Progettazione di Algoritmi **Anno Accademico 2021–2022**

Note per la Lezione 9

Ugo Vaccaro

Dato un array `a=a[0]...a[n-1]`, ricordiamo che il rango di un generico elemento `x` in `a` è il numero di elementi che sono $\leq$ di `x` nel vettore `a`. Ad esempio, l'elemento di rango 1 sarà il minimo di `a`, mentre l'elemento di rango n sarà l'elemento di valore massimo in `a`. In questa lezione, consideriamo il seguente problema, che chiameremo `Selezione`:

Input: array `a=a[0]...a[n-1]` di interi *differenti* tra di loro, intero $k \in \{1, 2, \ldots, n\}$
Output: l'elemento di rango k in `a`.

Esiste ovviamente un semplice algoritmo di complessità $O(n \log n)$ per risolvere il problema: ordina `a=a[0]...a[n-1]` e restituisci `a[k-1]`. Possiamo far meglio? Nel caso in cui k è una costante c (c =1, 2, ...) basterà cercare il minimo se $c = 1$, il secondo minimo se $c = 2$, etc...) e quindi risolvere il problema in tempo $\Theta(n)$. Analogamente se $k = n - c$ per qualche costante $c = 0, 2, ...$ basterà cercare il massimo se $c = 0$, etc., e di nuovo il problema lo si potrà risolvere $\Theta(n)$. Ed in generale?

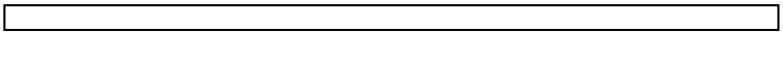
Prima ancora di chiederci come trovare l'elemento di rango k nell'array `a=a[0]...a[n-1]`, chiediamoci: come stabilire che un dato elemento `x` di `a=a[0]...a[n-1]` è (o non è) l'elemento di rango k di `a=a[0]...a[n-1]`? Un modo semplice potrebbe essere il seguente: dividiamo (in qualche modo) l'array `a=a[0]...a[n-1]` in due sottoarray A_1 ed A_2, dove A_1 contiene tutti gli elementi `y` di `a` che sono `<x` ed A_2 contiene tutti gli elementi `y` di `a` che sono `>x`. In figura:

A_1 =tutti gli elementi `y<x`	`x`	A_2 = tutti gli elementi `y>x`

Se il numero di elementi $|A_1$ in A_1 è tale che $|A_1| = k - 1$, l'elemento `x` è proprio l'elemento di rango k che cercavamo, mentre se $|A_1| \neq k - 1$ l'elemento `x` non è l'elemento di rango k che cercavamo. La cosa interessante è che il test ci dice *molto di più* del solo fatto che l'elemento `x` è o non è l'elemento di rango k di `a=[0]...a[n-1]`. Infatti, è semplice convincersi che valgono le seguenti affermazioni:

- Se $|A_1| = k - 1$, allora l'elemento di rango k è proprio `x`.
- Se $|A_1| \geq k$, allora l'elemento di rango k sarà in A_1
- Se $|A_1| < k - 1$, allora l'elemento di rango k sarà in A_2.

Un possibile algoritmo basato su Divide et Impera potrebbe quindi essere il seguente: Partiamo dall'array `a=[0]...a[n-1]`

`a` = 

e vogliamo trovare l'elemento di rango k in `a=a[0]...a[n-1]`.

Scegliamo un elemento `x` in `a=a[0]...a[n-1]` e dividiamo `a=[0]...a[n-1]` in A_1 e A_2 come prima detto:

A_1 =tutti gli elementi `y<x`	`x`	A_2 = tutti gli elementi `y>x`

Se $|A_1| = k-1$, allora l'elemento di rango k é proprio `x` e lo ritorniamo. Se $|A_1| \geq k$, allora l'elemento di rango k sarà in A_1 e quindi ricorreremo in A_1. Se $|A_1| < k-1$, allora l'elemento di rango k sarà in A_2 e quindi ricorreremo in A_2.

Come suddividere `a=[0]...a[n-1]` in A_1, contenente tutti gli elementi `y` di `a` che sono `<x` ed in A_2 contenente tutti gli elementi `y` di `a` che sono `>x`.? Ciò può essere fatto scorrendo `a` da sinistra a destra e confrontando uno ad uno tutti gli elementi di `a` con `x`. Se l'elemento confrontato è `<x` allora lo si mette in A_1, se l'elemento confrontato è `>x` allora lo si mette in A_1.

Il seguente algoritmo `Distribuzione(a, sx, px, dx)` esegue appunto ciò che abbiamo appena detto. Il significato dei parametri in input a `Distribuzione` è il seguente:

- `sx` è l'indice che definisce l'estremità sinistra della parte di `a` correntemente sotto esame,
- `dx` è l'indice che definisce l'estremità destra della parte di `a` correntemente sotto esame,
- `px` è l'indice dell'elemento `x` di `a` che useremo per suddividere `a` in A_1 ed A_2 (cioè `x=a[px]`). Chiameremo *pivot* l'elemento che usiamo, ad ogni rispettivo passo dell'esecuzione dell'algoritmo, per suddividere `a` in A_1 ed A_2.

```
Distribuzione(a, sx, px, dx)
1.  IF (px!= dx) Scambia (px, dx)
2.  i=sx
3.  j=dx-1
4.  WHILE (i<= j) {
5.     WHILE ((i<= j) && (a[i]<= a[dx]))
6.       i=i+1
7.     WHILE ((i<= j) && (a[j]>= a[dx]))
8.       j=j-1
9.  IF (i<j) Scambia (i, j)
}
10.  IF (i!=dx) Scambia (i,dx)
11.  RETURN i
```

ed infine

```
Scambia (i, j)
temp=a[j]
a[j]=a[i]
a[i]=temp
```

Per comprendere come l'algoritmo `Distribuzione` opera, eseguiamo `Distribuzione(a,0,1,7)` sulla sequenza `a=a[0]...a[7]= 6 4 2 10 9 3 5 8`. Quindi, `sx=0, px=1, dx=7`.
Poichè `px = 1 !=7`, scambiamo tra di loro le posizioni di `a[1]=4` con `a[7]=8`, ed otterremo la sequenza `6 8 2 10 9 3 5 4`.
Poniamo `i=0` e `j=6` ed iniziamo ad eseguire il `WHILE` al passo 4. Poichè `a[0] =6 > a[7]=4` ci fermiamo immediatamente senza incrementare il valore di `i` (e ciò è giusto in quanto vogliamo che a "sinistra" della sequenza

compaiano elementi più piccoli di 4) ed eseguiamo il `WHILE` al punto 7. Poichè `a[6]=5 > 4`, ciò ci va bene (infatti vogliamo che a destra compaiano elementi più grandi di 4) e continuiamo, decrementando il valore di `j` a 5. Confrontiamo `a[5]=3` con `a[7]=4` e ci fermiamo, in quanto `3<4`. Avendo eseguito i `WHILE` dei passi 5. e 7, verifichiamo che `i=0<5` e scambiamo gli elementi `a[0]=6` con `a[5]=3` tra di loro, per ottenere la sequenza `3 8 2 10 9 6 5 4`. Effettuiamo un'altra esecuzione del `WHILE` al passo 5. e aumentiamo il valore di `i` ad `1` e ci fermiamo in quanto `a[1]=8> 4 = a[7]`. Eseguiremo il `WHILE` al passo 7. fin quando `j` si decrementa fino a 2, in quanto solo in questo caso `a[j]<a[7]=4`. Quindi, adesso `i` varrà 1 e `j` varrà 2. Poichè `i<j`, effettuiamo lo scambio e la nuova sequenza sarà `3 2 8 10 9 6 5 4`. Adesso i verrà incrementato a 2 ci fermiamo (in quanto `a[2]=8 > a[7]=4`), `j` verrà decrementato a 1 e ci fermiamo (in quanto `a[1]=2<a[7]=4`), NON eseguiremo lo scambio in quanto adesso `i=2>1=j`. Terminremo il `WHILE` al passo 5. e poichè `i=2 !=7`, scambiamo `a[2]=8` con `a[7]=4` per ottenere la sequenza `3 2 4 10 9 6 5 8` e resituiamo il valore `i=2`. Osserviamo che tutti i valori a "sinistra" di `a[2]` sono più piccoli di `a[2]` e tutti i valori a "destra" di `a[2]` sono più grandi di `a[2]`.

É ovvio che l'algoritmo `Distribuzione` richiederà un tempo $\Theta(n)$ quando viene chiamato su di una sequenza composta di n elementi.

Osserviamo, inoltre, che la procedura `Distribuzione` permette di trovare il rango del pivot scelto, posizionando tutti gli elementi inferiori al pivot alla sua sinistra e tutti gli elementi più grandi del pivot alla sua destra. In base a tale osservazione, otteniamo il seguente algoritmo per calcolare l'elemento di rango k in `a=a[0]...a[n-1]`.

```
1.  QuickSelect (a,sinistra,k,destra)
2.  IF (sinistra == destra) {
3.     RETURN a[sinistra]
4.     } ELSE {
5.       scegli pivot nell'intervallo [sinistra...  destra]
6.       rango=Distribuzione (a, sinistra, pivot, destra)
7.       IF (k-1 == rango) {
8.        RETURN a[rango]
9.       } ELSE IF (k-1< rango) {
10.        RETURN QuickSelect (a,sinistra,k,rango-1)
11.       } ELSE {
12.        RETURN QuickSelect (a,rango+1,k,destra)
     }
}
```

Se volessimo analizzare la complessità $T(n)$ di `QuickSelect(a,0,k,n-1)` nel caso peggiore, allora non potremmo far altro che assumere che l'elemento che cerchiamo si trovi sempre nella parte di array `a` "più grande", per cui avremmo un'equazione di ricorrenza del tipo

$$T(n) \leq \begin{cases} c & \text{se } n \leq 1 \\ T(\max(r-1, n-r)) + dn & \text{altrimenti} \end{cases} \tag{1}$$

dove $r =$`rango +1` (ovvero la posizione occupata dall'elemento `a[pivot]` dopo la chiamata di `Distribuzione`). Non è difficile vedere che la (1) ammette diversi tipi di soluzione, a seconda del valore di r. Ad esempio, se la ricorsione fosse *sempre* del tipo $T(n) \leq T(n-1) + dn$ (e ciò potrebbe sicuramente accadere se ad ogni passo scegliessimo un pivot di rango 1) avremmo una soluzione $T(n) = O(n^2)$, mentre se fosse sempre del tipo $T(n) \leq T(n/2) + dn$ (e ciò potrebbe sicuramente accadere se ad ogni passo scegliessimo un pivot di rango $n/2$)

avremmo una soluzione $T(n) = O(n)$. Purtroppo, trovare in maniera efficiente un pivot di un *dato* rango è proprio il problema che vogliamo risolvere, e quindi non lo possiamo dare per già risolto!

Che succede se scegliessimo il pivot *a caso*? Supponendo di procedere in questo modo, avremo che per la equazione che governa la complessità $T(n)$ di `QuickSelect(a,0,k,n-1)` varrà la relazione $T(n) \leq T(\max(r-1, n-r)) + \Theta(n)$ se e solo se la nostra scelta a caso del pivot ci ha restituito un pivot di rango r, e ciò avverrà con probabilità $1/n$. Infatti, se calcoliamo la probabilità come (numero di casi favorevoli)/(numero di casi possibili), poichè esiste un *unico* elemento di rango r, per ogn dato r, otteniamo che la probabilità è appunto pari a $1/n$.

Ci troviamo quindi di fronte ad una quantità (il tempo di esecuzione di `QuickSelect`) che assume *differenti* valori con certe probabilità. Quindi, non ha più senso parlare di tempo di esecuzione nel caso peggiore, ma occorrerà valutare il tempo di esecuzione nel caso medio. Ricordiamo che se abbiamo una generica "quantità" T che può assumere differenti valori $t_1, t_2, \ldots, t_n$, con rispettive probabilità $\Pr\{T = t_i\} = p_i$, per $i = 1, \ldots, n$, allora il valore medio di T è

$$E[T] = \sum_{i=1}^{n} t_i \times p_i.$$

Occorre quindi valutare la seguente quantità per `QuickSelect(a,0,k,n-1)`:

$$T(n) = \sum_{\text{su tutti i tempi di esecuzione}} (\text{tempo di esecuzione}) \times \Pr\{\text{di avere quel tempo di esec.}\}$$

Nel caso specifico di `QuickSelect(a,0,k,n-1)` avremo quindi che tale valor medio sarà pari a:

$$\begin{aligned}
T(n) \leq & (T(n-1) + \Theta(n)) \times \Pr\{\text{di aver scelto un elemento } \texttt{a[pivot]} \text{ di rango } 1\} + \\
& (T(\max(1, n-2)) + \Theta(n)) \times \Pr\{\text{di aver scelto un elemento } \texttt{a[pivot]} \text{ di rango } 2\} + \\
& (T(\max(2, (n-3)) + \Theta(n)) \times \Pr\{\text{di aver scelto un elemento } \texttt{a[pivot]} \text{ di rango } 3\} + \\
& \quad \vdots \\
& (T(n-1) + \Theta(n)) \times \Pr\{\text{di aver scelto un elemento } \texttt{a[pivot]} \text{ di rango } n\} \\
= & \frac{1}{n} \sum_{k=1}^{n} T(\max(k-1, n-k)) + \Theta(n).
\end{aligned}$$

dove abbiamo assunto che $T(0) = 0$ ed abbiamo sfruttato il fatto che

$$\Pr\{\text{di aver scelto un elemento } \texttt{a[pivot]} \text{ di rango } k\} = 1/n$$

per ogni possibile valore di k.

Osserviamo ora che

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{se } k > \lceil n/2 \rceil \\ n-k & \text{se } k \leq \lceil n/2 \rceil. \end{cases}$$

Se n è pari, allora nella espressione di $T(n)$ di sopra ogni termine da $T(\lceil n/2 \rceil)$ fino a $T(n-1)$ compare due volte, mentre se n è dispari allora tutti questi termini appaiono due volte mentre il termine $T(\lfloor n/2 \rfloor)$ appare una sola volta. Pertanto, possiamo dire che

$$T(n) \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} T(k) + \Theta(n). \tag{2}$$

Risolveremo questa ricorrenza per induzione su n. Assumiamo che $T(k) \leq ck$, per qualche costante c, e per tutti i valori di $k < n$. Sostituendo in (2) otterremo, per opportuna costante a, che vale:

$$\begin{aligned}
T(n) &\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} T(k) + \Theta(n) \\
&\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an \\
&= \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an \\
&= \frac{2c}{n} \left(\frac{n(n-1)}{2} - \frac{\lfloor n/2 \rfloor (\lfloor n/2 \rfloor - 1)}{2} \right) + an \\
&\leq \frac{2c}{n} \left(\frac{n(n-1)}{2} - \frac{(n/2-1)(n/2-2)}{2} \right) + an \\
&= \frac{2c}{n} \left(\frac{n^2-n}{2} - \frac{n^2/4 - 3n/2 + 2}{2} \right) + an \\
&= \frac{c}{n} \left(\frac{3n^2}{4} + \frac{n}{2} - 2 \right) + an \\
&= c \left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n} \right) + an \\
&\leq \frac{3cn}{4} + \frac{c}{2} + an \\
&= cn - \left(\frac{cn}{4} - \frac{c}{2} - an \right).
\end{aligned}$$

Per terminare la prova, ovvero che $T(n) \leq cn$, occorre far vedere che possiamo sempre scegliere c tale che $(cn/4 - c/2 - an) > 0$. A tale scopo, basta scegliere $c > 8a$.

Morale della lezione: effettuando scelte casuali all'interno dell'algoritmo, possiamo trasformare `QuickSelect` (che ha complessità $\Theta(n^2)$ nel caso peggiore) in un algoritmo di complessità $\Theta(n)$ nel caso *medio*.