Algoritmos de ordenamiento

Julian Hidalgo

http://conclase.net

Algoritmos de ordenamiento

1. Introducción.

El ordenamiento es una labor común que realizamos continuamente. ¿Pero te has preguntado qué es ordenar? ¿No? Es que es algo tan corriente en nuestras vidas que no nos detenemos a pensar en ello. Ordenar es simplemente colocar información de una manera especial basándonos en un criterio de ordenamiento.

En la computación el ordenamiento de datos también cumple un rol muy importante, ya sea como un fin en sí o como parte de otros procedimientos más complejos. Se han desarrollado muchas técnicas en este ámbito, cada una con características específicas, y con ventajas y desventajas sobre las demás. Aquí voy a mostrarte algunas de las más comunes, tratando de hacerlo de una manera sencilla y comprensible.

2. Conceptos Preliminares.

Antes de comenzar a ver cada algoritmo vamos a ponernos de acuerdo en algunos conceptos, para que no haya confusiones:

- **Clave:** La parte de un registro por la cual se ordena la lista. Por ejemplo, una lista de registros con campos **nombre**, **direccion** y **telefono** se puede ordenar alfabéticamente de acuerdo a la clave **nombre**. En este caso los campos **direccion** y **telefono** no se toman en cuenta en el ordenamiento.
- **Criterio de ordenamiento (o de comparación):** EL criterio que utilizamos para asignar valores a los registros con base en una o más claves. De esta manera decidimos si un registro es ***mayor*** o ***menor*** que otro. En el pseudocódigo presentado más

adelante simplemente se utilizarán los símbolos **<** y **>**, para mayor simplicidad.
- **Registro:** Un grupo de datos que forman la lista. Pueden ser datos atómicos (enteros, caracteres, reales, etc.) o grupos de ellos, que en C equivalen a las estructuras.

Cuando se estudian algoritmos de todo tipo, no sólo de ordenamiento, es bueno tener una forma de evaluarlos antes de pasarlos a código, que se base en aspectos independientes de la plataforma o el lenguaje. De esta manera podremos decidir cuál se adapta mejor a los requerimientos de nuestro programa. Así que veamos estos aspectos:

- **Estabilidad:** Cómo se comporta con registros que tienen claves iguales. Algunos algoritmos mantienen el orden relativo entre éstos y otros no. Veamos un ejemplo. Si tenemos la siguiente lista de datos (nombre, edad): **"Pedro 19, Juan 23, Felipe 15, Marcela 20, Juan 18, Marcela 17",** y la ordenamos alfabéticamente por el *nombre* con un algoritmo estable quedaría así: **"Felipe 15, Marcela 20, Marcela 17, Juan 23, Juan 18, Pedro 19"**. Un algoritmo no estable podría dejar a **Juan 18** antes de **Juan 23**, o a **Marcela 20** después de **Marcela 17**.
- **Tiempo de ejecución:** La complejidad del algoritmo, que no tiene que ver con dificultad, sino con rendimiento. Es una función independiente de la implementación. Te la voy a explicar brevemente: tenemos que identificar una operación fundamental que realice nuestro algoritmo, que en este caso es comparar. Ahora contamos cuántas veces el algoritmo necesita comparar. Si en una lista de **n** términos realiza **n** comparaciones la complejidad es O(n). (En realidad es un poco más complicado que eso, pero lo vamos a hacer así: recuerda que dije que te iba a explicar brevemente). Algunos ejemplos de complejidades comunes son:
  - **O(1)** : Complejidad constante.
  - **O(n^2)** : Complejidad cuadrática.
  - **O(n log(n))** : Complejidad logarítmica.

Ahora podemos decir que un algoritmo de complejidad O(n) es más rápido que uno de complejidad $O(n^2)$. Otro aspecto a considerar es la diferencia entre el peor y el mejor caso. Cada algoritmo se comporta de modo diferente de acuerdo a cómo se le entregue la información; por eso es conveniente estudiar su comportamiento en casos extremos, como cuando los datos están prácticamente ordenados o muy desordenados.

- **Requerimientos de memoria:** El algoritmo puede necesitar memoria adicional para realizar su labor. En general es preferible que no sea así, pero es común en la programación tener que sacrificar memoria por rendimiento.

Hay bastantes otros aspectos que se pueden tener en cuenta, pero nosotros nos vamos a quedar con ésos.

Por último estableceremos algunas convenciones sobre el pseudocódigo:

- Vamos a ordenar la lista en forma ascendiente, es decir, de menor a mayor. Obviamente es esencialmente lo mismo que hacerlo en forma inversa.
- La forma de intercambiar los elementos depende de la estructura de datos: si es un arreglo (dinámico o estático) es necesario guardar una copia del primer elemento, asignarle el segundo al primero y el temporal al segundo. La variable temporal es necesaria, porque de lo contrario se perdería uno de los elementos. Si la estructura es una lista dinámica el procedimiento es parecido, pero se utilizan las direcciones de los elementos. En el pseudocódigo se utilizará el primer método.
- La lista se manejará como un arreglo de C: si tiene TAM elementos, el primer elemento es lista[0] y el último es lista[TAM-1]. Esto será así para todo el pseudocódigo presentado en este artículo.

Bien, ahora que ya tenemos todo claro vamos a lo que nos interesa...

3. Algoritmos más comunes.

La siguiente es una tabla comparativa de algunos algoritmos de ordenamiento. Si quieres saber más sobre alguno en particular haz un click sobre su nombre. En cada página encontrarás una descripción, pseudocódigo y un análisis sobre su rendimiento, ventajas y desventajas.

(Quizás quieras bajar ahora la demostración para ir observándola a medida que vayas leyendo)

Tabla comparativa de algoritmos

Nombre	Complejidad	Estabilidad	Memoria adicional
Ordenamiento Burbuja (Bubblesort)	$O(n^2)$	Estable	No
Ordenamiento por Selección	$O(n^2)$	No Estable	No
Ordenamiento por Inserción	$O(n^2)$	Estable	No
Ordenamiento Rápido (Quicksort)	$O(n * \log_2(n))$	No Estable	No

4. Eligiendo el más adecuado.

Ahora ya conoces una buena cantidad de algoritmos, pero... ¿cómo saber cuál es el que necesitas? ¿cuál es **EL** algoritmo?

Cada algoritmo se comporta de modo diferente de acuerdo a la cantidad y la forma en que se le presenten los datos, entre otras cosas. No existe EL algoritmo de ordenamiento. Sólo existe el mejor para cada caso particular. Debes conocer a fondo el problema que quieres resolver, y aplicar el más adecuado. Aunque hay algunas preguntas que te pueden ayudar a elegir:

- ***¿Qué grado de orden tendrá la información que vas a manejar?*** Si la información va a estar casi ordenada y no quieres complicarte, un algoritmo sencillo como el ordenamiento burbuja será suficiente. Si por el contrario los datos van a estar

muy desordenados, un algoritmo poderoso como Quicksort puede ser el más indicado. Y si no puedes hacer una presunción sobre el grado de orden de la información, lo mejor será elegir un algoritmo que se comporte de manera similar en cualquiera de estos dos casos extremos.

- ***¿Qué cantidad de datos vas a manipular?*** Si la cantidad es pequeña, no es necesario utilizar un algoritmo complejo, y es preferible uno de fácil implementación. Una cantidad muy grande puede hacer prohibitivo utilizar un algoritmo que requiera de mucha memoria adicional.
- ***¿Qué tipo de datos quieres ordenar?*** Algunos algoritmos sólo funcionan con un tipo específico de datos (enteros, enteros positivos, etc.) y otros son generales, es decir, aplicables a cualquier tipo de dato.
- ***¿Qué tamaño tienen los registros de tu lista?*** Algunos algoritmos realizan múltiples intercambios (burbuja, inserción). Si los registros son de gran tamaño estos intercambios son más lentos.

5. Demostración y Código Fuente.

Puedes descargar dos programas de demostración con los algoritmos presentados en este artículo:

OrdWin: En este programa puedes ver una demostración gráfica de cada algoritmo. También puedes experimentar ordenando listas de la longitud que quieras, observando el tiempo que demoran, la cantidad de comparaciones y de intercambios que realizan. Fue creado utilizando el compilador LccWin32 de Jacob Navia, pero el fichero descargable es un proyecto para Dev-C++. Incluye el ejecutable, el código fuente y este artículo completo.

Ordenar: Este programa es más indicado si lo que quieres es mirar código. No hay funciones gráficas ni nada del API de Windows. Debería funcionar en cualquier otro compilador sin mayores cambios, pues está hecho en ANSI C. Fue probado con éxito en Turbo C++, DJGPP, LccWin32, y Dev-C++. Quedó bastante

feo, pero es el precio que hay que pagar por la portabilidad ;-). Sólo incluye el código.

6. Algunas palabras para terminar.

No sabía si escribir este artículo o no. Probablemente no sea yo el indicado para hacerlo. Después de todo no soy ningún experto ni mucho menos, pero creo que puede ayudar a alguien que sepa menos que yo (no deben ser muchos :-)). Por eso pido tu colaboración para mejorar este documento y hacerlo algo útil. Si tienes sugerencias, comentarios o correcciones por favor házmelo saber.

7. Bibliografía.

- H.M. Deitel, P.J. Deitel: ***"Cómo programar en C/C++"***. Editorial Prentice Hall.
- Charles Bowman: ***"Algoritmos y estructuras de datos: Aproximación en C"***. Oxford University Press, 1999.
- ***"Dictionary of Algorithms, Data Structures, and Problems"***

1 Ordenamiento Burbuja (Bubblesort)

1. Descripción.

Este es el algoritmo más sencillo probablemente. Ideal para empezar. Consiste en ciclar repetidamente a través de la lista, comparando elementos adyacentes de dos en dos. Si un elemento es mayor que el que está en la siguiente posición se intercambian. ¿Sencillo no?

2. Pseudocódigo en C.

Tabla de variables

Nombre	Tipo	Uso
lista	Cualquiera	Lista a ordenar
TAM	Constante entera	Tamaño de la lista
i	Entero	Contador
j	Entero	Contador
temp	El mismo que los elementos de la lista	Para realizar los intercambios

```
1. for (i=1; i<TAM; i++)
2.      for j=0 ; j<TAM - 1; j++)
3.           if (lista[j] > lista[j+1])
4.                temp = lista[j];
5.                lista[j] = lista[j+1];
6.                lista[j+1] = temp;
```

3. Un ejemplo

Vamos a ver un ejemplo. Esta es nuestra lista:

4 - 3 - 5 - 2 - 1

Tenemos 5 elementos. Es decir, TAM toma el valor 5. Comenzamos comparando el primero con el segundo elemento. 4 es mayor que 3, así que intercambiamos. Ahora tenemos:

3 - **4** - 5 - 2 - 1

Ahora comparamos el segundo con el tercero: 4 es menor que 5, así que no hacemos nada. Continuamos con el tercero y el cuarto: 5 es mayor que 2. Intercambiamos y obtenemos:

3 - 4 - **2** - **5** - 1

Comparamos el cuarto y el quinto: 5 es mayor que 1. Intercambiamos nuevamente:

3 - 4 - 2 - **1** - **5**

Repitiendo este proceso vamos obteniendo los siguientes resultados:

3 - 2 - 1 - 4 - 5

2 - 1 - 3 - 4 - 5

1 - 2 - 3 - 4 - 5

4. Optimizando.

Se pueden realizar algunos cambios en este algoritmo que pueden mejorar su rendimiento.

- Si observas bien, te darás cuenta que en cada pasada a través de la lista un elemento va quedando en su posición final. Si no te queda claro mira el ejemplo de arriba. En la primera pasada el 5 (elemento mayor) quedó en la última posición, en la segunda el 4 (el segundo mayor elemento) quedó en la penúltima posición. Podemos evitar hacer comparaciones innecesarias si disminuimos el número de éstas en cada pasada. Tan sólo hay que cambiar el ciclo interno de esta manera:

  for (j=0; j<TAM - **i**; j++)
- Puede ser que los datos queden ordenados antes de completar el ciclo externo. Podemos modificar el algoritmo para que verifique si se han realizado intercambios. Si no se han hecho entonces terminamos con la ejecución, pues eso significa que los datos ya están ordenados. Te dejo como tarea que modifiques el algoritmo para hacer esto **:-)**.
- Otra forma es ir guardando la última posición en que se hizo un intercambio, y en la siguiente pasada sólo comparar hasta antes de esa posición.

5. Análisis del algoritmo.

Éste es el análisis para la versión no optimizada del algoritmo:

- Estabilidad: Este algoritmo nunca intercambia registros con claves iguales. Por lo tanto es *estable*.
- Requerimientos de Memoria: Este algoritmo sólo requiere de una variable adicional para realizar los intercambios.
- Tiempo de Ejecución: El ciclo interno se ejecuta **n** veces para una lista de n elementos. El ciclo externo también se ejecuta **n** veces. Es decir, la complejidad es n * n = O($\mathbf{n^2}$). El comportamiento del caso promedio depende del orden de entrada de los datos, pero es sólo un poco mejor que el del peor caso, y sigue siendo O($\mathbf{n^2}$).

Ventajas:

- Fácil implementación.
- No requiere memoria adicional.

Desventajas:

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

Este algoritmo es uno de los más pobres en rendimiento. Si miras la demostración te darás cuenta de ello. No es recomendable usarlo. Tan sólo está aquí para que lo conozcas, y porque su sencillez lo hace bueno para empezar. Ya veremos otros mucho mejores. Ahora te recomiendo que hagas un programa y lo pruebes. Si tienes dudas mira el programa de ejemplo.

2 Ordenamiento por Selección.

1. Descripción.

Este algoritmo también es sencillo. Consiste en lo siguiente:

- Buscas el elemento más pequeño de la lista.
- Lo intercambias con el elemento ubicado en la primera posición de la lista.
- Buscas el segundo elemento más pequeño de la lista.
- Lo intercambias con el elemento que ocupa la segunda posición en la lista.
- Repites este proceso hasta que hayas ordenado toda la lista.

2. Pseudocódigo en C.

Tabla de variables

Nombre	Tipo	Uso
lista	Cualquiera	Lista a ordenar
TAM	Constante entera	Tamaño de la lista
i	Entero	Contador
pos_men	Entero	Posición del menor elemento de la lista
temp	El mismo que los elementos de la lista	Para realizar los intercambios

```
1. for (i=0; i<TAM - 1; i++)
2.      pos_men = Menor(lista, TAM, i);
3.      temp = lista[i];
4.      lista[i] = lista [pos_men];
5.      lista [pos_men] = temp;
```

Nota: Menor(lista, TAM, i) es una función que busca el menor elemento entre las posiciones i y TAM-1. La búsqueda es lineal (elemento por elemento). No lo incluyo en el pseudocódigo porque es bastante simple.

3. Un ejemplo.

Vamos a ordenar la siguiente lista (la misma del ejemplo anterior **:-)**):

4 - 3 - 5 - 2 - 1

Comenzamos buscando el elemento menor entre la primera y última posición. Es el 1. Lo intercambiamos con el 4 y la lista queda así:

1 - 3 - 5 - 2 - **4**

Ahora buscamos el menor elemento entre la segunda y la última posición. Es el 2. Lo intercambiamos con el elemento en la segunda posición, es decir el 3. La lista queda así:

1 - **2** - 5 - **3** - 4

Buscamos el menor elemento entre la tercera posición (sí, adivinaste **:-D**) y la última. Es el 3, que intercambiamos con el 5:

1 - 2 - **3** - **5** - 4

El menor elemento entre la cuarta y quinta posición es el 4, que intercambiamos con el 5:

1 - 2 - 3 - **4** - **5**

¡Y terminamos! Ya tenemos nuestra lista ordenada. ¿Fue fácil no?

4. Análisis del algoritmo.

- Estabilidad: Aquí discrepo con un libro de la bibliografía que dice que no es estable. Yo lo veo así: si tengo dos registros con claves iguales, el que ocupe la posición más baja será el primero que sea identificado como *menor*. Es decir que será el primero en ser desplazado. El segundo registro será el menor en el siguiente ciclo y quedará en la posición adyacente. Por lo tanto se mantendrá el *orden relativo*. Lo que podría hacerlo inestable sería que el ciclo que busca el elemento menor revisara la lista desde la última posición hacia atrás. ¿Qué opinas tú? Yo digo que es *estable*, pero para hacerle caso al libro (el autor debe sabe más que yo ¿cierto?**:-)**) vamos a decir que *no es estable*.
- Requerimientos de Memoria: Al igual que el ordenamiento burbuja, este algoritmo sólo necesita una variable adicional para realizar los intercambios.
- Tiempo de Ejecución: El ciclo externo se ejecuta **n** veces para una lista de n elementos. Cada búsqueda requiere comparar todos los elementos no clasificados. Luego la complejidad es **O(n^2)**. Este algoritmo presenta un comportamiento constante independiente del orden de los datos. Luego la complejidad promedio es también **O(n^2)**.

Ventajas:

- Fácil implementación.
- No requiere memoria adicional.
- Realiza pocos intercambios.
- Rendimiento constante: poca diferencia entre el peor y el mejor caso.

Desventajas:

- Lento.
- Realiza numerosas comparaciones.

Este es un algoritmo lento. No obstante, ya que sólo realiza un intercambio en cada ejecución del ciclo externo, puede ser una buena opción para listas con registros grandes y claves pequeñas. Si miras el programa de demostración notarás que es el más rápido en la parte gráfica (por lo menos en un PC lento y con una tarjeta gráfica mala como el mío x-|). La razón es que es mucho más lento dibujar las barras que comparar sus largos (el desplazamiento es más costoso que la comparación), por lo que en este caso especial puede vencer a algoritmos como Quicksort.

Bien, ya terminamos con éste. Otra vez te recomiendo que hagas un programa y trates de implementar este algoritmo, de preferencia sin mirar el código ni el pseudocódigo otra vez.

3 Ordenamiento por Inserción

1. Descripción.

Este algoritmo también es bastante sencillo. ¿Has jugado cartas?. ¿Cómo las vas ordenando cuando las recibes? Yo lo hago de esta manera: tomo la primera y la coloco en mi mano. Luego tomo la segunda y la comparo con la que tengo: si es mayor, la pongo a la derecha, y si es menor a la izquierda (también me fijo en el color, pero omitiré esa parte para concentrarme en la idea principal). Después tomo la tercera y la comparo con las que tengo en la mano, desplazándola hasta que quede en su posición final. Continúo haciendo esto, *insertando* cada carta en la posición que le corresponde, hasta que las tengo todas en orden. ¿Lo haces así tu también? Bueno, pues si es así entonces comprenderás fácilmente este algoritmo, porque es el mismo concepto.

Para simular esto en un programa necesitamos tener en cuenta algo: no podemos desplazar los elementos así como así o se perderá un elemento. Lo que hacemos es guardar una copia del elemento actual (que sería como la carta que tomamos) y desplazar todos los elementos mayores hacia la derecha. Luego copiamos el elemento guardado en la posición del último elemento que se desplazó.

2. Pseudocódigo en C.

Tabla de variables

Nombre	Tipo	Uso
lista	Cualquiera	Lista a ordenar
TAM	Constante Entera	Tamaño de la lista

Nombre	Tipo	Uso
i	Entero	Contador
j	Entero	Contador
temp	El mismo que los elementos de la lista	Para realizar los intercambios

```
1. for (i=1; i<TAM; i++)
2.      temp = lista[i];
3.      j = i - 1;
4.      while ( (lista[j] > temp) && (j >= 0) )
5.           lista[j+1] = lista[j];
6.           j--;
7.      lista[j+1] = temp;
```

Nota: Observa que en cada iteración del ciclo externo los elementos 0 a i forman una lista ordenada.

3. Un ejemplo

¿Te acuerdas de nuestra famosa lista?

4 - 3 - 5 - 2 - 1

temp toma el valor del segundo elemento, 3. La *primera carta* es el 4. Ahora comparamos: 3 es menor que 4. Luego desplazamos el 4 una posición a la derecha y después copiamos el 3 en su lugar.

4 - **4** - 5 - 2 - 1

3 - 4 - 5 - 2 - 1

El siguiente elemento es 5. Comparamos con 4. Es mayor que 4, así que no ocurren intercambios.

Continuamos con el 2. Es menor que cinco: desplazamos el 5 una posición a la derecha:

3 - 4 - 5 - **5** - 1

Comparamos con 4: es menor, así que desplazamos el 4 una posición a la derecha:

3 - 4 - **4** - 5 - 1

Comparamos con 3. Desplazamos el 3 una posición a la derecha:

3 - **3** - 4 - 5 - 1

Finalmente copiamos el 2 en su posición final:

2 - 3 - 4 - 5 - 1

El último elemento a ordenar es el 1. Cinco es menor que 1, así que lo desplazamos una posición a la derecha:

2 - 3 - 4 - 5 - **5**

Continuando con el procedimiento la lista va quedando así:

2 - 3 - 4 - **4** - 5

2 - 3 - **3** - 4 - 5

2 - **2** - 3 - 4 - 5

1 - 2 - 3 - 4 - 5

Espero que te haya quedado claro.

4. Análisis del algoritmo.

- Estabilidad: Este algoritmo nunca intercambia registros con claves iguales. Por lo tanto es *estable*.
- Requerimientos de Memoria: Una variable adicional para realizar los intercambios.
- Tiempo de Ejecución: Para una lista de **n** elementos el ciclo externo se ejecuta **n-1** veces. El ciclo interno se ejecuta como máximo una vez en la primera iteración, 2 veces en la segunda, 3 veces en la tercera, etc. Esto produce una complejidad $O(n^2)$.

Ventajas:

- Fácil implementación.
- Requerimientos mínimos de memoria.

Desventajas:

- Lento.
- Realiza numerosas comparaciones.

Este también es un algoritmo lento, pero puede ser de utilidad para listas que están ordenadas o semiordenadas, porque en ese caso realiza muy pocos desplazamientos.

4 Ordenamiento Rápido (Quicksort)

1. Descripción.

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). 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).

Al finalizar este procedimiento el elemento de división queda en una posición en que todos los elementos a su izquierda son menores que él, y los que están a su derecha son mayores.

2. Pseudocódigo en C.

Tabla de variables

Nombre	Tipo	Uso
lista	Cualquiera	Lista a ordenar
inf	Entero	Elemento inferior de la lista
sup	Entero	Elemento superior de la lista
elem_div	El mismo que los elementos de la lista	El elemento divisor
temp	El mismo que los elementos de la lista	Para realizar los intercambios
i	Entero	Contador por la izquierda
j	Entero	Contador por la derecha
cont	Entero	El ciclo continua mientras cont tenga el valor 1

```
Nombre Procedimiento: OrdRap
Parámetros:
   lista a ordenar (lista)
   índice inferior (inf)
   índice superior (sup)

   // Inicialización de variables
   1. elem_div = lista[sup];
   2. i = inf - 1;
   3. j = sup;
   4. cont = 1;

   // Verificamos que no se crucen los límites
   5. if (inf >= sup)
   6.      retornar;

   //  Clasificamos la sublista
   7. while (cont)
   8.      while (lista[++i] < elem_div);
   9.      while (lista[--j] > elem_div);
  10.      if (i < j)
  11.           temp = lista[i];
  12.           lista[i] = lista[j];
  13.           lista[j] = temp;
  14.      else
  15.           cont = 0;

  // Copiamos el elemento de división
  // en su posición final
  16. temp = lista[i];
  17. lista[i] = lista[sup];
  18. lista[sup] = temp;

  // Aplicamos el procedimiento
  // recursivamente a cada sublista
  19. OrdRap (lista, inf, i - 1);
  20. OrdRap (lista, i + 1, sup);
```

Nota: La primera llamada debería ser con la lista, cero (0) y el tamaño de la lista menos 1 como parámetros.

3. Un ejemplo

Esta vez voy a cambiar de lista ;-D

5 - 3 - 7 - 6 - 2 - 1 - 4

Comenzamos con la lista completa. El elemento divisor será el 4:

5 - 3 - 7 - 6 - 2 - 1 - **4**

Comparamos con el 5 por la izquierda y el 1 por la derecha.

5 - 3 - 7 - 6 - 2 - **1** - **4**

5 es mayor que cuatro y 1 es menor. Intercambiamos:

1 - 3 - 7 - 6 - 2 - **5** - **4**

Avanzamos por la izquierda y la derecha:

1 - **3** - 7 - 6 - **2** - 5 - **4**

3 es menor que 4: avanzamos por la izquierda. 2 es menor que 4: nos mantenemos ahí.

1 - 3 - **7** - 6 - **2** - 5 - **4**

7 es mayor que 4 y 2 es menor: intercambiamos.

1 - 3 - **2** - 6 - **7** - 5 - **4**

Avanzamos por ambos lados:

1 - 3 - 2 - **6** - 7 - 5 - **4**

En este momento termina el ciclo principal, porque los índices se cruzaron. Ahora intercambiamos lista[i] con lista[sup] (pasos 16-18):

1 - 3 - 2 - **4** - 7 - 5 - **6**

Aplicamos recursivamente a la sublista de la izquierda (índices 0 - 2). Tenemos lo siguiente:

1 - **3** - **2**

1 es menor que 2: avanzamos por la izquierda. 3 es mayor: avanzamos por la derecha. Como se intercambiaron los índices termina el ciclo. Se intercambia lista[i] con lista[sup]:

1 - **2** - **3**

Al llamar recursivamente para cada nueva sublista (lista[0]-lista[0] y lista[2]-lista[2]) se retorna sin hacer cambios (condición 5.).Para resumir te muestro cómo va quedando la lista:

Segunda sublista: lista[4]-lista[6]

7 - **5** - **6**

5 - **7** - **6**

5 - **6** - **7**

Para cada nueva sublista se retorna sin hacer cambios (se cruzan los índices).

Finalmente, al retornar de la primera llamada se tiene el arreglo ordenado:

1 - 2 - 3 - 4 - 5 - 6 - 7

Eso es todo. Bastante largo ¿verdad?

4. Optimizando.

Sólo voy a mencionar algunas optimizaciones que pueden mejorar bastante el rendimiento de quicksort:

- Hacer una versión iterativa: Para ello se utiliza una pila en que se van guardando los límites superior e inferior de cada sublista.
- No clasificar todas las sublistas: Cuando el largo de las sublistas va disminuyendo, el proceso se va *encareciendo*. Para solucionarlo sólo se clasifican las listas que tengan un largo menor que **n**. Al terminar la clasificación se llama a otro algoritmo de ordenamiento que termine la labor. El indicado es uno que se comporte bien con listas casi ordenadas, como el ordenamiento por inserción por ejemplo. La elección de **n** depende de varios factores, pero un valor entre 10 y 25 es adecuado.
- Elección del elemento de división: Se elige desde un conjunto de tres elementos: lista[inferior], lista[mitad] y lista[superior]. El elemento elegido es el que tenga el valor medio según el criterio de comparación. Esto evita el comportamiento degenerado cuando la lista está prácticamente ordenada.

5. Análisis del algoritmo.

- Estabilidad: No es *estable*.
- Requerimientos de Memoria: No requiere memoria adicional en su forma recursiva. En su forma iterativa la necesita para la pila.
- Tiempo de Ejecución:
  - Caso promedio. La complejidad para dividir una lista de **n** es O(n). Cada sublista genera en promedio dos sublistas más de largo n/2. Por lo tanto la complejidad se define en forma recurrente como:

$$f(1) = 1$$

$$f(n) = n + 2\, f(n/2)$$

La forma cerrada de esta expresión es:

$$f(n) = n \log_2 n$$

Es decir, la complejidad es **$O(n \log_2 n)$**.

- El peor caso ocurre cuando la lista ya está ordenada, porque cada llamada genera sólo una sublista (todos los elementos son menores que el elemento de división). En este caso el rendimiento se degrada a $O(n^2)$. Con las optimizaciones mencionadas arriba puede evitarse este comportamiento.

Ventajas:

- Muy rápido
- No requiere memoria adicional.

Desventajas:

- Implementación un poco más complicada.
- Recursividad (utiliza muchos recursos).
- Mucha diferencia entre el peor y el mejor caso.

La mayoría de los problemas de rendimiento se pueden solucionar con las optimizaciones mencionadas arriba (al costo de complicar mucho más la implementación). Este es un algoritmo que puedes utilizar en la vida real. Es muy eficiente. En general será la mejor opción. Intenta programarlo. Mira el código si tienes dudas.

Tabla de contenido