Introducción a la Algoritmia (IV): Coste de ordenar

aprende.olimpiada-informatica.org/algoritmia-introduccion-4-coste-ordenar

Enviado por fmoreno el Vie, 20/09/2019 - 16:09

Continuamos con el último ejemplo del manual anterior: ordenar un vector. Presentamos una solución en $\mathcal{O}(n \log n)$ conocida como ordenación por fusión o *Merge Sort*. La solución utiliza recursividad, así que se recomienda a los lectores que no estén familiarizados con este concepto que lean el manual de recursividad antes de comenzar.

```
void mergesort(int i, int j) {
     if (j == i) return;

    mergesort(i, (i+j)/2);
    mergesort((i+j)/2+1, j);

    int a = i;
    int b = (i+j)/2+1;
    vector<int> aux (j-i+1);

    for (int k = 0; k <= j-i; ++k) {
          if(a <= (i+j)/2 and (b > j or v[a] < v[b])) {
                aux[k] = v[a];
                ++a;
          }
          else {
                aux[k] = v[b];
                ++b;
          }
    }

    for (int k = 0; k <= j-i; ++k) v[i+k] = aux[k];

}
```

Vamos a explicar el anterior código. Tenemos v guardado como variable global y la función mergesort lo que hace es ordenar el intervalo [i, j] del vector: así, si quisiéramos ordenar todo el vector, llamaríamos mergesort(0, n-1). Lo primero que hace la función es, si el intervalo tiene más de un elemento, llama recursivamente a la propia función para tener ordenados la primera y la segunda mitad del vector. Una vez tiene las dos mitades ordenadas, lo que hace es unir (*merge*) la información de las dos mitades para obtener el vector completo ordenado. Esto lo hace de la siguiente manera: primero compara los dos primeros elementos de cada mitad y el más pequeño de ellos será el primero del vector ordenado. Después compara el segundo de la mitad que tenía el mínimo con el primero de la otra mitad: en general, se mantienen dos índices a y b y en cada iteración se compara el a-ésimo de la primera mitad con el b-ésimo de la segunda, el menor de los dos es el

siguiente elemento del vector ordenado y el índice correspondiente se aumenta en uno. Dados dos vectores ordenados, este algoritmo produce el vector ordenado que contiene los elementos de los dos vectores, ya que el siguiente elemento en el vector total siempre tiene que ser uno de los dos primeros elementos de los vectores originales que aún no se ha añadido.

Las llamadas recursivas de este algoritmo acaban en el caso base de intervalos de un elemento, que ya están ordenados. Vamos ahora a analizar la complejidad de este algoritmo. Cuando la función mergesort se aplica sobre un intervalo de longitud $L = j-i+1$, la parte no recursiva de la función tiene una complejidad $\mathcal{O}(L)$. (Tanto la creación de un vector de L elementos, como el bucle para unir las dos mitades y el segundo bucle para copiar el contenido al vector original son $\mathcal{O}(L)$). La clave para calcular la complejidad teniendo en cuenta las llamadas recursivas es que, en cada nivel de recursividad, las longitudes de los intervalos suman n (la longitud del vector original). Por ejemplo, en el segundo nivel de recursividad hay dos llamadas a la función, cada una con un intervalo de longitud $\frac{n}{2}$, así que el coste temporal de su parte no recursiva será $\mathcal{O}(n)$. En el tercer nivel hay 4 llamadas a intervalos de logitud $\frac{n}{4}$, así que el coste sigue siendo n. En el último nivel hay n llamadas a intervalos de un elemento que tienen coste constante, $\mathcal{O}(1)$, así que el coste total del nivel es $\mathcal{O}(n)$. Hay un total de $\mathcal{O}(\log n)$ niveles de recursividad; a coste $\mathcal{O}(n)$ por nivel, en total tenemos una complejidad de $\mathcal{O}(n \log n)$.

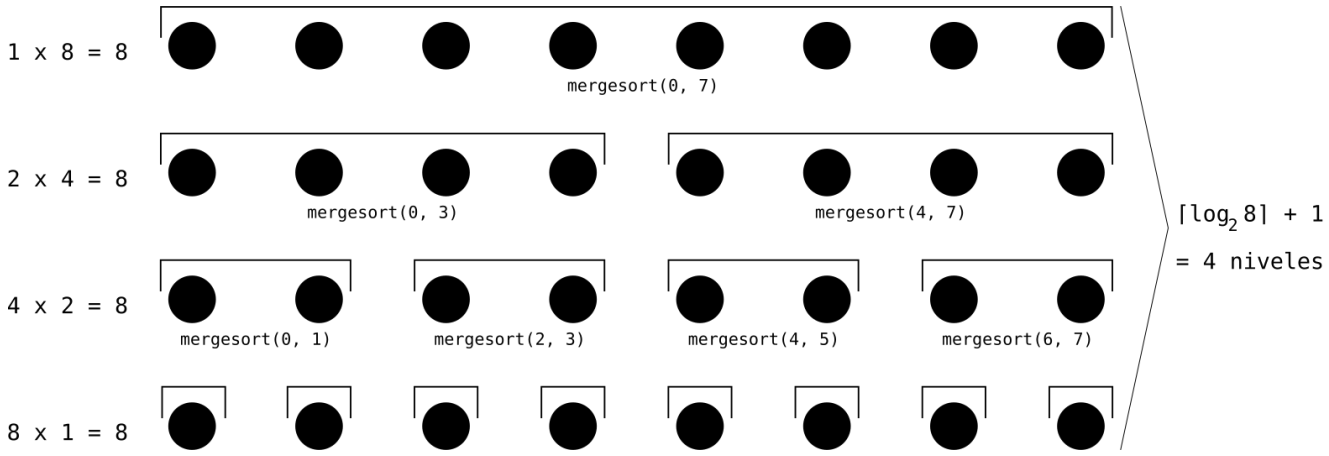

Ilustración para n = 8

Notemos que, al igual que en la estrategia de búsqueda dicotómica de la que se hablaba en el primer manual de esta serie, la idea principal del algoritmo Merge Sort es ir dividiendo el problema en partes más pequeñas. Esta idea, a la que a veces se le refiere con el nombre de *Divide&Conquer*, aparece en muchos algoritmos y es importante tenerla en mente.

Hemos visto que el problema de ordenar un vector se puede resolver de forma más eficiente que con el sencillo algoritmo $\mathcal{O}(n^2)$ que habíamos explicado en el anterior manual. Una pregunta natural que hacerse es: ¿cuál es la forma más eficiente de hacerlo?

Y, como dijimos en el anterior manual, la respuesta es que, en términos de complejidad asintótica, no se puede bajar de $n \log n$. Aquí hay un razonamiento sencillo que justifica este resultado:

Proposición: Todos los algoritmos que ordenan un vector basándose en comparaciones realizan $\Omega(n \log n)$ operaciones de comparación en el peor caso. (La notación Ω es como la notación O pero indica una cota inferior en lugar de superior)

Pensemos en el algoritmo como un proceso que realiza una cierta cantidad de comparaciones y, una vez las ha realizado, retorna una permutación del vector original que corresponde al vector ordenado. Por ejemplo, si al algoritmo le damos el vector 1, 7, 4, puede comparar los dos primeros elementos, obtener que el primero es menor, luego comparar el segundo y el tercero, obtener que el tercero es menor y comparar el primero y el tercero y obtener que el primero es menor, y concluir que el vector ordenado se obtiene permutando el vector original según la permutación 1, 3, 2 (es decir, el primer elemento va en la primera posición, el segundo en la tercera y el tercero en la segunda). Si codificamos los resultados de cada comparación con un 0 o un 1 (dependiendo de qué elemento es menor), la ejecución del algoritmo se podría escribir en cierto modo como [0, 1, 0]->[1, 3, 2]. Las comparaciones que hace el algoritmo sólo dependen de los resultados de comparaciones anteriores; por tanto, a cada cadena de ceros y unos le corresponde como máximo una única permutación. Como hay $n!$ permutaciones posibles (y el algoritmo tiene que ser capaz de reconocer todas ellas) y el número de cadenas binarias de longitud como máximo m es $2^{m+1}-1$, habrá alguna permutación que requiera una cadena de longitud $\log n!$ para llegar a ella. Es decir, el algoritmo tendrá que realizar $\log n!$ comparaciones. La fórmula de Stirling nos dice que para n grande $\log n!$ crece como $n \log n$, por tanto la complejidad del algoritmo es al menos $n \log n$.

Bien. Ahora veamos el siguiente código para ordenar un vector v de n enteros no negativos que son todos menores que una cota (c) dada. (Este algoritmo se conoce como *Counting Sort*)

```
vector<int> cuenta = vector<int>(c, 0);
for (int i=0; i < n; ++i) {
    ++cuenta[v[i]];
}
int ind = 0;
for (int j=0; j < c; ++j) {
    while(cuenta[j] > 0) {
       --cuenta[j];
      v[ind] = j;
      ++ind;
    }
}
```

El programa cuenta cuántos elementos hay de cada número, y después los pone en orden iterando por todos los valores en orden creciente y añadiendo al vector tantos elementos para cada número como números había. Las iteraciones totales que hace el bucle while suman n, así que tenemos un algoritmo de ordenación de complejidad $\mathcal{O}(n)$...

...en realidad es $\mathcal{O}(n + c)$, porque no hemos de olvidar el coste de crear el vector cuenta y de iterar por él. Pero sigue siendo mejor que $n \log n$ en ciertos contextos: por ejemplo, si en un problema de olimpiada queremos ordenar un vector de 10^6 enteros no negativos menores que 10^6, el Counting Sort tardaría muy poco, mientras que la mayoría de algoritmos de ordenamiento con complejidad $n \log n$ superarían un segundo de ejecución. ¿Qué ha pasado? ¿Contradice esto la proposición que hemos razonado que era cierta?

La hipótesis de "basándose en comparaciones" es importante. En nuestro razonamiento hemos asumido que la única forma que tenemos de obtener información acerca del orden de los elementos del vector es haciendo comparaciones entre ellos. Asumir esto es útil en cierto sentido, porque por ejemplo los algoritmos de Selection Sort y Merge Sort (que únicamente hacen comparaciones) no nos sirven sólo para ordenar vectores de números enteros, sino para ordenar vectores de cualquier objeto abstracto que admita una operación de comparación. Pero los números enteros (no negativos y dentro de un rango, en este caso) tienen propiedades adicionales que podemos aprovechar para hacer algoritmos más eficientes. En concreto, el hecho de que podamos indizar un vector con números enteros y acceder a su contenido en tiempo constante es lo que hace que el Counting Sort tenga complejidad lineal. El problema del ordenamiento es un problema muy estudiado en el área de la algoritmia y hay muchas otras soluciones, tanto como para ordenar cualquier tipo de objeto comparable (con complejidad $n \log n$ o superior) como para ordenar objetos específicos aprovechando sus propiedades. Recomendamos leer la fascinante página de Wikipedia con la lista de algoritmos de ordenación.

Con este último ejemplo concluimos la serie de manuales introductorios. Aunque en la OIE no es necesario demostrar matemáticamente ni la corrección ni la complejidad temporal de la solución que se envía, los razonamientos del estilo del que se ha hecho en este manual pueden ser de utilidad para saber en qué aspectos son o no son mejorables los programas. En los siguientes manuales daremos ideas y herramientas más específicas para optimizar la complejidad de los algoritmos.

Añadir nuevo comentario

Acerca de formatos de texto

Texto sin formato

- No se permiten etiquetas HTML.
- Saltos automáticos de líneas y de párrafos.

- Las direcciones de correos electrónicos y páginas web se convierten en enlaces automáticamente.