Algoritmos de ordenamiento

Octavio Alberto Agustín Aquino

20 de diciembre de 2005

Índice

1. Introducción

El problema de la ordenación es antiguo en lo que refiere al estudio de algoritmos y la computación. Bien se puede decir que en estos casos "nada nuevo hay bajo el Sol" pues muchas soluciones del problema son clásicas.

Dicho problema es comúnmente de la siguiente forma (o en muchos casos puede reducirse a éste):

Entrada: Una sucesión de n números $A = \{a_1, \ldots, a_n\}$.

Salida: Una permutación $A' = \{a'_1, \ldots, a'_n\}$ de la secuencia original tal que $a'_1 \leq \ldots \leq a'_n$.

Esencialmente, existen dos tipos de algoritmos de ordenamiento: los que dependen de comparaciones y cuyas complejidades se encuentran entre lo polinomial y lo logarítmico, y los de conteo, que asumen ciertas cosas sobre el conjunto de datos y cuyos tiempos son lineales. Hablaremos sucintamente de cada caso con sus correspondientes ejemplares más comunes, sólo para los polinomiales y logarítmicos.

También es prudente aquí señalar el uso de la notación O.

Definición 1 *Para dos funciones f y g se dice que f es de orden g si existen $a, b \in \mathbb{Z}^+$ tales que $f(n) \leq ag(n)$ para $n \geq b$. Se escribe así que f es $O(g(n))$.*

Siempre se tomará la función más sencilla posible que satisfaga la desigualdad de la definición. Así, por ejemplo, si $f(n) \leq n^2$, entonces se escribe simplemente que f es $O(n^2)$, a pesar de que también f es $O(2n^2)$.

Otra forma de entender el orden es decir que la función f está *asintóticamente acotada* por g, es decir, que la gráfica de f no sobrepasa a la de g a partir de un cierto punto.

No es difícil ver que la propiedad de orden en transitiva y aditiva, en el sentido de que la complejidad de una suma finita de funciones con un mismo orden tiene el mismo orden que sus sumandos.

2. Método de la burbuja

El método de la burbuja es un ejemplo típico de ordenación por comparación de orden $O(n^2)$. El algoritmo es el siguiente. Se sigue denotando la sucesión de datos por A, que en este caso se representa como un arreglo de enteros en C.

```
void burbuja(int *A, int n)
{
 int i, j, temp;
  for(i=0; i<n;  i++)
   {
    for(j=0; j<n-1; j++)
     {
      if(A[j+1]<A[j])
       {
        temp=A[j+1];
        A[j+1]=A[j];
        A[j]=temp;
       }
     }
   }
}
```

Dado que tenemos dos ciclos de n y $n-1$ iteraciones es fácil ver que el orden de este algoritmo es $O(n^2)$. Su nombre es debido a que cada elemento del arreglo A sube "burbujeando" hasta su lugar. Este algoritmo tiene la ventaja de que ocupa poca memoria y no requiere mucho código. Sin embargo, es demasiado lento para conjuntos grandes de datos.

El método de la burbuja puede mejorarse si "agitamos" de tal forma que no solo los elementos más grandes del arreglo suban, sino también los más pequeños desciendan.

```
void agitar_burbuja(int *A, int A)
{
```

```
 int cambio, ab, auf;
 int izquierda=1;
 int derecha=n;
 int listo=derecha;
  do
   {
    cambio=0;
     for(ab=derecha; ab>=izquierda; ab--)
      if(A[ab]<A[ab-1])
       {
        cambio=1; listo=ab;
        temp=A[ab-1]
        A[ab-1]=A[ab]; A[ab]=temp;
       }
      izquierda=listo+1;
     for(auf=izquierda; auf<=derecha; auf++)
      if(A[auf]<A[auf-1])
       {
        cambio=1; listo=auf;
        temp=A[auf-1]
        A[auf-1]=A[auf]; A[auf]=temp;
       }
    derecha=listo-1;
   }while (cambio);
}
```

3. Inserción y selección

El ordenamiento por inserción funciona de la misma manera en la cual se ordena la mano de una baraja. Se empieza con la mano izquierda vacía y las cartas cara abajo en la mesa. Entonces tomamos una carta a la vez de la mesa y la insertamos en la posición correcta en la mano izquierda. Para encontrar la posición correcta de la carta, la comparamos con cada una de las cartas que ya se han colocado. El código para este algoritmo es:

```
void insercion(int *A, int n)
{
 int j, i, llave;
 for(j=1; j<n; j++)
  {
   llave=A[j]; i=j-1;
   while(i>0&&A[i]>llave)
    {
     A[i+1]=A[i];
     i=i-1;
    }
```

```
   A[i+1]=llave;
  }
}
```

En este caso hacemos $\sum_{i=1}^{n} i = \frac{1}{2}n(n-1)$ comparaciones en el peor de los casos. Este es un polinomio cuadrático, por lo que el orden del algoritmo es $O(n^2)$.

Otra variante es la llamada *selección*. En este caso encontramos primero el elemento más pequeño del conjunto y lo ponemos en la primera posición; a continuación se halla el segundo más pequeño y se pone en segundo lugar, y así sucesivamente. Por supuesto, el conjunto de datos con el que se trabaja se reduce cada vez, pues ya no hay que considerar los elementos ya ordenados. El algoritmo sería el siguiente:

```
void seleccion(int *A, int n)
{
 int j, i, llave;
 for(i=0; i<n; i++)
  for(j=n-1; j>i; j--)
   if (A[j-1]>A[j])
    {
     llave=A[j-1];
     A[j-1]=A[j];
     A[j]=llave;
    }
}
```

En este caso, para ordenar el elemento i-ésimo del arreglo se tienen que realizar $n-i-1$ comparaciones en el peor de los casos. Al sumar se tienen en total $\sum_{i=0}^{n-1}(n-i-1) = \sum_{i=1}^{n}(n-i)$ comparaciones, o lo que es lo mismo $\sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1)$ igual que del algoritmo anterior.

Otra manera de insertar es a través de un árbol: se coloca el primer elemento del arreglo en la raíz, y se agrega el siguiente elemento a la izquierda si es menor que la raíz o a la derecha en caso contrario. Continuando recursivamente, se obtiene un árbol que al recorrerse en orden devuelve el conjunto ordenado de manera creciente. Se reproduce el procedimiento principal que construye el árbol.

```
void inserta(nodoapt *raiz, int num)
{
  void agregader(nodoapt *, int);
  void agregaizq(nodoapt *, int);
  void inserta(nodoapt *, int);
  if((*raiz)->info>num)
   {
    if((*raiz)->izq==NULL)
     agregaizq(raiz, num);
```

```
    else
     inserta(&((*raiz)->izq), num);
   }
  else
   {
    if((*raiz)->der==NULL)
     agregader(raiz, num);
    else
     inserta(&((*raiz)->der), num);
   }
}
```

El peor de los casos ocurre cuando el conjunto está ordenado, pues entonces realiza el mismo número de comparaciones que los algoritmos anteriores, es decir, es de orden $O(n^2)$. Sin embargo, si en el arreglo original los datos están organizados de tal manera que la mitad de los números anteriores a uno dado, a, en el arreglo sean menores que a y la otra mitad mayores que a, resultan árboles más balanceados. La profundidad p del árbol binario resultante no es inferior a $\log_2(n+1)-1$, y el número de nodos en cualquier nivel, con excepción quizá del último, es 2^l. El número de comparaciones necesarias para llegar al nivel $l \neq 0$ es $l+1$. Entonces el número de comparaciones está entre $p + \sum_{l=1}^{p-1} 2^l(l+1)$ y $\sum_{l=1}^{p} 2^l(l+1)$, sumas que nos indican que este algoritmo es de orden $O(n \log n)$, en lo que se considera el caso promedio del algoritmo, pues en general no se puede saber si un conjunto está ordenado o no.

4. Intercalación

La intercalación es otro procedimiento recursivo que además pertenece a los del paradigma *divide y vencerás*. Dicho paradigma se basa en tres pasos:

Dividir el problema en un cierto número de subproblemas.

Vencer cada problema acudiendo al principio de *divide y vencerás* nuevamente. Si el problema ya es lo suficientemente fácil, resolverlo de manera directa.

Combinar las soluciones parciales en la solución global.

En el caso de la intercalación esto ocurre de la manera siguiente:

Dividir: Partir el arreglo de n elementos en dos mitades.

Vencer: Ordenar los dos subarreglos con ordenamiento por intercalación.

Combinar: Intercalar los dos subarreglos en forma ordenada para producir el arreglo final ordenado.

El proceso debe llegar siempre a subarreglos de un elemento, pues éstos ya estan ordenados. El código será el siguiente.

```
void intercala (int *A, int l, int r)
{
 int m;
 void intercala(int *, int, int);
 void combina(int *, int, int, int);
 if (l<r)
  {
   m=(l+r)/2;
   intercala(A, l, m);
   intercala(A, m+1, r);
   combina(A,l,m,r);
  }
}

void combina(int *A, int l, int m, int r)
{
 int i, j, k;
 for(i=l, j=m, k=l; k<r; k++)
  if ((j>=r)||((i<m)&&(A[i]<=A[j])))
   A[k]=A[i++];
  else
   A[k]=A[j++];
}
```

Para los algoritmos divide y vencerás el analisis de orden nos indica que, de alcanzar la etapa "más simple" para ciertos c datos en cada recursión, sus complejidades serán $O(1)$. Para el orden O_T del algoritmo global supongamos O_D es el orden de la división en k partes más pequeñas y O_C el orden de la reconstitución de la solución total. Entonces

$$O_T(n) = \begin{cases} O(1) & O(1) \text{ si } n \leq c. \\ kO_T(\frac{n}{k}) + O_D(n) + O_C(n) & \text{en cualquier otro caso.} \end{cases} \quad (1)$$

Para el caso de la intercalación tenemos:

$$O_T(n) = \begin{cases} O(1) & \text{si } n = 1. \\ 2O_T(\frac{n}{2}) + O(1) & \text{si } n > 1. \end{cases}$$

Lo cual se reduce a que $O_T = O(n \log n)$.

5. Ordenación rápida

Llamado también Quicksort (como si fuera una marca registrada o algo, no veo la necesidad de no traducirlo), es un algoritmo introducido por C. A. R. Hoare. Goza de gran popularidad por su orden logarítmico en promedio y su superioridad frente a la intercalación en el sentido de que no hay necesidad de

recombinar para obtener el resultado final. Es del tipo divide y vencerás en este modo:

Dividir: Partir el arreglo de dos partes A y A' tales que para $a \in A$, $a' \in A'$, se tiene que $a \leq a'$.

Vencer: Ordernar los dos subarreglos con ordenación rápida.

Combinar: No se necesita combinar dada la forma en la que se hizo la división..

Se implementa así:

```
void ordena_rapido(int *A, int l, int r)
{
 int pivote;
 int partir(int *, int, int, int);
 if (l<r)
  {
   pivote=partir(A, l, r, r);
   ordena_rapido(A, l, pivote-1);
   ordena_rapido(A, pivote+1, r);
  }
 else
  {
   ordena_rapido(A, pivote+1, l);
   ordena_rapido(A, l, pivote-1);
  }
}

int partir(int *A, int l, int r, int pivote)
{
 void cambiar(int &, int &);
 int i=l-1;
 int j=r;
 int pivote=r;
 cambiar(A[pivote], A[r]);
 while(i<j)
  {
   do i++; while ((i<j)&&(A[i]<A[pivote]));
   do j--; while ((j>i)&&(A[j]>A[pivote]));
   if (i>=j)
    cambiar(A[i],A[pivote]);
   else
    cambiar(A[i],A[j]);
  }
 return i;
}
```

```
void cambiar(int &a, int &b)
{
 int temp=a; a=b; b=temp;
}
```

La función `partir` funciona de este modo, después de colocar un índice del arreglo abajo `i` y otro arriba `j` y considerando al `pivote`:

1. Incrementar en una posición el indice inferior mientras sea inferior al indice superior y al pivote.

2. Decrementar en una posición el indice superior mientras sea superior al indice inferior y al pivote.

3. Si el indice inferior es superior al indice superior, entonces intercambiamos el elemento inferior con el pivote, en caso contrario intercambiamos el elemento inferior y el superior.

En el peor de los casos, en el cual la partición nos devuelve el conjunto menos su último elemento y su último elemento como partición, según (1) tenemos que $O_T(n) = O_T(n-1) + O(n)$, y así $O_T(n) = \sum_{k=1}^{n} O(k) = O(\sum_{k=1}^{n} k) = O(n^2)$, recordando lo dicho en la introducción sobre la aditividad de O.

En los casos regulares tenemos, al igual que en la intercalación, que $O_T(n) = O(n \log n)$.

6. Montículo

La ordenación por montículo se sirve de la estructura del mismo nombre y es de orden logarítmico. Es un arreglo que se comporta como un arbol binario completo bajo las relaciones $padre(i) = \lfloor i/2 \rfloor$, $hijo\_izquierdo(i) = 2i$, $hijo\_derecho(i) = 2i - 1$, y que además tiene la *propiedad de montículo*, a saber, que $A[padre(i)] \geq A[i]$ en el caso de un montículo ascendente, y viceversa si es un montículo descendente. En otras palabras, cada nodo padre es mayor que sus hijos. Es fácil ver que en cada caso que en la raíz del montículo está el elemento más grande del arreglo y que cada camino que conecta a las hojas con la raíz es una sucesión ordenada.

El amontonamiento en una subrutina importante para manipular los montículos. Sus entradas son un arreglo A y un índice i del arreglo. Al ejecutarse, se asume que $hijo\_izquierdo(i)$ e $hijo\_derecho(i)$ son montículos. Entonces el amontonamiento reconstruye el submontículo con raíz en i en caso de que no sea un montículo.

Con el amontonamiento podemos convertir un arreglo en un montículo. Puesto que los elementos $A[\lfloor n/2 \rfloor + 1], A[n]$ son hojas del árbol, ya son montículos y son punto de partida, desde donde amontonamos a todo A.

Para ordenar, tomamos la raíz del montículo, la movemos a la última posición, reamontonamos el arreglo que queda después de remover el último elemento

y repetimos el procedimiento. Presentamos ahora el algoritmo de ordenación por montículo.

```
void monticulo(int *A, int len)
{
 void cambiar(int &, int &);
 void amontonar(int *, int, int);
 int i; int l;
 for (i=len-1; i>=0; i--)
  amontonar(A, len, i);
 for (l=len-1; l>=1; l--)
  {
   cambiar(A[0], A[l]);
   amontonar(A,l,0);
  }
}

void amontonar(int *A, int len, int r)
{
 int i=r;
 int j=2*r+1;
 while (j<len)
  {
   if((j+1<len)&&(A[j+1]>A[j])) j++;
   if(A[j]>A[i])
    {
     cambiar(A[i],A[j]);
     i=j;
     j=2*j+1;
    }
   else break;
  }
}

void cambiar(int &a, int &b)
{
 int temp=a; a=b; b=temp;
}
```

La complejidad de la primera parte del algoritmo es $O(n)$, y la del amontonamiento, que obedece a $O_T(n) \leq O_T(2n/3) + O(1) = O(\log n)$. Luego, el orden del montículo es $O(n \log n)$.

7. Conclusiones

El tema de las ordenaciones es muy interesante desde el punto de vista matemático y computacional. Tiene su aplicación en la vida diaria cuando se tienen que ordenar cosas como fichas de un directorio personal, publicaciones periódicas por fecha, o páginas fotocopiadas que están en desorden. Sin duda esto adquiere mayor importancia ahora que las personas manejamos volúmenes de información cada vez más grandes dada la revolución en las telecomunicaciones.

Referencias

[1] CORMEN, Thomas et al. *Introduction to algorithms*. MIT Press-McGraw-Hill, USA, 1996.

[2] TENENBAUM, Aaron et al. *Estructuras de datos en C*. Prentice-Hall Hispanoamericana. México, 1993.

[3] HEUN, Volker. *Grundlangen Algorithmen*. Vieweg & Sohn. Alemania, 2001.

[4] SOLYMOSI, Andreas. *Grundkurs Algorithmen und Datenstrukturen*. Vieweg & Sohn. Alemania, 2001.