-Se trata de un algoritmo de ordenación interna; los datos están en memoria principal. Podría utilizarse para ordenación externa (en memoria secundaria) siempre y cuando se disponga de acceso aleatorio, pero el algoritmo no fue ideado para eso.

-Se basa en comparaciones e intercambios.

-Necesita que el tiempo de acceso a cualquier dato sea constante (es decir, fue ideado para trabajar con arrays, de referencias o punteros, etc...). Ojo a otras estructuras, como listas enlazadas, etc... ya que en ese caso, el tiempo de acceso a un elemento no es constante, depende de la posición del elemento.

-El estudio de su complejidad no es trivial, sino todo lo contrario. La implementación original de Shell tiene una complejidad en el peor caso de O(n2), aunque en un caso promedio o en casos típicos comprobados empíricamente, los resultados son mucho mejores que con la burbuja, selección directa o inserción directa, cuya complejidad en el peor caso también es del orden de O(n2).

-En cierto modo, puede considerarse una ampliación del algoritmo de inserción directa, con lo cual, conviene tenerlo claro antes de meterse con el de Shell.

-No es un algoritmo en-línea.

**Código:**

public void shell() {

for (int increment = this.vec.length / 2; increment > 0;

increment = (increment == 2 ? 1: (int) Math.round(increment / 2.2))) {

for (int i = increment; i < this.vec.length; i++) {

for (int j = i; j >= increment

&& this.vec[j - increment] > this.vec[j];

j -= increment) {

double temp = this.vec[j];

this.vec[j] = this.vec[j - increment];

this.vec[j - increment] = temp;

}

}

}

}

### BUBBLE SORT

El Ordenamiento de Burbuja (Bubble Sort en inglés) es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada. Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas". También es conocido como el método del intercambio directo.

Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo el más sencillo de implementar.

**Como funciona**

El algoritmo bubble sort tiene dos bucles for internos que recorren el vector comparando el elemento j-esimo-1 con el elemento con el j-esimo elemento y en caso de que este sea mayor hace un cambio de los elementos. Al tener dos bucles internos el comportamiento es en general O(n2), y en las mejores condiciones se comporta como O(n).

Consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados. Con el array anterior, {40,21,4,9,10,35}:

Primera pasada:

{21,40,4,9,10,35} <-- Se cambia el 21 por el 40.

{21,4,40,9,10,35} <-- Se cambia el 40 por el 4.

{21,4,9,40,10,35} <-- Se cambia el 9 por el 40.

{21,4,9,10,40,35} <-- Se cambia el 40 por el 10.

{21,4,9,10,35,40} <-- Se cambia el 35 por el 40.

Segunda pasada:

{4,21,9,10,35,40} <-- Se cambia el 21 por el 4.

{4,9,21,10,35,40} <-- Se cambia el 9 por el 21.

{4,9,10,21,35,40} <-- Se cambia el 21 por el 10.

Ya están ordenados, pero para comprobarlo habría que acabar esta segunda comprobación y hacer una tercera.

**Codigo:**

public void burbuja() {

double temp;

for (int i = 1; i < this.vec.length; i++) {

for (int k = this.vec.length - 1; k >= i; k--) {

if (this.vec[k] < this.vec[k - 1]) {

temp = this.vec[k];

this.vec[k] = this.vec[k - 1];

this.vec[k - 1] = temp;

}

}

}

}

### INSERCION Y SELECCION DIRECTA

**INSERCION DIRECTA**

El algoritmo de ordenación por el método de inserción directa es un algoritmo relativamente sencillo y se comporta razonablemente bien en gran cantidad de situaciones.

Como funciona:

Se basa en intentar construir una lista ordenada en el interior del array a ordenar.

El insertion sort trabaja insertando el item en su lugar correspondiente al final de la lista. Como el bubblesort, éste algoritmo se comporta como O(n2), pero a pesar de tener el misma complejidad, este algoritmo es casi el doble más eficiente que el bubblesort.

En este método lo que se hace es tener una sublista ordenada de elementos del array e ir insertando el resto en el lugar adecuado para que la sublista no pierda el orden. La sublista ordenada se va haciendo cada vez mayor, de modo que al final la lista entera queda ordenada.

Para el ejemplo {40,21,4,9,10,35}, se tiene:

{40,21,4,9,10,35} <-- La primera sublista ordenada es {40}.

Insertamos el 21:

{40,40,4,9,10,35} <-- aux=21;

{21,40,4,9,10,35} <-- Ahora la sublista ordenada es {21,40}.

Insertamos el 4:

{21,40,40,9,10,35} <-- aux=4;

{21,21,40,9,10,35} <-- aux=4;

{4,21,40,9,10,35} <-- Ahora la sublista ordenada es {4,21,40}.

Insertamos el 9:

{4,21,40,40,10,35} <-- aux=9;

{4,21,21,40,10,35} <-- aux=9;

{4,9,21,40,10,35} <-- Ahora la sublista ordenada es {4,9,21,40}.

Insertamos el 10:

{4,9,21,40,40,35} <-- aux=10;

{4,9,21,21,40,35} <-- aux=10;

{4,9,10,21,40,35} <-- Ahora la sublista ordenada es {4,9,10,21,40}.

Y por último insertamos el 35:

{4,9,10,21,40,40} <-- aux=35;

{4,9,10,21,35,40} <-- El array está ordenado.

En el peor de los casos, el número de comparaciones que hay que realizar es de N\*(N+1)/2-1, lo que nos deja un tiempo de ejecución en O(n2). En el mejor caso (cuando la lista ya estaba ordenada), el número de comparaciones es N-2. Todas ellas son falsas, con lo que no se produce ningún intercambio. El tiempo de ejecución está en O(n).

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

El peor caso es igual que en los métodos de burbuja y selección, pero el mejor caso es lineal, algo que no ocurría en éstos, con lo que para ciertas entradas podemos tener ahorros en tiempo de ejecución.

La ordenación por inserción utiliza aproximadamente N2/4 comparaciones y N2/8 intercambios en el caso medio y dos veces más en el peor caso(N2/2 comparaciones y N2/4 intercambios). Nótese que cuando el archivo está ordenado la ordenación por inserción es lineal (O(N)).

SELECCIÓNDIRECTA

El algoritmo de ordenación por el método de selección directa es un algoritmo relativamente sencillo y uno de los más fáciles de recordar e implementar.

Se basa en realizar varias pasadas, intentando encontrar en cada una de ellas el elemento que según el criterio de ordenación es mínimo y colocándolo posteriormente en su sitio.

A efectos prácticos, no suele dar resultados buenos si se compara con otros métodos de ordenación. Realiza una enorme cantidad de comparaciones, pero en contrapartida, muy pocos intercambios. Eso hace que su utilización se restrinja en general a dos situaciones: o bien necesitamos un algoritmo sencillito para ordenar unos pocos datos y cogemos éste mismo que no está mal y es fácil de recordar, o bien tenemos una situación en la cual escribir en el array es mucho más gravoso que leer, como puede ser un escenario en el que intervengan determinados dispositivos de almacenamiento o memorias tipo flash, eeprom, etc. para el soporte de los datos.

Este algoritmo se basa en hacer comparaciones, así que para que realice su trabajo de ordenación son imprescindibles dos cosas: un array o estructura similar de elementos comparables y un criterio claro de comparación, tal que dados dos elementos nos diga si están en orden o no.

**Características de Rendimiento de los Métodos de Ordenación Elementales (INSERCION Y SELECCION DIRECTA)**

* La simple inspección de las implementaciones anteriores sirve de evidencia de que los métodos elementales son cuadráticos (tanto en el peor caso como en el caso medio) y no necesitan memoria extra (in situ).
* La ordenación por selección utiliza aproximadamente N2/2 comparaciones y N intercambios. El número de comparaciones está dado por: (N-1)+ (N-2)+ (N-3)+…+2+1 = [N(N-1 )]/2.   
  Si N🡪 ∞ entonces el número de comparaciones es igual a N2/2.   
  **Por otra parte, puede haber, máximo N-1 intercambios. Estos resultados son independientes del conjuntos de datos de entrada, entonces se dice que el método de selección es insensible a los datos de entrada.**
* Se puede demostrar que cualquier algoritmo de ordenación que intercambie elementos adyacentes tiene un tiempo de ejecución promedio Ω(N2) (cota inferior).

**Codigo:**

public void selecccion() {

for (int i = 0; i < this.vec.length - 1; i++) {

int min = i;

for (int j = i + 1; j < this.vec.length; j++) {

if (this.vec[j] < this.vec[min]) {

min = j;

}

}

if (i != min) {

double aux = this.vec[i];

this.vec[i] = this.vec[min];

this.vec[min] = aux;

}

}

}

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Listado de algoritmos de ordenamiento**  Algunos algoritmos de ordenamiento agrupados según estabilidad tomando en cuenta la complejidad computacional.   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Estables** | | | | | | **Nombre traducido** | **Nombre original** | **Complejidad** | **Memoria** | **Método** | | **Ordenamiento de burbuja** | Bubblesort | O(n²) | O(1) | Intercambio | | **Ordenamiento de burbuja bidireccional** | Cocktail sort | O(n²) | O(1) | Intercambio | | **Ordenamiento por inserción** | Insertion sort | O(n²) | O(1) | Inserción | | **Ordenamiento por casilleros** | Bucket sort | O(n) | O(n) | No comp.. | | **Ordenamiento por cuentas** | Counting sort | O(n+k) | O(n+k) | No comp. | | **Ordenamiento por mezcla** | Merge sort | O(n log n) | O(n) | Mezcla | | **Ordenamiento con árbol binario** | Binary tree sort | O(n log n) | O(n) | Inserción | |  | Pigeonhole sort | O(n+k) | O(k) |  | | **Ordenamiento Radix** | Radix sort | O(nk) | O(n) | No comp. | |  | Stupid sort | O(n³) versión recursiva | O(n²) |  | |  | Gnome sort | O(n²) |  |  | | **Inestables** | | | | | | **Nombre traducido** | **Nombre original** | **Complejidad** | **Memoria** | **Método** | | **Ordenamiento Shell** | Shell sort | O(n1.25) | O(1) | Inserción | |  | Comb sort | O(n log n) | O(1) | Intercambio | | **Ordenamiento por selección** | Selection sort | O(n²) | O(1) | Selección | | **Ordenamiento por montículos** | Heapsort | O(n log n) | O(1) | Selección | |  | Smoothsort | O(n log n) | O(1) | Selección | | **Ordenamiento rápido** | Quicksort | Promedio: O(n log n),  peor caso: O(n²) | O(log n) | Partición | |  | Several Unique Sort | Promedio: O(n u),  peor caso: O(n²); u=n;  u = número único de registros |  |  | |

//Clase Vector

**public** **class** Vector {

**public** **double** vec[];

**public** **int** dim;

**public** Vector(**int** dim) {

**this**.dim = dim;

vec = **new** **double**[dim];

**for** (**int** i = 0; i < dim; i++)

vec[i] = 0;

}

**public** Vector(**double**[] vec) {

**this**.vec = vec;

dim = vec.length;

}

**public** **void** cargaAscendente() {

**for** (**int** i = 0; i < **this**.dim; i++)

**this**.vec[i] = i;

}

**public** **void** cargaAleatorea() {

**for** (**int** i = 0; i < **this**.dim; i++)

**this**.vec[i] = i \* ((**double**) Math.*random*() \* 100);

}

**public** **void** cargaDescendente() {

**for** (**int** i = **this**.dim; i > 0; i--)

**this**.vec[**this**.dim - i] = i;

}

**public** **void** muestra() {

System.*out*.println("\n");

**for** (**int** i = 0; i < **this**.dim; i++)

System.*out*.printf(**this**.vec[i] + " ");

}

// Ordenamiento por Burbuja

**public** **void** burbuja() {

**double** temp;

**for** (**int** i = 1; i < **this**.vec.length; i++)

**for** (**int** k = **this**.vec.length - 1; k >= i; k--)

**if** (**this**.vec[k] < **this**.vec[k - 1]) {

temp = **this**.vec[k];

**this**.vec[k] = **this**.vec[k - 1];

**this**.vec[k - 1] = temp;

}

}

// Ordenamiento por Insercion

**public** **void** insercion() {

**for** (**int** i = 1; i < **this**.vec.length; i++) {

**double** aux = **this**.vec[i];

**int** j;

**for** (j = i - 1; j >= 0 && **this**.vec[j] > aux; j--)

**this**.vec[j + 1] = **this**.vec[j];

**this**.vec[j + 1] = aux;

}

}

// Ordenamiento Seleccion

**public** **void** selecccion() {

**for** (**int** i = 0; i < **this**.vec.length - 1; i++) {

**int** min = i;

**for** (**int** j = i + 1; j < **this**.vec.length; j++)

**if** (**this**.vec[j] < **this**.vec[min])

min = j;

**if** (i != min) {

**double** aux = **this**.vec[i];

**this**.vec[i] = **this**.vec[min];

**this**.vec[min] = aux;

}

}

}

// Ordenamiento Shell

**public** **void** shell() {

**for** (**int** increment = **this**.vec.length / 2; increment > 0; increment =

(increment == 2 ? 1: (**int**) Math.*round*(increment / 2.2))) {

**for** (**int** i = increment; i < **this**.vec.length; i++)

**for** (**int** j = i; j >= increment

&& **this**.vec[j - increment] > **this**.vec[j];

j -= increment) {

**double** temp = **this**.vec[j];

**this**.vec[j] = **this**.vec[j - increment];

**this**.vec[j - increment] = temp;

}

}

}

// Ordenamiento QuickSort

**public** **void** quicksort(Vector vec, **int** primero, **int** ultimo) {

**int** i = primero, j = ultimo;

**double** pivote = **this**.vec[(primero + ultimo) / 2];

**double** auxiliar;

**do** {

**while** (**this**.vec[i] < pivote)

i++;

**while** (**this**.vec[j] > pivote)

j--;

**if** (i <= j) {

auxiliar = **this**.vec[j];

**this**.vec[j] = **this**.vec[i];

**this**.vec[i] = auxiliar;

i++;

j--;

}

} **while** (i <= j);

**if** (primero < j)

quicksort(vec, primero, j);

**if** (ultimo > i)

quicksort(vec, i, ultimo);

}

}

//Merge (Fusion) solo admite datos del tipo int

**public** **int**[] mergeSort(**int**[] vec) {

**return** mergeSort( vec, 0, vec.length-1, **new** **int**[vec.length]);

}

**public** **int**[] mergeSort(**int**[] vec, **int** izquierda, **int** derecha, **int**[] aux) {

**if** (derecha > izquierda) {

**int** mitad = (derecha + izquierda) / 2;

vec = mergeSort(vec, izquierda, mitad, aux);

vec = mergeSort(vec, mitad + 1, derecha, aux);

merge(vec, izquierda, mitad+1, derecha, aux);

}

**return** vec;

}

**public** **void** merge(**int** vec[], **int** izquierda, **int** medio, **int** derecha,

**int**[] aux) {

**int** izqEnd = medio - 1;

**int** tmpPos = izquierda;

**int** numElements = derecha - izquierda + 1;

**while** (izquierda <= izqEnd && medio <= derecha)

**if** (**this**.vec[izquierda] < (**this**.vec[medio]))

aux[tmpPos++] = **this**.vec[izquierda++];

**else**

aux[tmpPos++] = **this**.vec[medio++];

// Copy rest of first half

**while** (izquierda <= izqEnd)

aux[tmpPos++] = **this**.vec[izquierda++];

// Copy rest of der half

**while** (medio <= derecha)

aux[tmpPos++] = **this**.vec[medio++];

// Copio aux a vec.

**for** (**int** i = 0; i < numElements; i++, derecha--)

**this**.vec[derecha] = aux[derecha];

}