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

Note per la Lezione 8

Ugo Vaccaro

Consideriamo il seguente problema. Dato un array di numeri `a=a[0]a[1]...a[n-1]`, diremo che una coppia di indici `(i,j)` è un'inversione, se `i<j` e `a[i]>a[j]`. Ad esempio, se `a=a[0]a[1]...a[9]= 1 5 4 8 10 2 6 9 3 7`, allora `(1,2)` è un'inversione in quanto `1<2` e `a[1]=5>a[2]=4`. Il problema algoritmico in questione è così definito:

Input: sequenza `a=a[0]a[1]...a[n-1]` di numeri
Output: numero di inversioni in `a`

Una prima (semplice) soluzione sarebbe quella di controllare tutte le possibili coppie di indici `(i,j)` con `i<j`, e verificare se `a[i]>a[j]`. Ovviamente, ciò darebbe origine ad un algoritmo di complessità $\Theta(n^2)$. Applicando la tecnica Divide-et-Impera possiamo progettare un (più efficiente) algoritmo che risolve il problema del calcolo del numero di inversioni nel modo seguente:

1. Se la sequenza `a` ha un solo elemento, restituisci 0 (essa non ha inversioni)
2. Altrimenti, dividi la sequenza in due sottosequenze `l` e `r`
3. Calcola (ricorsivamente) il numero di inversioni in `l`
4. Calcola (ricorsivamente) il numero di inversioni in `r`
5. Calcola il numero di inversioni `(i,j)`, con `a[i]` elemento di `l` e `a[j]` elemento di `r`
6. Restituisci la somma dei valori calcolati al passo 3. 4. e 5.

Considerando l'esempio `a=a[0]a[1]...a[9]= 1 5 4 8 10 2 6 9 3 7`, come primo passo divideremmo `a` in `l= 1 5 4 8 10` e `r=2 6 9 3 7`. L' unica inversione in `l` corrisponderebbe alla coppia di elementi `5-4`. Le inversioni in `r=2 6 9 3 7` corrisponderebbero alle coppie di elementi `6-3`, `9-3`, `9-7`. Le inversioni `x-y`, dove `x` appartiene a `l` e `y` appartiene a r sarebbero, invece, `4-2, 4-3, 5-2, 5-3, 8-2, 8-3, 8-6, 8-7, 10-2, 10-3, 10-6, 10-7, 10-9`. In totale, avremmo quindi 1+3+13=17 inversioni in `a`.

É evidente che la difficoltà maggiore risiede nel passo 5., ovvero nel calcolare il numero di inversioni `(i,j)`, con `a[i]` elemento di `l` e `a[j]` elemento di `r`. Un algoritmo ovvio consisterebbe nel verificare per ogni coppia di elementi, il primo in `l` ed il secondo in `r`, se essa è o meno un'inversione. Ma questo modo di procedere ci condannerebbe di nuovo ad un algoritmo di complessità $\Theta(n^2)$. Il problema diventerebbe, però, più semplice se `l` ed `r` fossero *ordinate*. In tal caso, infatti, potremmo procedere nel modo seguente.

1. Esamina `l` ed `r` da sinistra a destra
2. confronta `l[i] con r[j]`
3. se `l[i] < r[j]` allora `l[i]` non è invertito con nessun elemento a destra di `r[j]` (perchè? Perchè a destra di `r[j]` ci sono elementi ancora più grandi)
4. se `l[i] > r[j]` allora `r[j]` è invertito con ogni elemento a destra di `l[i]` (perchè? Perchè a destra di `l[i]` ci sono elementi ancora più grandi)

5. inserisci il minimo tra `l[i]` e `r[j]` in una sequenza ordinata `c` ed itera (come in MergeSort)

Ritornando all'esempio di prima, avremmo la versione ordinata di `l=1 4 5 8 10` e `r=2 3 6 7 9`. Confrontando `1` con `2` scopriamo che `1` non è invertito con alcun altro elemento di `r`. Metteremo `1` in `c` e proseguiamo. Confrontiamo ora `4` con `2`. Poichè `4>2` questo ci dice che sicuramente che tutti i rimanenti elementi di `l` sono invertiti con `2`, ovvero abbiamo trovato 4 inversioni. Metteremo `2` in `c` e proseguiamo. Adesso confrontiamo `4` con `3`. Poichè `4>3` questo ci dice che sicuramente che tutti i rimanenti elementi di `l` sono invertiti con `3`, ovvero abbiamo trovato altre 4 inversioni. Metteremo `3` in `c` e proseguiamo. Adesso confrontiamo `4` con `6`. Poichè `4<6` questo ci dice che `4` non è invertito con alcun altro elemento di `r` a destra di `6`. Metteremo `4` in `c` e proseguiamo...

Un possibile pseudocodice per l'algoritmo è il seguente.

```
ContaInversioni(a,i,j)
IF(i>=j) {
RETURN 0
} ELSE {
   m=(i+j)/2
   c1= ContaInversioni(a,i,m)
   c2= ContaInversioni(a,m+1,j)
   c3= ContaInversioniMerge(a,i,m,j)
   RETURN c1+c2+c3
}
```

Lo pseudocodice per `ContaInversioniMerge(a,i,m,j)` può essere il seguente:

```
ContaInversioniMerge(a,i,m,j)
c=0
k=1
i1=i
i2=m+1
WHILE ((i1<=m) && (i2<=j)) {
    IF(a[i1]<=a[i2]){
      b[k]=a[i1]
      i1=i1+1
    } ELSE {
      c=c+(m+1-i1)
      b[k]=a[i2]
      i2=i2+1
    }
    k=k+1
}
WHILE (i1<=m){
    b[k]=a[i1]
    i1=i1+1
    k=k+1
}
```

```
WHILE (i2<=j){
    b[k]=a[i2]
    i2=i2+1
    k=k+1
}
FOR(k=1; k<=j-i+1;k=k+1){
    a[i+k-1]=b[k]
}
RETURN c
```

Per contare le inversioni in `a=a[0]a[1]...a[n-1]` chiameremo quindi l'algoritmo `ContaInversioni(a,0,n-1)` Poichè l'algoritmo `ContaInversioniMerge(a, 0, m, n-1)` richiede tempo dn, per qualche costante d, otteniamo che l'equazione di ricorrenza che la complessità $T(n)$ di `ContaInversioni(a,0,n-1)` soddisfa sarà:

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

Dal risultati precedenti sappiamo che tale equazione di ricorrenza ha soluzione $T(n) = O(n \log n)$. Pertanto, abbiamo ottenuto un bel miglioramento rispetto all'algoritmo semplice che aveva complessità $\Theta(n^2)$.

$\diamond$

Consideriamo il seguente esercizio.

Una sequenza $a = a[0] \dots a[n-1]$ di n interi *distinti* è detta *unimodale* se esiste un indice h tale che $a[0] > \dots > a[h] < ... < a[n-1]$ (cioè la sequenza è dapprima decrescente e poi crescente ed $a[h]$ è il minimo; se $h = 0$ allora la sequenza è crescente, mentre se $h = n-1$ allora è decrescente). Data una sequenza unimodale, si vuole trovare $a[h]$.

Possiamo utilizzare la stessa idea della ricerca binaria per risolvere il problema. Il modo di procedere è il seguente: prendiamo l'elemento *mediano* $a[m]$ del sottovettore considerato, e consideriamo l'elemento a sinistra $a[m-1]$ e a destra $a[m+1]$:

- se tutte le diseguaglianze $a[m-1] > a[m] < a[m+1]$ valgono, allora l'elemento mediano è proprio $a[m]$ ed abbiamo terminato;
- altrimenti, se $a[m-1] > a[m]$, l'elemento che cerchiamo si trova a destra di $a[m]$,
- altrimenti, l'elemento che cerchiamo si trova a sinistra $a[m]$.

Ci sono un paio di casi base: il sottovettore considerato ha 1 o 2 elementi, nel qual caso scegliamo il minimo. La procedura è illustrata nel seguente algoritmo

```
UNIMODALE(a, i, j)
IF (i == j){
   RETURN a[i]
   } ELSE {
      IF (j == i + 1) {
      RETURN min(a[i], a[j])
      }
   }
m = ⌊(i + j)/2⌋
IF ((a[m − 1] > a[m])&&(a[m + 1] > a[m])){
RETURN a[m]
   }
IF (a[m − 1] > a[m]) {
   RETURN UNIMODALE(a, m + 1, j)
   } ELSE {
  RETURN UNIMODALE(a, i, m − 1)
   }
```

La equazione di ricorrenza che descrive la complessità $T(n)$ dell'algoritmo UNIMODALE$(a, 0, n-1)$ è sempre la solita

$$T(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ T(n/2) + c & \text{altrimenti} \end{cases}$$

per costanti c_0 e c opportune. La soluzione dell'equazione di ricorrenza è $T(n) = O(\log n)$.

◇

Consideriamo il seguente esercizio. Dato un array $a = a[0]a[1] \dots a[n-1]$, un minimo relativo di a è un elemento di a che è *minore* dei suoi adiacenti. Per esempio, l'array $a = [3, 4, 2, 8, 7, 10, 12, 15, 14, 20]$ ha tre minimi relativi che sono $2, 7, 14$.

Vogliamo risolvere il seguente problema algoritmico.

Input: array $a = a[0]a[1] \dots a[n-1]$
Output: il numero dei minimi relativi di a.

```
ContaMin(a, i, j)
IF(i == j) {
   IF((a[i] < a[i − 1])&&(a[i] < a[i + 1])) {
      RETURN 1 } ELSE { RETURN 0 }
   } ELSE {c = ⌊(i + j)/2⌋
RETURN (ContaMin(a, i, c)+ContaMin(a, c + 1, j)
  }
```

Basterà poi chiamare la funzione ricorsiva sulla parte interna di $a = a[0]a[1]\dots a[n-1]$, (ossia `ContaMin`$(a, 1, n-2)$), avendo prima verificato che $n > 2$.

Sia $T(n)$ il numero di operazioni effettuate dall'algoritmo `ContaMin`$(a, 1, n-2)$. É ovvio che

$$T(n) \leq \begin{cases} c_0 & \text{se } n = 1 \\ 2T(n/2) + c & \text{altrimenti} \end{cases}$$

per opportune costanti c_0 e c, da cui ne discende che $T(n) = O(n)$.

$\diamond$

Consideriamo il seguente esercizio. Dato una sequenza di numeri $a = a[0]a[1]\dots a[n-1]$, un massimo relativo di a è un elemento di a che è *maggiore o uguale* dei suoi adiacenti (o di un solo adiacente, se stiamo considerando $a[0]$ o $a[n-1]$. Per esempio, la sequenza $a = [5, 10, 20, 15]$ ha un solo massimo relativo, che è 20, mentre la sequenza $[10, 20, 15, 2, 23, 90, 67]$ ha due massimi relativi: 20 e 90. É chiaro che ogni sequenza $a = a[0]a[1]\dots a[n-1]$ ha almeno un massimo relativo. Esaminiamo qualche caso particolare per ottenere un pò di intuizione. Se, ad es., a è ordinato in senso crescente allora l'unico massimo relativo corrisponde ad $a[n-1]$, se a fosse ordinato in senso decrescente allora l'unico massimo relativo corrisponde ad $a[0]$, se a ha tutti gli elementi uguali, allora ogni elemento di a è un massimo relativo, etc..

Vogliamo risolvere il seguente problema algoritmico.

Input: array $a = a[0]a[1]\dots a[n-1]$
Output: un (qualsivoglia) massimo relativo di a.

Una semplice soluzione si può ottenere scorrendo a da sinistra a destra, e fermarci appena si trova un massimo relativo (che sicuramente esiste). L'algoritmo avrebbe complessità $\Theta(n)$ nel caso peggiore.

Una soluzione più efficiente, basata sulla Ricerca Binaria potrebbe essere la seguente. Confrontiamo l'elemento di mezzo di a con i suoi adiacenti. Se tale elemento di mezzo non è inferiore a nessuno dei suoi adiacenti, allora esso è un massimo relativo e lo ritorniamo. Se l'elemento di mezzo è inferiore al suo adiacente "sinistro", allora esiste sicuramente un massimo relativo nella metà di sinistra di a (perchè?). Se l'elemento di mezzo è inferiore al suo adiacente "destro", allora esiste sicuramente un massimo relativo nella metà di destra di a (per gli stessi motivi di prima).

```
TrovaMaxRel (a,i,j,n)
   c=(i+j)/2
   IF ((c == 0 || a[c-1] ≤ a[c]) && (a[c+1] ≤ a[c] || c == n-1)){
         return c
    } ELSE {
     IF (c > 0 && a[c-1] > a[c]) {
         return TrovaMaxRel(a,i, c-1,n)
   } ELSE {
         return TrovaMaxRel(a,c+1, j,n)
 }
```

Sia $T(n)$ il numero di operazioni effettuate dall'algoritmo `TrovaMaxRel (a,0,n-1,n)`. É ovvio che

$$T(n) \leq \begin{cases} c_0 & \text{se } n = 1 \\ T(n/2) + c & \text{altrimenti} \end{cases}$$

per opportune costanti c_0 e c, da cui ne discende che $T(n) = O(\log n)$.

$\diamond$

Consideriamo il seguente esercizio.

Input: array `a=a[1]a[2]...a[n]`, con `[1]<a[n]`
Output: un intero `i` per cui `a[i]<a[i+1]` (esiste sempre!).

Risolviamo (piu in generale) il problema di identificare due valori consecutivi crescenti in un vettore `a[i]...a[j]` con `a[i]<a[j]`. Il problema originale corrisponde al caso `a=a[1]a[2]...a[n]`

Il problema relativo ad `a[i]...a[j]` può essere ridotto ad **uno solo** dei sottoproblemi `a[i]...a[m]` e `a[m]...a[j]`, dove `m=(i + j)/2` è l' indice mediano fra `i` e `j`, in base alle seguenti osservazioni:

- Se `a[i]<a[m]`, allora è possibile considerare il **solo** sottoproblema `a[i]...a[m]`, in cui il primo estremo è minore dell'ultimo.
- Se `a[m]`$\leq$`a[i]<a[j]` allora è possibile considerare il **solo** sottoproblema `a[m]...a[j]` in cui il primo estremo è minore dell'ultimo.

```
Trova-i(a,i,j)
 IF(i+1==j){
 return i
    } ELSE {
     m=(i+j)/2
     IF(a[i]<a[m]){
        return Trova-i(a,i,m)
        } ELSE {
        return Trova-i(a,m,j)
  }
 }
```

Sia $T(n)$ il numero di operazioni effettuate dall'algoritmo `Trova-i(a,1,n)` É ovvio che

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 2 \\ T(n/2) + c & \text{altrimenti} \end{cases}$$

per opportune costanti c_0 e c, da cui ne discende che $T(n) = O(\log n)$.