Ordenar eficientemente

Introducción

Estamos preparados para hablar de maneras eficientes de ordenar listas de números (o de cualquier tipo de objetos sobre los que haya un orden parcial). En esta entrada introduciremos tres ideas importantes y un poco más avanzadas para el desarrollo de algoritmos: el uso de buenas estructuras de datos, el uso de recursión y el uso de componentes aleatorios.

TreeSort: SelectionSort + una buena estructura

El algoritmo `InsertionSort` nos permite ordenar una `Lista` de elementos a partir de una idea muy simple: ir eligiendo elementos mínimos. Como recordatorio, comenzamos con una `Lista` L y una `Lista` vacía M. Buscamos el menor elemento L, y lo pasamos al final de M. Luego, repetimos esto hasta agotar los elementos de L. Al final, M tendrá los elementos de L ordenados. Es difícil encontrar una forma de ordenar más sencilla que esta.

Sin embargo, esta idea tan simple tiene una desventaja: no alcanza la mejor complejidad asintótica de ordenamiento. Si la implementamos con estructuras de datos básicas, el algoritmo corre en tiempo cuadrático. Por ejemplo, en una lista enlazada el encontrar el mínimo toma tiempo $\Theta(n)$ y hay que hacer esto $\Theta(n)$ veces, de modo que por lo menos requerimos tiempo $\Theta(n^2)$.

Al utilizar una mejor estructura de datos podemos mejorar notablemente el tiempo de ejecución. Si tenemos una lista con n objetos, entonces podemos pasarla a un árbol binario de búsqueda auto-balanceable en tiempo $O(n\log n)$ pues para cada uno de los n objetos debemos hacer una inserción que toma tiempo $O(\log n)$.

Ya que tenemos a todos los elementos en un árbol de búsqueda binario, podemos encontrar al mínimo en tiempo $O(\log n)$, ponerlo al final de una nueva lista y eliminarlo del árbol. Tras repetir esto n veces obtendremos una lista con todos los elementos originales, pero ahora en orden.

Pasar al árbol binario de búsqueda auto-balanceable toma tiempo $O(n\log n)$ y pasar todos los mínimos también toma tiempo $O(n\log n)$, de manera que en total esta forma de ordenar toma tiempo $O(n\log n)$.

MergeSort: Utilizando recursión

Otra forma de ordenar eficientemente es usando recursión. A grandes rasgos, cuando hacemos recursión tomamos una instancia grande de un problema, la dividimos en una o varias instancias pequeñas, y luego usamos esas soluciones para responder la instancia grande.

Ya hicimos esto anteriormente cuando hablamos de búsqueda binaria: para buscar un número en una colección ordenada lo que hacemos es comparar el número con el que está a la mitad de la colección. Esto nos permite reducir el problema de buscar o bien a la primer mitad, o bien a la segunda mitad. Es decir, de un problema de tamaño n pasamos a un problema de tamaño aproximadamente $n/2$.

Podemos usar esta misma idea para ordenar una lista de números, aunque los detalles son un poco más elaborados. Los describimos a continuación.

Si tenemos una lista de únicamente un elemento, entonces ya está ordenada y no hay nada que hacer. Tomemos una lista $a_1,a_2,\ldots,a_n$ que queramos ordenar. Para ordenarla, haremos lo siguiente:

- La dividiremos en dos listas L_1 y L_2 aproximadamente con la mitad de elementos cada una.
- Ordenaremos por separado L_1 y L_2.

- Ordenaremos por separado L_1 y L_2.
- Luego combinaremos ambas listas ordenadas para obtener la ordenación de la lista original.

Llamemos $f(n)$ a la cantidad de pasos que toma el algoritmo.

Dependiendo de cómo esté implementada la lista, el primer punto puede tomar $O(n)$ (si es un arreglo) o $O(1)$ pasos (si es una lista enlazada). Pensemos que es $O(n)$. El segundo punto es parte recursiva, pues en vez de resolver una instancia de tamaño n, estamos resolviendo dos instancias de tamaño aproximadamente $n/2$.

En el tercer punto hay que ser cuidadosos al combinar las listas ordenadas de manera rápida. Esto lo podemos hacer como sigue. Si $c_1,c_2,\ldots,c_k$ y $b_1,b_2,\ldots,b_l$ son listas que ya están ordenadas por separado, entonces podemos comenzar leyendo los elementos c_i y los b_i de izquierda a derecha simultáneamente. Si $c_1<b_1$, ponemos primero a c_1 en nuestra lista final y nos movemos a la derecha en los c_i, para ahora comparar c_2 con b_1. Seguimos así sucesivamente, poniendo el elemento más chico de las comparaciones que vamos haciendo y moviéndonos a la derecha en esa lista. Al final habremos combinado los elementos en orden en tiempo $O(n)$.

Así, la cantidad de pasos que satisface esta forma de ordenar elementos satisface la recursión (asintótica)

$$f(n)=2f(n/2)+O(n).$$

Afirmamos que esto nos dice que $f(n)=O(n\log n)$. En efecto, supongamos que la parte $O(n)$ de la recursión satisface ser menor a cn para $n\geq n_0$ suficientemente grande. Tomemos $k_1=\frac{c}{\log 2}$ y k_2 a la cantidad de tiempo que nos toma ordenar una lista de tamaño hasta n_0 (que puede ser algo muy grande, pero como n_0 es constante, k_2 también lo es). Tomemos $k=\max(k_1,k_2)$.

Afirmamos que entonces $f(n)\leq k n\log n$. Esto puede ser probado inductivamente. Para $n\leq n_0$, esto es cierto pues $n\log n>1$ y k_2, de modo que $kn\log n> k_2$, así que esta cantidad de pasos es suficiente para ordenar cualquier lista de tamaño hasta n_0. Supongamos que el resultado es cierto para todos los valores hasta antes de un $n\geq n_0$. Veamos que también se vale para n.

Para ello usamos la recursión que tenemos y la hipótesis inductiva:

$$\begin{aligned} f(n)&\leq 2f(n/2) + cn \\ &\leq 2k \frac{n}{2} \log \frac{n}{2} + cn\\ &=kn(\log n - \log 2) + cn\\ &=kn\log n + (cn-kn\log 2)\\ &\leq kn\log n + (c-k_1\log 2)n\\ &=kn\log n. \end{aligned}$$

Esto muestra que se puede hacer el paso inductivo, y por lo tanto que se cumple lo que queremos. $\square$

Paréntesis: El teorema maestro

Hay una herramienta matemática muy general para saber cuánto tiempo se tardan los algoritmos recursivos. Supongamos que tenemos un algoritmo recursivo que procede de la siguiente manera:

```
problema(n):
    dividir el problema en a sub-problemas de tamaño b/n
    resolver problema para cada uno de ellos (recursión)
    combinar las respuestas
```

El teorema maestro ayuda a saber cuánto tiempo se tardan este tipo de algoritmos bajo ciertas hipótesis sobre a, b y el tiempo de dividir el problema en subproblemas y de combinar las respuestas.

Teorema. Supongamos que tenemos una función que satisface la siguiente ecuación recursiva:

$$T(n)=aT(n/b)+f(n).$$

en donde $a\geq 1$ y $b>1$ son constantes y $f(n)$ es una función positiva. Definamos $d=\log_b a=\log a / \log b$, al cual le llamaremos el **exponente crítico**. Entonces, tenemos los siguientes tres casos:

1. Si $f(n)=O(n^{d-\epsilon})$ para alguna constante $\epsilon>0$, entonces $T(n)=\Theta(n^d)$.

2. Si $f(n)=\Theta(n^d \log ^k n)$ para alguna $k\geq 0$, entonces $T(n)=\Theta(n^d\log^{k+1}n)$.
3. Si $f(n)=\Omega(n^{d+\epsilon})$ para alguna constante $\epsilon>0$ y además $f(n)$ satisface la **condición de regularidad** $af(n/b)<cf(n)$ para alguna constante $c<1$ y n suficientemente grande, entonces $T(n)=\Theta(f(n))$.

Más adelante veremos más ejemplos de cómo usar el teorema maestro y su demostración.

Tarea moral

Los siguientes problemas te ayudarán a practicar lo visto en esta entrada. Para resolverlos, necesitarás usar herramientas matemáticas, computacionales o ambas.

1. Aplica manualmente `MergeSort` y `TreeSort` a los siguientes números para asegurarte de lo que están haciendo:
   $$[76,35,76,23,65,32,12,56,87,54,21,11,63,12,78,22,12.]$$
2. Demuestra que `MergeSort` y `TreeSort` son algoritmos de ordenamiento correctos.
3. En `MergeSort`, ¿qué pasa si en vez de dividir en dos listas de tamaño aproximadamente $n/2$, dividimos en una de tamaño aproximadamente $n/3$ y otra de tamaño aproximadamente $2n/3$? En este caso, tendríamos la recursión asintótica
   $$[f(n)=f(n/3)+f(2n/3)+O(n).]$$
   ¿Cómo sería f asintóticamente?
4. Haz implementaciones de `TreeSort` y de `MergeSort` en Python.
5. Estudia otras propiedades de `TreeSort` y de `MergeSort` en términos de aspectos deseables que deben tener los algoritmos de ordenamiento. ¿Son estables? ¿Cuánto espacio usan asintóticamente?

Anterior
Diccionarios y árboles de búsqueda balanceados

Siguiente
Aplicaciones de ordenar

Por Leonardo Ignacio Martínez Sandoval