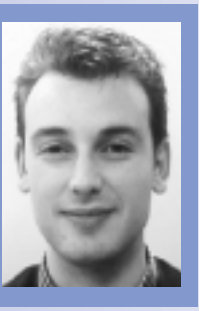
Algoritmos de ordenación (II)

Todos sabemos que los ordenadores, entre otras cosas, sirven para ordenar. Pero ¿Cómo lo hacen? ¿Existen distintos modos de hacerlo? ¿Depende de lo que tengamos que ordenar? ¿Es tan crítico elegir correctamente el método de ordenación?

En este artículo divulgativo se pretende dar respuesta a estas y a otras preguntas, mostrando como funcionan los algoritmos de ordenación más importantes y discutiendo las claves a tener en cuenta para seleccionar el algoritmo más apropiado para un problema concreto.

Eugenio Francisco Sánchez Úbeda

Doctor Ingeniero del ICAI
Instituto de Investigación Tecnológica y Departamento de Sistemas Informáticos de la Universidad Pontificia Comillas de Madrid, donde es coordinador del Área de Sistemas Inteligentes (ASI)
E-mail: Eugenio.Sanchez@iit.upco.es



Introducción

En la primera parte de este artículo, publicado en el número anterior de esta revista (Vol. LXXVIII, Fascículo II, pp. 28-32), se ha argumentado la importancia de los algoritmos de ordenación; explicándose por qué es crítico seleccionar correctamente el algoritmo de ordenación y discutiendo las claves a tener en cuenta para realizar dicha elección correctamente.

En computación los algoritmos de ordenación se pueden clasificar atendiendo a distintos criterios. Una posible clasificación consiste en distinguir entre algoritmos simples y sofisticados. Anteriormente se ha mostrado como funciona Inserción Directa, uno de los algoritmo simples más utilizados.

En esta segunda parte del artículo se describen Selección Directa y la Burbuja, otros dos algoritmos simples, así como los dos algoritmos sofisticados más importantes: Heapsort y Quicksort. Finalmente se analiza el tiempo de ejecución de estos algoritmos desde un punto de vista teórico y práctico.

Algoritmos

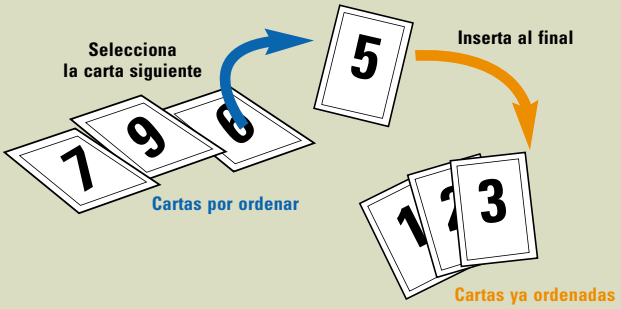
Selección directa

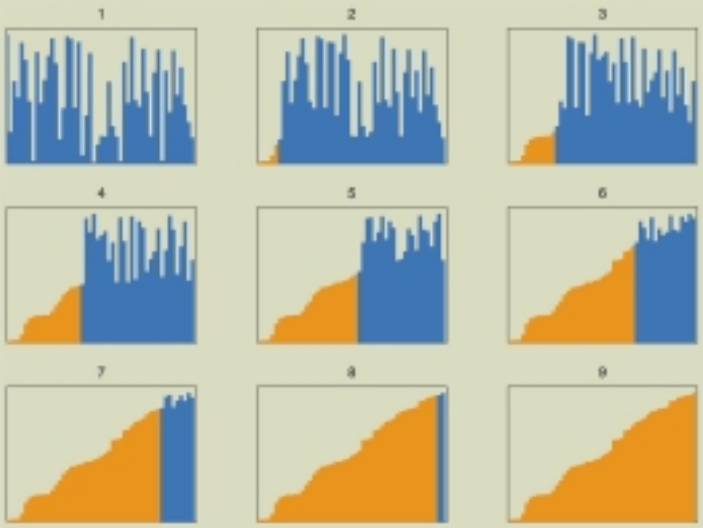
Este algoritmo utiliza otra forma de ordenar que también la empleamos con frecuencia en la vida cotidiana y que, en cierta

FIGURA 6. SELECCIÓN DIRECTA

Hay dos zonas: una zona desordenada (en azul) y una zona ordenada (en naranja).

En cada paso se selecciona de la zona desordenada el elemento menor y se inserta al final de la secuencia.





Selección Directa(A)

```
1 For i=1 to Tamaño(A) -1
        k=i; x = A[i];
        2 For j=i+1 to Tamaño (A)  % selección
              if (A[j]<x) then k=j; x=A[j];
        end 2
        A[k]=A[i];      % inserta al final
        A[i]=x;
end 1
```

Elementos de la zona ordenada

Elementos de la zona desordenada

manera, es la inversa u "opuesta" a como trabaja Inserción Directa. Nuevamente los elementos se dividen en dos secuencias, una ordenada y otra desordenada.

En cada paso se selecciona de la zona desordenada el elemento con clave menor y se inserta al final de la secuencia ordenada. El nombre de este algoritmo hace referencia al paso más costoso del mismo: la búsqueda o selección en la zona desordenada del elemento más pequeño. Esta búsqueda se realiza de forma exhaustiva recorriendo sistemáticamente toda la secuencia desordenada. Lógicamente, ahora los elementos de la zona desordenada siempre tienen una clave mayor o igual que los elementos de la zona ordenada.

Si tenemos que ordenar pocos datos y no tenemos conocimiento alguno sobre el grado de desorden que presentan; o bien, son pocos datos y tenemos la certeza de que están muy desordenados, entonces debemos utilizar Selección Directa.

Burbuja (el peor)

Este algoritmo simple explota otra idea básica de ordenación, el intercambio de elementos. Si dos elementos están mal colocados, se intercambian. Es necesario hacer repetidas pasadas sobre el vector para conseguir ordenar los datos.

Aunque este algoritmo aparece frecuentemente en la literatura, a parte del interés didáctico que pueda tener, nunca debe emplearse en la práctica. De he-

cho, un análisis sencillo del mismo demuestra de forma teórica y contundente que es inferior a otros algoritmos simples como Inserción o Selección Directa.

Algoritmos sofisticados

Heapsort

Ya se ha comentado que Selección Directa se basa en ir seleccionando y sacando repetidamente el mínimo de la secuencia desordenada. Si en un momento dado hay N elementos en la zona desordenada, para encontrar el mínimo Selección Directa recorrerá dicha secuencia realizando N-1 comparaciones. En la siguiente iteración la zona desordenada tendrá N-1 elementos, siendo necesarias N-2 comparaciones para encontrar el nuevo mínimo.

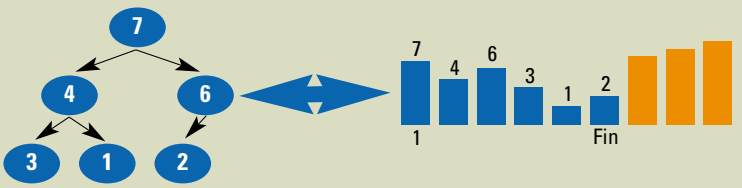
Heapsort nace como una sofisticación de Selección Directa. La idea sigue siendo la misma (buscar en la zona desordenada e insertar en la ordenada), pero con una mejora sustancial en el proceso de búsqueda de la zona desordenada.

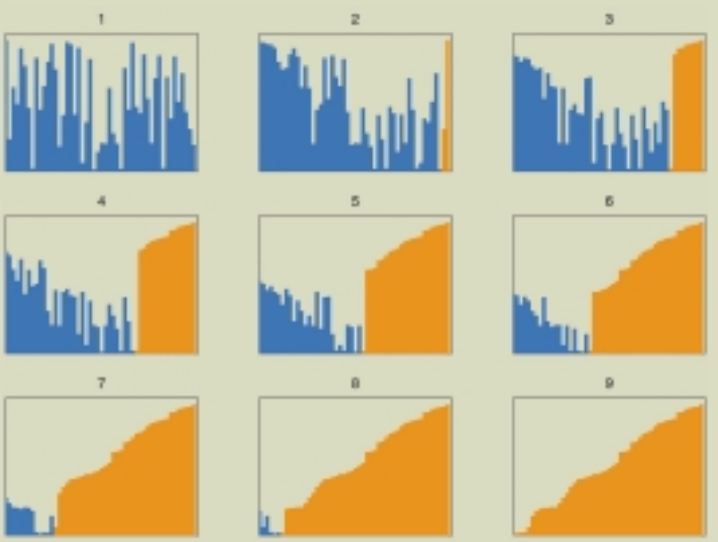
Heapsort consigue acelerar búsquedas sucesivas del elemento siguiente a insertar guardando en la búsqueda anterior mucha más información de la zona desordenada que meramente la posición del elemento a insertar (como se hace en Selección Directa). Para ello, el algoritmo emplea una estructura especial denominada montículo ("heap" en inglés, de ahí el nombre del algoritmo).

FIGURA 7. HEAPSORT

Hay dos zonas: una zona desordenada (en azul) y una zona ordenada (en naranja).

En cada paso se selecciona de la zona desordenada el elemento mayor y se inserta al final de la secuencia ordenada. Una estructura montículo en la zona desordenada permite una selección muy rápida del elemento mayor.





Heapsort (A)

```
ConstruyeMonticuloinicial  (A);
Fin = Tamaño(A);  % montículo de 1 a Fin
1 For i = N to 2, step -1
        intercambiar A[1] con A[i];
        Fin = Fin - 1;
        Monticuliza (A, Fin, 1);
end 1
```

Construyemonticuloinicial (A)

```
Fin = N;
1 For i = floor (N/2) to 1, step -1
        Monticuliza (A, Fin, i);
end 1
```

Monticuliza (A, Fin, i)

```
iz=2*i ; de = 2*i+1;
1 if (iz ≤ Fin & A[iz]>A[i]) then
        mayor = iz;
else 1
        mayor = i;
end 1
2 if (de ≤ Fin & A[de]>A[mayor])  then
        mayor = de;
end 2
3 if (mayor ≠ i) then
        intercambiar A[i] con A [mayor];
        Monticuliza  (A, Fin, mayor);
end3
```

Un montículo es una estructura en donde cada elemento (nodo) está relacionado con otros elementos, formando un árbol. Cada nodo puede tener un padre y como mucho dos hijos. El nodo inicial o raíz se caracteriza por no tener padre. Además, en un montículo cada nodo debe ser mayor que todos sus descendientes.

Para determinar rápidamente cuál es el elemento siguiente a insertar, el algoritmo mantiene actualizada la información de la zona desordenada (montículo) con un coste computacional muy bajo.

Además, Heapsort ordena in situ ya que el montículo se puede construir directamente sobre la zona desordenada, utilizando la posición en el vector para determinar cuál es el nodo padre y los hijos derecha e izquierda.

Dado que ahora los pasos del algoritmo son más complejos, con un manejo de los datos mucho más laborioso, puede no compensar emplear Heapsort si son pocos los datos a ordenar. Un algoritmo simple como Inserción o Selección Directa puede ser más rápido. Por otro lado, si tenemos bastantes datos a ordenar, seguramente será más interesante utilizar Quicksort.

Quicksort

En 1962 Hoare publicó un nuevo algoritmo, al que bautizó con el nombre de Quicksort ("ordenación rápida"). Han pasado casi 40 años y continúa siendo el algoritmo de ordenación preferido, a pesar del esfuerzo realizado durante este tiempo para diseñar algoritmos más rápidos.

Hay varias razones para tal supremacía: es muy eficiente en la mayoría de las situaciones, no emplea espacio adicional para realizar la ordenación (es in situ) y trabaja muy bien incluso en entornos de memoria virtual.

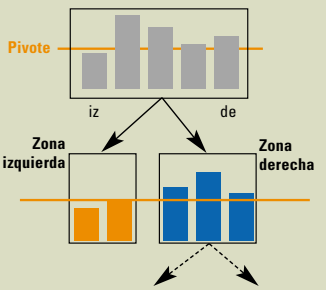
Quicksort emplea la estrategia "divide y vencerás" para resolver el problema de ordenación. Esta estrategia la utilizamos con frecuencia en la vida cotidiana. Por ejemplo, si tene-

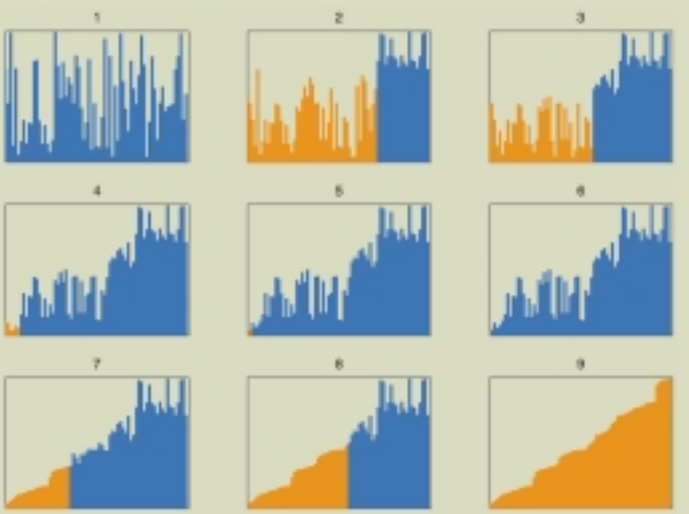
FIGURA 8. QUICKSORT

```
Quicksort (A, iz, de)
1 if (iz < de) then
        d = Divide (A, iz, de);
        Quicksort (A, iz, d);
        Quicksort (A, d+1, de);
end 1

Divide (A, iz, de)
r= numero_aleatorio_entre (iz, de);
intercambiar A[iz] con A [r];
pivote = A[iz];
i = iz ; j = de;
1 repetir
        while (A[i] < pivote) i = i+1; end;
        while (pivote < A[j]) j = j- 1; end;
        2 if (i<j) then
                intercambiar A[i] con A[j];
                i=i+1; j=j-1;
        else
                return j;
        end 2
end 1

Empezar con Quicksort(A,1, Tamaño(A))
```

Basado en la estrategia "divide y vencerás", es el algoritmo de ordenación preferido.

En cada paso genera un subgrupo con todos los elementos menores al pivote y otro con los mayores a él. Colocando los elementos del primer subgrupo (en naranja) a la izquierda de los del segundo (en azul) y repitiendo el proceso recursivamente se ordenan todos los datos.





mos muchas facturas de varios años y las queremos ordenar por fecha, uno puede dividir las facturas en varios montones, uno por año. De este modo, podemos ordenar cada montón con independencia del resto, sabiendo que, una vez ordenados todos los montones, tendremos ordenadas todas las facturas. Para ordenar el montón de facturas de un año se puede repetir la misma estrategia, dividiendo ese montón en varios subgrupos, uno por trimestre. Nuevamente, podríamos dividir cada trimestre en meses y éstos a su vez en días. Ordenados todos meses de un trimestre, tendríamos el trimestre ordenado, y ordenados todos los trimestres de un año, el año completo.

Ahora bien, ¿Qué ocurre si al separar las facturas por años, todas menos una son del mismo año? ¿y si al intentar clasificar estas últimas por meses, todas menos una son del mismo mes? Evidentemente algo no va bien. Hemos revisado dos veces todas las facturas (con el tiempo que esto supone) y sólo tenemos dos ordenadas. Esta estrategia de ordenación sólo es interesante si al dividir un grupo de elementos se consiguen subgrupos con un número similar de elementos, es decir, cuando obtenemos subproblemas de complejidad claramente inferior a la del problema original.

El éxito de Quicksort radica en su capacidad para dividir rápidamente un conjunto de datos en dos particiones bastante equilibradas. Para ello emplea un pivote, generando una partición o subgrupo con todos los elementos menores al pivote y la otra con los mayores a él. Colocando los elementos del primer subgrupo a la izquierda de los del segundo se consigue avanzar en la ordenación. Repitiendo recursivamente este proceso con cada subgrupo de elementos se consigue ordenar finalmente todos los datos. Lógicamente, el proceso de división termina cuando tengamos un conjunto con dos elementos.

La selección del pivote es determinante para asegurar la rapidez del algoritmo. Idealmente el pivote debería ser la mediana, pues garantizaría una partición óptima (dos grupos de igual tamaño). Desgraciadamente en el cálculo de la mediana hay implícita una ordenación, por lo que su cálculo exacto está descartado. Existen varias formas de seleccionar un pivote razonable, entre las que destacan la estimación de la mediana con tres muestras (primer elemento, central y último), o la selección aleatoria de un elemento entre el primero y el último del grupo a dividir.

Por último comentar que las versiones sofisticadas de Quicksort no aplican la estrategia de partición hasta el final. Cuando el número de elementos es inferior a un cierto umbral, se emplea un algoritmo simple como Inserción Directa para terminar la ordenación de ese subconjunto. Un umbral razonable, basado en la experiencia, puede ser 10 elementos.

Análisis de los algoritmos

El tiempo necesario para ejecutar un algoritmo de ordenación depende, entre otras cosas, de la cantidad de datos a ordenar. El sentido común nos dice que ordenar más datos debe ser más costoso que ordenar menos ¿Pero cuánto?

Existe una herramienta matemática que permite expresar de forma rigurosa el tiempo de ejecución de un algoritmo, la llamada notación asintótica. Como su nombre indica, esta notación sólo es apropiada para una situación asintótica en la que el número N de elementos a ordenar sea suficientemente grande. Por ejemplo, si decimos que el tiempo de ejecución de un algoritmo es $O(N^2)$, significa que, en el peor de los casos y para N suficientemente grande, el tiempo de ejecución del algoritmo crecerá de forma cuadrática con N. Para N pequeño no tiene por qué ser así.

Utilizando esta herramienta matemática, se han realizado numerosos estudios teóricos que permiten comparar el comportamiento asintótico de los distintos algoritmos de forma abstracta y en distintos casos hipotéticos (típicamente en el caso mejor, caso medio y peor).

En estos estudios no se tiene en cuenta consideraciones prácticas importantes como la calidad del compilador utilizado o la gestión de memoria que en la actualidad realizan los ordenadores (denominada caching[1]). Además,

[1] Esta técnica de gestión de la memoria hace que sea más rápido acceder a posiciones consecutivas de un vector de datos.

FIGURA 9. ANÁLISIS PRÁCTICO

Datos ya ordenados

Coincide con el Caso Mejor para los algoritmos naturales.

Inserción directa es el más rápido ya que sólo comprueba que los datos están ordenados.

Datos aleatorios

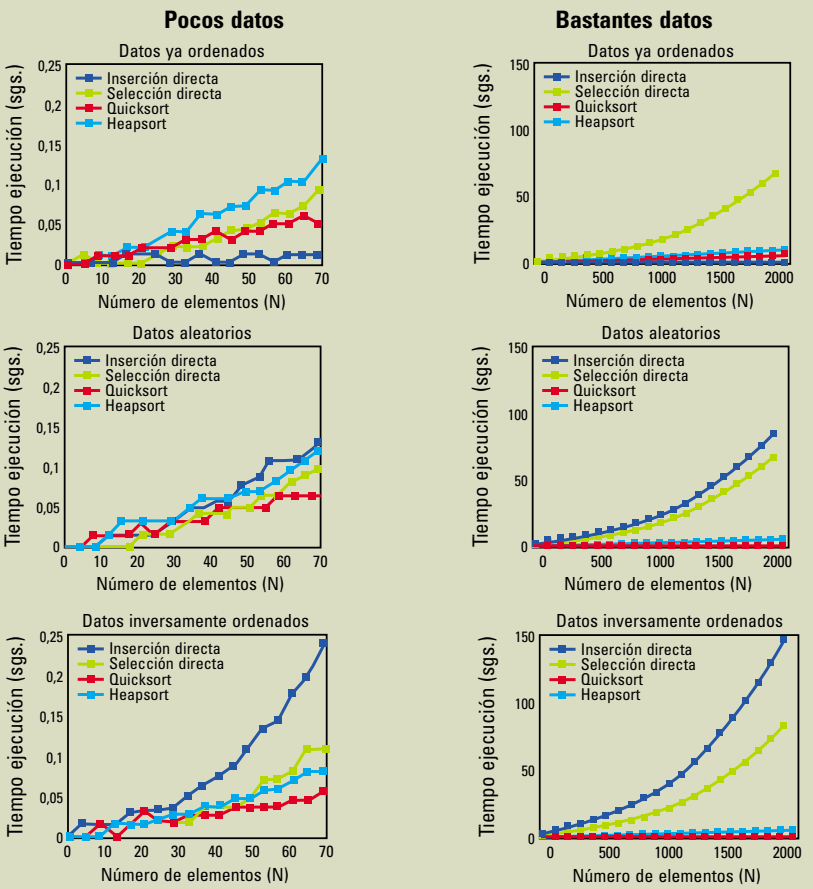
Coincide con el Caso Medio para los algoritmos naturales.

Quicksort es el más rápido para N suficientemente grande.

Datos inversamente ordenados

Coincide con el Caso Peor para los algoritmos naturales.

Heapsort tiene un comportamiento un poco antinatural.



suele ser difícil determinar para un algoritmo complejo cuáles son los casos peor, medio y peor.

Todo ello hace que sea práctica común el analizar los algoritmos de forma experimental, midiendo el tiempo de ejecución necesario para ordenar vectores de datos de distinto tamaño y diferente nivel de desorden. Típicamente se suele estudiar el comportamiento en tres situaciones diferentes:

- Vector ya ordenado
- Vector desordenado (datos aleatorios)
- Vector inversamente ordenado

Aunque es de esperar que estas tres situaciones coincidan con los casos mejor, medio y peor del algoritmo, puede ocurrir que no sea así, en cuyo caso se dice que el algoritmo es antinatural. Heapsort es un poco antinatural ya que tarda más cuando los datos están ya ordenados. Inserción Directa es un ejemplo claro de algoritmo natural.

En general los algoritmos simples son O (N^2), mientras que los algoritmos sofisticados son O (N log N). Por tanto, para N suficientemente grande (un valor relativamente pequeño en la práctica) los algoritmos simples no pueden competir en velocidad con los sofisticados. Una excepción es Inserción Directa; si los datos están ya ordenados (o casi), entonces el algoritmo es O (N), por lo que es superior incluso a Quicksort.

Sin embargo, si tenemos que ordenar muy pocos datos, entonces los algoritmos simples pueden ser más veloces que los sofisticados. En esta situación la notación asintótica no refleja el comportamiento del algoritmo.

Además, también se puede observar que Selección Directa es preferible a Inserción Directa, salvo que los datos estén ordenados. En cuanto a los algoritmos sofisticados, Quicksort es un poco más rápido que Heapsort.

Conclusiones

Aunque se ha mostrado el funcionamiento de algunos de los algoritmos de ordenación más importantes, existen otros muchos que pueden resultar muy útiles en determinadas circunstancias. Por ejemplo, existen algoritmos especiales que para ordenar no realizan ningún tipo de comparación entre elementos. Estos algoritmos son extremadamente rápidos (de orden lineal) si se tiene la seguridad de que los datos a ordenar cumplen ciertos requisitos (unos exigen que las claves sean números enteros menores que una determinada cantidad, otros que sean valores distribuidos uniformemente entre 0 y 1, etc.).

Por otro lado, los algoritmos de ordenación presentados pertenecen a la categoría de los llamados "algoritmos de ordenación interna". Sin embargo, también existen otros algoritmos, denominados de "ordenación externa", especialmente diseñados para cuando la masa de datos a ordenar no cabe en memoria RAM. Estos algoritmos dependen sensiblemente de la tecnología de almacenamiento externo empleada (cintas, discos duros, CDs, etc.).

Para mayor información, se recomienda consultar la bibliografía que se adjunta, en donde se puede encontrar abundante información sobre el tema.

Bibliografía

[1] T.H. Cormen, C.E. Leiseron y R.L. Rivest, Introduction to Algorithms, The MIT Press, 1994.
[2] C.A.R. Hoare, "Quicksort", The Computer Journal, Vol. 5, No. 1, pp. 10-15, Abril 1962.
[3] J. Jaja, "A perspective on Quicksort", Computing in Science and Engineering, pp. 43-49, Enero-Febrero 2000.
[4] D.E. Knuth, El arte de programar ordenadores. Volumen 3: Ordenación y búsqueda, Editorial Reverté, 1986.
[5] R. Sedgewick, Algoritmos en C++, Editorial Díaz de Santos, 1995.
[6] M.A. Weiss, Estructuras de datos en Java, Addison Wesley, 2000.
[7] N. Wirth, Algoritmos + estructuras de datos = programas, Ediciones del Castillo, 1985.