Azzolini Riccardo 2019-04-08

Quicksort

1 Operazione di partizionamento

Problema:

- *Input*: una sequenza $V \in U$ e un elemento $x \in V$, chiamato **pivot** (“perno”).
- *Output*: una sequenza $\tilde{V} = V_1 \cdot x \cdot V_2$, con $V_1, V_2 \in U^\star$, $|V_1| + |V_2| = n - 1$, che contiene tutti gli elementi di V e nella quale il pivot x fa da “spartiacque” tra gli elementi $< x$ e $> x$, cioè

$$\begin{aligned} \forall y \in V_1, \quad & y \in V,\ y \leq x \\ \forall z \in V_2, \quad & z \in V,\ z \geq x \end{aligned}$$

  Di conseguenza, x occupa la sua posizione definitiva nella sequenza ordinata.

Quest’operazione si può implementare sia su vettori che su liste.

2 Partizionamento di un vettore

```java
private static int partition(Comparable[] a, int l, int r) {
    Comparable v = a[l];
    int i = l;
    int j = r + 1;
    while (true) {
        do { i++; } while (less(a[i], v) && i < r);
        do { j--; } while (less(v, a[j]) && j > l);
        if (i >= j) break;
        exch(a, i, j);
    }
    exch(a, l, j); // Posiziona il pivot
    return j;
}
```

1. Si sceglie come pivot, `v`, il primo elemento del vettore.
2. Si usano due indici, `i` e `j`, per cercare gli elementi “fuori posto”:

- `i` scorre da sinistra a destra fino a trovare un elemento maggiore o uguale al pivot;
- `j` scorre da destra a sinistra fino a un elemento minore o uguale al pivot.

I due elementi fuori posto individuati vengono poi scambiati, e il procedimento si ripete (a partire dall'elemento successivo per `i` e precedente per `j`) fino a quando i due indici si "incrociano", cioè finché non ci sono più elementi fuori posto.

3. Al termine del ciclo `while` esterno, l'indice `j` punta alla posizione in cui deve essere inserito il pivot, la quale al momento è occupata da un elemento minore o uguale a `v`. Si esegue quindi uno scambio per mettere l'elemento pivot in posizione `j`.

4. Viene restituita la posizione del pivot.

Osservazione: L'indice `i` è inizializzato a `l`, la posizione del pivot, ma viene incrementato prima del primo confronto, quindi la scansione parte effettivamente da `l + 1`, saltando il pivot. `j` viene invece inizializzato a `r + 1`, per evitare che venga saltato l'ultimo elemento.

2.1 Complessità

Gli indici `i` e `j` scorrono l'intero vettore fino a incrociarsi, quindi effettuano in totale $\Theta(n)$ passi. Per ogni passo, si eseguono un confronto e un numero costante di altre operazioni a tempo costante. Di conseguenza, sia il numero di confronti che il tempo di calcolo sono $\Theta(n)$.

In particolare, se il vettore non contiene elementi ripetuti, vengono effettuati esattamente $n + 1$ confronti. Infatti, il pivot viene confrontato una sola volta con tutti gli altri $n - 1$ elementi, tranne i due dove gli indici si incrociano, con i quali viene confrontato una seconda volta, e allora il numero di confronti è $n - 1 + 2 = n + 1$.

2.2 Esempio

$$(\underline{6}, 2, 8, 1, 9, 3, 7, 0, 5, 4)$$

$$(\underline{6}, 2, \overset{i}{8}, 1, 9, 3, 7, 0, 5, \overset{j}{4})$$

$$(\underline{6}, 2, 4, 1, \overset{i}{9}, 3, 7, 0, \overset{j}{5}, 8)$$

$$(\underline{6}, 2, 4, 1, 5, 3, \overset{i}{7}, \overset{j}{0}, 9, 8)$$

$$(\underline{6}, 2, 4, 1, 5, 3, \overset{j}{0}, \overset{i}{7}, 9, 8)$$

$$(0, 2, 4, 1, 5, 3, \overset{j}{\underline{6}}, 7, 9, 8)$$

3 Quicksort

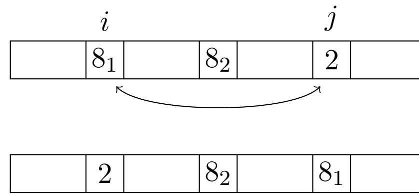
Il **quicksort** è un algoritmo di ordinamento basato sull'operazione di partizionamento di un vettore: dopo di essa, infatti, il pivot si trova nella sua posizione definitiva, quindi ordinando tutti gli elementi minori a sinistra e quelli maggiori a destra si ottiene una sequenza ordinata.

La versione di base è ricorsiva:

```
public class Quick {
    public static void sort(Comparable[] a) {
        sort(a, 0, a.length - 1);
    }

    private static void sort(Comparable[] a, int l, int r) {
        if (r <= l) return;
        int j = partition(a, l, r);
        sort(a, l, j - 1);
        sort(a, j + 1, r);
    }
}
```

Il quicksort *non è stabile*, perché l'operazione di partizionamento può cambiare l'ordine relativo di elementi uguali. Ad esempio:



3.1 Complessità

Le equazioni di ricorrenza per il numero di confronti sono

- *caso peggiore*:

$$T(n) = (n+1) + \max\{T(k) + T(n-k-1) \mid 0 \le k < n\}$$

- *caso migliore*:

$$T(n) = (n+1) + \min\{T(k) + T(n-k-1) \mid 0 \le k < n\}$$

- *caso medio*:

$$T(n) = (n+1) + \frac{1}{n}\sum_{k=0}^{n-1}(T(k) + T(n-k-1))$$

dove

- $n+1$ è il costo dell'operazione di partizionamento, che è fisso (indipendentemente dai dati);
- k e $n-k-1$ sono le dimensioni dei due sottoproblemi ottenuti in seguito al partizionamento. Siccome essi non hanno sempre ugual dimensione, quicksort *non* è (in generale) un algoritmo *divide et impera*.

3.1.1 Caso peggiore

Il caso peggiore si verifica quando, a ogni esecuzione di `partition`, il pivot è il valore minimo o massimo, cioè se

$$k = 0 \implies T(k) + T(n-k-1) = T(0) + T(n-1) = T(n-1)$$

oppure

$$k = n - 1 \implies T(k) + T(n-k-1) = T(n-1) + T(0) = T(n-1)$$

L'equazione di ricorrenza diventa allora

$$\begin{aligned} T(n) &= (n+1) + T(n-1) \\ &= (n+1) + (n) + (n-1) + \cdots + 1 \\ &= \sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2} = \Theta(n^2) \end{aligned}$$

quindi il quicksort *non è ottimale.*

Informalmente, si hanno prestazioni simili in tutti i casi in cui il pivot è vicino al minimo o al massimo.

3.1.2 Caso migliore

Il caso migliore si ha quando, a ogni passo, il pivot è la mediana, cioè l'elemento che si trova a metà nella sequenza ordinata.

In questo caso, entrambi i sottoproblemi hanno dimensione $\frac{n}{2}$, quindi l'equazione di ricorrenza diventa

$$T(n) = (n+1) + 2T\left(\frac{n}{2}\right)$$

che è un'equazione divide et impera con $m = a = 2$, $c = 1$ (il $+1$, così come il coefficiente b, si può trascurare), la cui soluzione asintotica è $T(n) = \Theta(n \log n)$ perché $m = a^c$.

3.1.3 Caso medio

L'equazione di ricorrenza è

$$\begin{aligned} T(n) &= (n+1) + \frac{1}{n}\sum_{k=0}^{n-1}(T(k) + T(n-k-1)) \\ &= (n+1) + \frac{1}{n}(T(0) + T(n-1) + T(1) + T(n-2) + \cdots + T(n-1) + T(0)) \\ &= (n+1) + \frac{2}{n}\sum_{k=0}^{n-1} T(k) \end{aligned}$$

che, moltiplicando entrambi i membri per n, diventa

$$nT(n) = n(n+1) + 2\sum_{k=0}^{n-1} T(k)$$

Sottraendo la stessa equazione, valutata per $n-1$, si ottiene

$$\begin{aligned} nT(n) - (n-1)T(n-1) &= n(n+1) + 2\sum_{k=0}^{n-1} T(k) \\ &- (n-1)n - 2\sum_{k=0}^{n-2} T(k) \\ &= n(n+1) - (n-1)n + 2T(n-1) \\ &= n(n+1-n+1) + 2T(n-1) \\ &= 2n + 2T(n-1) \end{aligned}$$

cioè

$$\begin{aligned} nT(n) &= 2n + (2+n-1)T(n-1) \\ &= 2n + (n+1)T(n-1) \end{aligned}$$

Si dividono poi entrambi i membri per $n(n+1)$:

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{T(n-1)}{n}$$

Ponendo $S(n) = \dfrac{T(n)}{n+1}$, si ha

$$S(n) = \frac{2}{n+1} + S(n-1)$$

con $S(2) = \frac{2}{3}$ per la condizione di arresto $T(2) = 1$. La soluzione asintotica di quest'ultima equazione è

$$S(n) = 2\sum_{k=3}^{n+1} \frac{1}{k} \sim 2\int_1^n \frac{1}{x}\,\mathrm{d}x = 2\ln n$$

da cui si ricava

$$\begin{aligned}\frac{T(n)}{n+1} &\sim 2\ln n \\ T(n) &\sim 2n\ln n \approx 1.39n\log_2 n\end{aligned}$$

ovvero che, in media, il quicksort è (asintoticamente) più lento solo del 39% rispetto al limite teorico per il caso peggiore, $n\log_2 n$.

Esistono numerose versioni avanzate del quicksort, che hanno l'obiettivo di abbassare il coefficiente 1.39.