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

Note per la Lezione 10

Ugo Vaccaro

Vediamo qualche ulteriore esempio dell'applicazione della tecnica induttiva alla risoluzione di equazioni di ricorrenza. Supponiamo di avere la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} d & \text{se } n \leq 1 \\ T(n/2) + T(n/3) + \Theta(n) & \text{altrimenti.} \end{cases}$$

Vogliamo provare che $T(n) = O(n)$. In altre parole, vogliamo provare che possiamo trovare una costante $c > 0$ tale che $T(n) \leq cn$, per n sufficientemente grande. Procediamo per induzione. Assumiamo l'esistenza di una costante c per cui $T(k) \leq ck$, per tutti i valori di $k < n$, e proviamo che $T(n) \leq cn$. Dall'ipotesi induttiva otteniamo che

$$\begin{aligned} T(n) &= T(n/2) + T(n/3) + \Theta(n) \\ &\leq T(n/2) + T(n/3) + an \qquad \text{(per qualche costante } a\text{)} \\ &\leq c(n/2) + c(n/3) + an \qquad \text{(dall'ipotesi induttiva)} \\ &= c\left(\frac{5n}{6}\right) + an \\ &= cn - c\left(\frac{n}{6}\right) + an \\ &= cn - \left(\frac{cn}{6} - an\right) \\ &\leq cn \qquad \text{(purchè uno scelga } c > 6a\text{).} \end{aligned}$$

Il modo di preocedere sopra delineato può essere applicato anche a ricorrenze più generali. Ad esempio, supponiamo che la funzione $T(n)$ soddisfi la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} d & \text{se } n \leq 1 \\ T(c_1 n) + T(c_2) + \ldots + T(c_k n) + \Theta(n) & \text{altrimenti,} \end{cases}$$

dove $c_1, \ldots, c_k$ sono costanti tali che $c_1 + c_2 + \ldots + c_k < 1$. Allora si può provare (sempre mediante lo stesso metodo induttivo, e sarebbe un utile esercizio applicarlo) che $T(n) = O(n)$. Notiamo che l'equazione di ricorrenza prima risolta (cioè $T(n) = T(n/2) + T(n/3) + \Theta(n)$) è un caso particolare di quella generale, in cui $c_1 = 1/2, c_2 = 1/3$.

$\diamond$

Consideriamo ora il seguente problema.
Input: Matrice di numeri `a[i][j], i=1...n, j=1...n`, in cui ogni riga e colonna è ordinata in senso crescente, numero `x`.
Output: Coppia `(i,j)` se `a[i][j]=x`, "`non c'è`", altrimenti.

Un primo algoritmo basato sulla tecnica Divide et Impera potrebbe essere il seguente. Confrontiamo l'elemento `x` con l'elemento centrale della matrice `a`. Possono accadere due casi:

1. `x<=` dell'elemento centrale della matrice `a`. Sappiamo allora che `x` non può apparire nel quadrante in basso a destra della matrice `a`, per cui ricorreremo nella restante parte della matrice `a`.

2. `x>` dell'elemento centrale della matrice `a`. Sappiamo allora che `x` non può apparire nel quadrante in alto a sinistra della matrice `a`, per cui ricorreremo nella restante parte della matrice `a`.

Ovviamente la ricorsione termina quando la matrice in cui stiamo cercando è composta da un unico elemento. Lo sviluppo del codice per l'algoritmo prima descritto informalmente è dato per esercizio. Osserviamo che si ricorre sempre in una parte di matrice composta da 3 matrici di dimensione $n/2 \times n/2$, per cui avremo che il tempo di esecuzione $T(n)$ dell'algoritmo soddisfa la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} d & \text{se } n \leq 1 \\ 3T(n/2) + 1 & \text{altrimenti.} \end{cases}$$

Usando i risultati generali visti nelle precedenti lezioni, sappiamo che $T(n) = O(n^{\log_2 3})$. Possiamo far meglio. L'idea è di confrontare `x` con l'elemento nella riga `1` e colonna `n` della matrice `a`. Sappiamo che

1. se `x<` dell'elemento nella riga `1` e colonna `n` della matrice `a`, allora che `x` non può apparire nell'ultima colonna di `a`, (in quanto in essa compaiono sicuramente elementi ancora di più grandi di `a[1][n]`, per cui ricorreremo nella restante parte della matrice `a`.

2. nel caso contrario, possiamo senz'altro escludere da future richerche la prima riga di `a`.

L'algoritmo sarà quindi

```
CercaInMatrice(a,x)
1. i=1, j=n
2. while (i<n+1&&j>=1)
3.   if(a[i][j]==x){
4.    Return(i,j)
    }
5.   if(x<a[i][j]){
6.    j=j-1
7.    } else {
8.    i=i+1
  }
}
Return(‘‘non c’è’’)
```

La complessità $T(n)$ dell'algoritmo è agevole da valutare. Ad ogni passo o si elimina un'intera colonna oppure si elimina un'intera riga, per cui dopo n passi abbiamo sicuramente terminato. Poichè ogni passo ha un costo costante, si ha che $T(n) = \Theta(n)$ nel caso peggiore.

$\diamond$

Consideriamo ora il seguente problema. Data un vettore numerico `a=a[1]...a[n]`, diremo che un elemento di `a` è maggioritario se e solo se esso compare almeno $\lfloor n/2 \rfloor + 1$ volte in `a`. Il problema che vogliamo risolvere è:

Input: `a=a[1]...a[n]`
Output: `True`, se `a` contiene un elemento maggioritario, `False` altrimenti.

Un primo algoritmo potrebbe procedere in questo modo: innanzitutto si ordina il vettore `a`, poi lo si scorre da sinistra a destra incrementando un contatore quando si incontrano elementi uguali (e riportandolo ad 1 nel caso opposto). Si restituirà `True` appena il contatore supera $n/2$, si restituirà `False` se usciamo dal ciclo senza aver trovato un elemento maggioritario. L'algoritmo completo potrebbe essere il seguente:

```
CercaMaggioritario(a)
1. Ordina(a)
2. last=a[1]; cont=1
3. for(i=2;i<n+1;i=i+1){
4.   if(a[i]==last){
5.    cont=cont+1
6.    if (cont>n/2){
7.    return True
8.    } else {
9.    last=a[i]
10.    cont=1
  }
}
return False
```

La complessità dell'algoritmo è chiaramente dominata dalla complessità del passo 1, che richiede un tempo $\Theta(n \log n)$.

Possiamo migliorare l'algoritmo, effettuando la seguente osservazione: se esiste un elemento maggioritario, esso necessariamente comparirà nella posizione $n/2$ della versione ordinata di `a`, ovvero esso è un elemento mediano (vedi lezione precedente). Di conseguenza, basterà cercare l'elemento mediano di `a` e verificare se esso compare o meno almeno $\lceil n/2 \rceil + 1$ volte in `a`. L'algoritmo completo potrebbe essere il seguente:

```
CercaMaggioritario2(a)
1. m=QuickSelect(a,1,n/2,n)
2. cont=0
3. for(i=1;i<n+1;i=i+1){
4.   if(a[i]==a[m]){
5.    cont=cont+1
  }
6.   if (cont>n/2){
7.    return True
8.    } else {
9.    return False
}
```

In questo caso, la complessità dell'algoritmo è dominata dalla complessità del passo 1, che richiede un tempo (medio) $\Theta(n)$.

◇

Esercizio: Si consideri un array `a=a[1]...a[n]` contenente valori interi ordinati in senso non decrescente; possono essere presenti valori duplicati. Scrivere un algoritmo ricorsivo di tipo divide-et-impera che, dato in input `a=a[1]...a[n]` e un intero `x`, restituisce l'indice (la posizione) della prima occorrenza di `x` in `a=a[1]...a[n]`, oppure restituisce `n+1` se il valore `x` non è presente.

Ad esempio, se `a=[1,3,4,4,4,5,6,6]` e `x=4`, l'algoritmo deve restituire 3, in quanto `a[3]` è la prima occorrenza del valore 4.

Modificare quindi l'algoritmo per restituire la posizione dell'ultima occorrenza di `x` in `a=a[1]...a[n]`, oppure `0` se `x` non è presente,

L'esercizio verrà risolto a lezione.

L'algoritmo `CERCAPRIMAOCCORRENZA(A,x,i,j)` restituisce l'indice della prima occorrenza del valore `x` all'interno del sottovettore ordinato `a[i]...a[j]`. I parametri di input dell'algoritmo devono soddisfare le seguenti precondizioni (oltre al fatto che `a[i]...a[j]` deve essere ordinato in senso non decrescente):

- $\forall$ `k=1,..., i-1` vale che `a[k]<x`
- $\forall$ `k=j+1,...,n` vale che `a[k]`$\geq$ `x`

Date le precondizioni sopra e detta `m` la posizione centrale del sottovettore `a[i]...a[j]`, il valore `m` è il risultato cercato se `A[m]==x`, **e** vale una delle seguenti condizioni:

1. siamo all'inizio del sottovettore `(m==i)`, oppure
2. l'elemento precedente `a[m-1]` ha valore diverso da `x`.

Se nessuna delle due condizioni vale, la ricerca prosegue ricorsivamente su una delle due metà del sottovettore `a[i]...a[j]`, in modo tale da mantenere valida la precondizione di cui sopra.

```
CERCAPRIMAOCCORRENZA(A,x,i,j)
1.  IF(i>j) {
2.       Return n+1
3.      } ELSE {
4.       m=(i+j)/2
5.       IF((a[m]==x)&&(m==i||a[m]≠a[m-1])){
6.       Return m
7.      } ELSE {
8.        IF((a[m]>=x){
9.          Return CERCAPRIMAOCCORRENZA(A,x,i,m-1)
10.         } ELSE {
11.          Return CERCAPRIMAOCCORRENZA(A,x,m+1,j)
    }
   }
```