Esquema de Dividir y Vencer

Amalia Duch

Barcelona, marzo de 2006

Índice

1. Esquema general **1**

2. Búsqueda binaria (binary–search) **2**

3. Ordenación por fusión (merge–sort) **2**

4. Ordenación rápida (quick–sort) **4**

1. Esquema general

El esquema de **dividir y vencer** (también conocido como divide y vencerás) es una técnica para resolver problemas que consiste en dividir el problema original en subproblemas (de menor tamaño), resolver los subproblemas y finalmente combinar las soluciones de los subproblemas para dar una solución al problema original.

Si los subproblemas son similares al problema original entonces puede utilizarse el mismo procedimiento para resolver los subproblemas recursivamente.

En cada nivel del procedimiento este paradigma consta de los siguientes tres pasos:

1. **Dividir** el problema en subproblemas.
2. **Resolver** cada uno de los subproblemas.
3. **Combinar** las soluciones de los subproblemas para obtener la solución del problema original.

Para que esta técnica funcione deben cumplirse los dos requisitos siguientes:

1. El primero es que los subproblemas deben ser más simples y sencillos que el problema original.
2. El segundo es que después de un número finito de subdivisiones debe encontrarse un subproblema que pueda resolverse de manera directa.

2. Búsqueda binaria (binary–search)

Consideremos el problema de buscar un elemento x en un vector T ordenado crecientemente. Un algoritmo que solucione este problema debe retornar un 1 en caso de que x esté en T y un 0 en caso contrario (existen otras variantes como por ejemplo retornar un -1 en caso de que el lemento no esté en el vector o la posición en la que se encuentra en caso contrario).

Un algoritmo ya estudiado previamente para resolver este problema eficientemente es el de búsqueda binaria o búsqueda dicotómica, que representa un ejemplo sencillo del esquema de dividir y vencer. Este algoritmo consiste en buscar x o bien en la primera mitad del vector o bien en la segunda. A grandes rasgos, para determinar cuál de las dos búsquedas es la apropiada se compara x con el elemento central del vector de tal manera que si $x < T[1 + \lfloor \frac{n}{2} \rfloor]$ la búsqueda siguiente será en $T[1 \ldots \lfloor \frac{n}{2} \rfloor]$, en caso contrario será en $T[\lfloor \frac{n}{2} \rfloor \ldots n]$.

Código. Dado en clase.

Ejemplos. En cada iteración el programa anterior encuentra el elemento x o bien se auto llama recursivamente con una sublista que es como mucho la mitad de la anterior. Empezando este procedimiento con un vector de tamaño n no podemos dividir a la mitad la longitud del vector más de $\log_2 n$ veces, hasta obtener un vector de longitud 1. Esta última observación justifica intuitivamente el coste en tiempo de la búsqueda dicotómica.

Para obtener este coste formalmente supongamos, sin pérdida de generalidad, que el tamaño del vector es una potencia de 2, es decir $n = 2^m$ para algún entero no negativo m. Sea $T(n)$ el coste en tiempo de buscar un elemento x en un vector T de tamaño n. $T(n)$ puede expresarse mediante la recurrencia divisora siguiente:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ T(n/2) + \Theta(1) & \text{si } n > 1 \end{cases}$$

Aplicando el teorema maestro para recurrencias divisoras es fácil ver que $T(n) = \Theta(\log n)$.

Como ejercicio, puede ser interesante modificar el algoritmo anterior para el caso en el que se quieran contar todas las ocurrencias del elemento x en el vector y no solamente determinar si está en el vector o no.

3. Ordenación por fusión (merge–sort)

Dado un vector T de n elementos, el objetivo es retornar T ordenado crecientemente.

El algoritmo de ordenación por fusión (merge–sort) sigue de manera muy cercana el esquema de dividir y vencer. Intuitivamente procede de la siguiente manera:

1. **Divide** el vector T de n elementos en dos subvectores de aproximadamente $n/2$ elementos cada uno.

2. **Ordena** los dos subvectores recursivamente utilizando merge–sort.

3. **Fusiona** los dos subvectores ordenados para producir el vector ordenado que soluciona el problema original.

Es fácil observar que la base de la recursión es el caso en el que el vector a ordenar contiene un único elemento, ya que este vector ya está ordenado y por tanto no queda trabajo por hacer. Otra observación pertinente es el hecho de que la operación clave de merge–sort es la de fusionar los dos subvectores ya que es el momento en el que realmente se produce la ordenación.

Código. Mirar el código en las transparencias de la asignatura (`http://www.lsi.upc.edu/~ada`).

Ejemplos. Sea $T(n)$ el coste en tiempo de ordenar un vector T de n elementos utilizando merge–sort. Es fácil ver que $T(n)$ consta de dos costes distintos. El primero es el coste de dividir T en dos subvectores y ordenarlos utilizando merge–sort. El segundo es el coste de hacer la fusión de los dos subvectores. También es fácil ver que el coste de hacer esta fusión es lineal con respecto al tamaño de los subvectores. Por tanto, $T(n)$ puede expresarse mediante la recurrencia divisora siguiente:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

Aplicando el teorema maestro para recurrencias divisoras es fácil ver que $T(n) = \Theta(n \log n)$.

Si miramos el código de merge–sort es fácil observar que uno de sus inconvenientes es que la etapa de fusionar dos subvectores requiere de memoria adicional (función `merge` del código).

Otro inconveniente es que el trabajo que requieren las divisiones sucesivas de los subvectores es, en cierto sentido, innecesario ya que no se está realizando ningún trabajo de ordenación. Para solventar este incoveniente, puede utilizarse la variante de *abajo hacia arriba* de merge–sort. Esta variante simplemente se ahorra las etapas de división del vector haciendo fusiones desde el principio con los elementos del vector, de dos en dos, cuatro en cuatro y así sucesivamente. El código para su implementación puede consultarse también en las transparencias de la asignatura.

Cabe mencionar que a pesar de que el coste en tiempo en el caso peor de merge–sort es $\Theta(n \log n)$ y el coste en tiempo en el caso peor del algoritmo de ordenación por inserción (insertion–sort) es $\Theta(n^2)$, insertion–sort es más rápido que merge–sort para ordenar vectores pequeños ya que los factores constantes del coste de insertion–sort (escondidos dentro de la notación asintótica) son menores que los de merge–sort. Por tanto, tiene sentido utilizar insertion–sort dentro

de merge–sort cuando el tamaño del subproblema es suficientemente pequeño (vectores de 50 elementos o menos). El código para implementar esta adaptación del algoritmo puede encontrarse en las transparencias de la asignatura.

4. Ordenación rápida (quick–sort)

El algoritmo de ordenación rápida (quick–sort) fue propuesto en el año 1960 por C. A. R. Hoare, motivo por el cual también se le conoce como algoritmo de ordenación de Hoare. El coste en tiempo en el peor de los casos de quick–sort para ordenar un vector de n elementos es $\Theta(n^2)$. A pesar de que este coste en el caso peor es muy lento, en la práctica quick–sort es con frecuencia la mejor opción para ordenar pues es muy eficiente en promedio. Su coste en tiempo en este caso es $\Theta(n \log n)$, y los factores escondidos dentro de la notación asintótica son pequeños. También tiene la ventaja de ordenar *in situ* (es decir, los elementos a ordenar se reorganizan dentro del mismo vector, con sólo un número constante de elementos almacenado fuera del vector, en cualquier momento de la ejecución del algoritmo).

Al igual que merge–sort, quick–sort está basado en el esquema de dividir y vencer. Intuitivamente las etapas que sigue quick–sort para ordenar un vector T de n elementos son las siguientes:

1. **Divide** (reorganiza, particiona) el vector en dos subvectores no vacíos de tal manera que cada elemento del primer subvector es menor o igual que cada elemento del segundo. El cálculo del punto de corte para hacer esta separación forma parte de esta etapa de división del problema.

2. **Ordena** recursivamente utilizando quick–sort cada uno de los subvectores.

3. **Combina** las soluciones. Pero como cada subvector se ordena *in situ* realmente no se requiere ningún trabajo para combinar subvectores que ya están ordenados.

La idea de quick–sort es dedicarle mucho esfuerzo a la parte de dividir, ya que es la etapa en la que se realiza el trabajo de ordenación, y muy poco esfuerzo (ninguno) a la etapa de combinar. Esto contrasta con la idea de merge–sort en donde los esfuerzos se realizan en las etapas contrarias.

Usualmente se escoge al primer elemento del vector como *pivote* (punto de corte) para hacer la partición. De esta manera en el primer subvector quedan los elementos que son menores o iguales que el pivote y en el segundo subvector los elementos mayores o iguales al pivote.

Código. El código para implementar quick–sort y las variantes que se comentarán posteriormente puede consultarse en las transparencias de la asignatura (`http://www.lsi.upc.edu/~ada`).

Ejemplos. Sea $T(n)$ el coste en tiempo de ejecución de ordenar un vector T de n elementos utilizando quick–sort. Es fácil ver que $T(n)$ consta de dos costes distintos. El primero es el coste de hacer la partición de T en dos subvectores de tamaños k y $n-k$ respectivamente. El segundo es el coste de ordenar cada subvector utilizando quick–sort. Es fácil ver que el coste de hacer la partición es lineal con respecto al tamaño de los subvectores mientras que el segundo corresponde a las llamadas recursivas del algoritmo. Por tanto, $T(n)$ puede expresarse mediante la recurrencia siguiente:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ T(k) + T(n-k) + \Theta(n), 1 \leq k \leq n-1 & \text{si } n > 1 \end{cases}$$

La solución de la recurrencia anterior depende por tanto del valor de k.

Coste en tiempo en el caso mejor. El coste en este caso es $\Theta(n \log n)$ y corresponde a remplazar k por $n/2$ en la recurrencia anterior. En este caso estaríamos dividiendo el vector en dos subvectores del mismo tamaño e intuitivamente es la forma más rápida de ordenar el vector.

Coste en tiempo en el caso promedio. El coste en este caso es $\Theta(n \log n)$. Para calcularlo se ha de suponer una distribución de probabilidad de las entradas. Es común suponer en este caso que todas las permutaciones del vector de entrada son equiprobables y que por tanto todos los elementos tienen igual probabilidad $(1/n)$ de ser el pivote.

Coste en tiempo en el caso peor. El caso peor de quick–sort sucede cuando el vector de entrada está ordenado creciente o decrecientemente. En este caso se divide el vector en un subvector de tamaño 1 y otro de tamaño $n-1$. Intuitivamente parece que que no estamos reduciendo suficientemente el tamaño del problema pues hemos de ordenar un vector casi del mismo tamaño del original. Reemplazando este valor de k $(k = 1)$ en la recurrencia anterior puede verse que el coste en este caso es $\Theta(n^2)$.

Variantes de quick–sort. Existen varias variantes de quick–sort para evitar el caso peor. La más simple de todas es elegir como pivote a un elemento al azar (considerando que todos son equiprobables) en vez de elegir sistemáticamente al de la primera posición. Con esta variante estamos forzando la ocurrencia del caso promedio. Otra heurística simple y muy conocida es la llamada mediana de tres. En este caso se escogen tres elementos del vector al azar (muestra) y como pivote se elige a la mediana de los tres. También puede elegirse como pivote a la mediana de 5, de 7, etc. El tamaño óptimo de la muestra dependerá del tamaño del vector (de hecho es $\sqrt{n}$ para vectores de tamaño n.

Como ya se había mencionado, quick–sort es un algoritmo de ordenación que tiene muchas ventajas, entre ellas, que es fácil de implementar y que se ejecuta muy rápidamente en la mayoría de los casos, por tanto en la práctica es el método que conviene utilizar. También tiene algunas desventajas, como por

ejemplo que no es estable, que su coste en tiempo en el caso peor es alto y que es un algoritmo frágil en el sentido de que un simple error de implementación puede pasar desapercibido y causar un mal desempeño en algunos vectores.