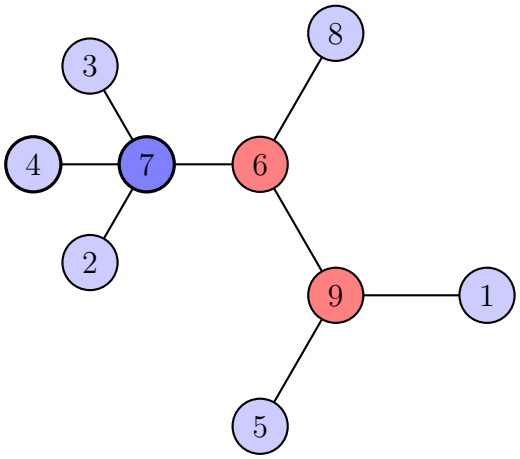
Problemas y Algoritmos



9 7 7 7 9 6 6

Por Luis E. Vargas Azcona
Algunas imagenes por Roberto López

Acuerdo de Licencia

Capítulo 18

¿Mas rápido que O(N log N)?

Ya se dijo varias veces en este libro que los algoritmos de ordenamiento mas rápidos son de $O(N\ log\ N)$; pero no hemos dado la prueba de ello y además es posible romper esta restriccion si nos restringimos a ordenar números enteros en lugar de ordenar cualquier conjunto con un órden total.

El siguiente algoritmo es importante conocerlo, las últimas dos secciones de este capítulo resultan interesantes para aquellos que gustan de conocer algoritmos pero no presentan utilidad alguna en los concursos de programación y tampoco tienen muchas aplicaciones así que esas dos secciones son opcionales.

18.1. Count-Sort

Consideremos que tenemos una sucesión finita de N enteros entre 0 y M, la cual denotaremos por $S = (s_1, s_2, ..., s_N)$.

Existe una manera relativamente sencilla de ordenar estos enteros que hemos pasado por alto hasta ahora: se trata de crear un arreglo `Cont` de tamaño M, tal que `Cont[k]` indique cuántas veces aparece k en la secuencia S.

La forma de crear este arreglo es mucho mas sencilla que los algoritmos de mezcla y los de ordenamiento usando árboles. Lo único que hay que hacer es inicializar el arreglo `Cont` en 0, recorrer S e incrementar el contador.

Una vez creado el arreglo `Cont` con las características deseadas, obtener el arreglo `S` ordenado se vuelve trivial:

```
1  void countSort(int S[], int N, int M){
2    int i, k;
3    for(i=0;i<=M;i++){
```

```
4       Cont[i]=0;
5     }
6     for(i=0;i<N;i++){
7       Cont[S[i]]++;
8     }
9     for(i=0,k=0;i<=M;i++){
10      while(Cont[i]>0){
11        S[k]=Cont[i];
12        Cont[i]--;
13        k++;
14      }
15    }
16  }
```

Este algoritmo puede aparentar ser el sueño mas maravilloso imaginado entre los algoritmos de ordenamiento: muy sencillo de implementar y funciona en tiempo lineal.

Pero las cosas no son tan maravillosas como parecen: en primer lugar su complejidad no es de órden $O(N)$ sino de órden $O(N+M)$, lo cual significa que si M es grande(por ejemplo $M = 2^31 - 1$) este algoritmo puede resultar ser mucho mas lento que todos los demás que hemos visto(¡incluso mas lento que burbuja!) y en segundo lugar sólamente sirve para ordenar números enteros no negativos, lo cual es muy diferente a poder órdenar cualquier conjunto con un órden total.

Este algoritmo puede ser adaptado para ordenar también números negativos sumando una constante a todo el arreglo y al final restándolo, también se pueden ordenar números racionales, pero para eso es necesario multiplicar todos los datos por una constante ¡e incluso multiplicar M por esa constante!.

Como conclusión, este algoritmo sólo es útil en casos muy específicos.

18.2. *Bucket-Sort

Ya vimos que el Count-Sort tiene la terrible desventaja de que el tamaño de M importa demasiado en el rendimiento, esto se puede corregir de una manera muy sencilla:

Hacemos count-sort con cada dígito, es decir, si c es el número de dígitos del mayor número de la secuencia entonces creamos 10 secuencias, agregamos a la primera todos aquellos números cuyo c-ésimo dígito(de derecha a izquierda) sea 0, a la segunda todos aquellos números cuyo c-ésimo dígito sea 1, y así sucesivamente.

Una vez separada la secuencia en 10 secuencias, es posible proceder recursivamente y al final simplemente concatenar las secuencias ordenadas.

Como se puede apreciar, este algoritmo también está pensado para ordenar únicamente números enteros no negativos, pero puede ser extendido para ordenar otras cosas.

Por ejemplo, se pueden ordenar números con expansión decimal finita siguiendo el mismo principio, también se pueden representar enteros simulando el digno con un dígito extra; otra cosa que se puede hacer es representar vectores ordenando primero los dígitos de una componente y luego los de la otra.

En resúmen: este algoritmo puede ordenar todo aquello cuyo órden total sea representado a través de números enteros no negativos. Cualquier conjunto finito con un órden total puede ser representado así, pero no siempre resulta práctico(es posible que para saber cómo representar los elementos sea necesario primero ordenarlos).

Otra cosa que hay que hacer notar, es que el algoritmo se ilustró con base 10 pero funciona con cualquier base, debido a que los procesadores trabajan mejor con potencias bits, es preferible usar una base que sea una potencia de 2.

La complejidad de este algoritmo es $O(N\ log\ M)$, asi que como se puede apreciar, de manera parecida a count-sort, este algoritmo sólo resulta mas rápido que los tradicionales cuando M es lo suficientemente pequeño.

Un detalle interesante es que si la base de numeración usada es igual a M entonces obtendremos el count-sort.

Debido a que este algoritmo puede presentar tantas implementaciones distintas según sea el caso, aquí sólo se mostrará una que ordena enteros y funciona usando base 2, por lo que sólo divide la secuencia original en 2 secuencias. Se asume que se tiene una función llamada `BIT(a, b)` la cual devuelve el $b-$ésimo bit de a.

```
1  void ordena(int S[], int inicio, int fin, int bits){
2          int i, k, m;
3          if(inicio>fin)
4                  return;
5          for(i=inicio;BIT(S[i], bits)==0 && i<=fin;i++){
               }
6          for(k=i+1;k<=fin;i++){
7                  for( ;BIT(S[k], bits)!=0 && k<=fin;k++)
                       { }
8                  if(k<=fin)
9                          intercambia(S[k], S[i]);
```

```
10              }
11              i=min(i, fin);
12              while(BIT(S[i], bits)!=0 && i>=inicio)
13                      i--;
14              if(bits>0){
15                      ordena(S, inicio, i, bits-1);
16                      ordena(S, i+1, fin, bits-1);
17              }
18      }
```

18.3. *La Inexistencia de Mejores Algoritmos

A pesar de que esta sección está marcada con (*) se advierte nuevamente que el lector puede optar por saltarse esta sección y se asegura que no se volverá a utilizar en el libro.

De hecho a lo largo del libro las únicas matemáticas que se dan por conocidas han sido matemáticas muy elementales, pero este capítulo es diferente, pues usaremos una ideantidad llamada la **fórmula de Stirling**.

La demostración de esta identidad requiere de conocimientos de cálculo y lo cual se aleja de los propósitos de este libro. Si lector quiere conocer la demostración de la fórmula puede consultarla el apéndice de bibliografía recomendada.

Teorema 34 (Fórmula de Stirling)**.** *Se cumple la siguiente igüaldad.*

$$log(n!) = n\ log(n) - n + f(n) \tag{18.1}$$

Donde n es un entero positivo y f es de órden $O(log(n))$

Para obtener una cota inferior respecto a la complejidad de un algoritmo de ordenamiento, recordemos qué es lo que tenemos permitido hacer.

Cuando decimos “algoritmo de ordenamiento“ normalmente nos resferimos a un ordenamiento sobre un elementos que tienen un órden total; es decir lo único que podemos hacer es comparar pares de elementos, ya que en un órden total sólo estamos seguros que los elementos *saben* cómo compararse.

También tenemos n elementos, por simplicidad asumiremos que dichos elementos son todos distintos.

Ordenar dichos elementos consiste en encontrar una permutación de ellos que cumpla con ciertas características, dicha permutación es única; y con cada comparación que se realiza, nos vamos haciendo la idea de cuál puede ser la permutación correcta.

De una manera mas precisa, inicialmente hay $n!$ permutaciones que podrían ser la correcta y podemos hacer $\frac{(n)(n-1)}{2}$ posibles comparaciones; cada vez que realicemos una comparación obtendremos un 0 o un 1, y la intención es que la cantidad de permutaciones que podrían ser la correcta disminuya.

Vamos a imaginar un árbol binario, en el cual a cada hoja corresponde a una permutación única y a cada nodo v corresponde a un conjunto de permutaciones posibles $C(v)$ de tal manera que si los hijos de v son a y b entonces $C(v) = C(a) \cup C(b)$ y además $a \cap b = \emptyset$.

Este árbol representa un algoritmo, cada nodo representa un estado posible del algoritmo, de tal manera que la raiz r representa el estado inicial y $C(r)$ es el conjunto de todas las permutaciones de los n elementos.

Como el algoritmo está bien definido entonces no hay duda que la primera comparación es siempre la misma, esa comparación puede regresar 0 o 1, el hijo izquierdo representa el estado del algoritmo si la primera comparación regresa 0 y el hijo derecho representa el estado del algoritmo si la primera comparación regresa 1.

De tal manera que si a y b son los hijos izquierdo y derecho de la raiz entonces $C(a)$ y $C(b)$ representan los conjuntos de posibles permutaciones válidas si la primera comparación regresa 0 y si regresa 1 respectivamente.

De manera análoga todas las comparaciones realizadas(y los valores devueltos) por el algoritmo en un estado quedan determinadas por el camino a la raíz del nodo que representa dicho estado; los hijos izquierdo y derecho representarán lo que sucede si la siguiente comparación regresa 0 y si regresa 1 respectivamente.

Una vez convencidos de que cada árbol puede representar de manera única todas las posibles comparaciones que realiza un algoritmo de ordenamiento con una entrada de tamaño n, procederemos a medir la complejidad en cuanto a número de comparaciones.

Descender un nivel en el árbol equivale a realizar una comparación, de esta manera al alcanzar una hoja, el número de comparaciones realizadas será su distancia a la raíz. Como se explico antes, *somos pesimistas*, así que vamos a medir la complejidad del algoritmo con respecto a la hoja mas lejana a la raíz, es decir, respecto a la altura del árbol.

El número de hojas del árbol binario es $n!$(una hoja por cada permutación), y bien sabemos que un árbol binario con $n!$ hojas tiene altura de al menos $log_2(n!)$, por lo tanto por la fórmula de Stirling la altura del árbol es de órden $O(nlogn)$.

De este resultado obtenemos el siguiente teorema:

Teorema 35. *Cualquier algoritmo de ordenamiento(sobre un órden total) tiene complejidad de al menos* $O(n\ log\ n)$.