Progettazione di Algoritmi **Anno Accademico 2023–2024**

Note per la Lezione 7

Ugo Vaccaro

Ricordiamo la struttura di algoritmi basati sulla tecnica Divide-et-Impera:

```
Algoritmo D&I(x)
IF (l'input x é sufficientemente piccolo o semplice) {
        RETURN adhoc(x) (% ovvero risolvi x direttamente)
        } ELSE { decomponi l'istanza di input x in k istanze piú piccole x_1, x_2, ..., x_k
                }
  s1= Algoritmo D&I(x_1)
  s2= Algoritmo D&I(x_2)
    ⋮
  sk= Algoritmo D&I(x_k)
componi le sottosoluzioni s1, s2, ..., sk alle istanze x_i per ottenere una soluzione globale
s alla istanza completa x
RETURN(s)
```

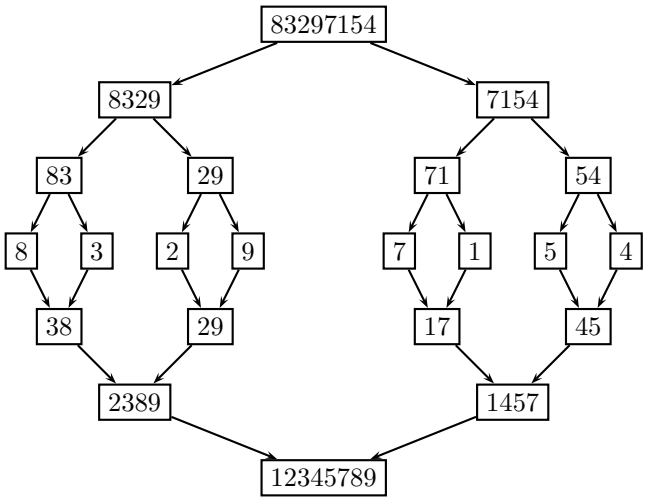
Applichiamo la tecnica Divide-et-Impera al problema dell'ordinamento, così definito:

Input: sequenza `a=a[0]a[1]...a[n-1]` di numeri
Output: una permutazione $j_0, j_1, \ldots, j_{n-1}$ degli indici `0,1, ... , n-1` tale che `a[`j_0`]`$\leq$ `a[`j_1`]`$\leq \ldots \leq$ `a[`j_{n-1}`]`

In ossequio alla tecnica Divide-et-Impera possiamo progettare un algoritmo che risolve il problema dell'ordinamento che procede nel modo seguente:

- Se la sequenza ha un solo elemento, restituisci l'elemento
- Altrimenti, dividi la sequenza in due sottosequenze
- Ordina (ricorsivamente) ciascuna delle due sottosequenze
- Fondi le due sottosequenza ordinate in un'unica sequenza ordinata

Come esempio, immaginiamo di eseguire l'algoritmo sulla seguente sequenza di numeri 83297154. Avremmo una sequenza di operazioni del tipo:



Formalmente, l'algoritmo potrebbe essere di questo tipo:

```
MergeSort(a,i,j)   % 0≤ i ≤ j≤ n-1
1. IF(i<j) {
2. c=(i+j)/2
3. MergeSort(a,i,c)
4. MergeSort(a,c+1, j)
5. MERGE(a,i,c,j)
}
```

dove l'algoritmo `MERGE(a,i,c,j)` prende in input le due sottosequenze `a[i]...a[c]` e `a[c+1]...a[j]` *ordinate* e restituisce un'unica sequenza ordinata contenente tutto gli elementi delle due sottosequenza in input. L'algoritmo `MERGE(a,i,c,j)` può essere descritto dal seguente pseudocodice, in cui si usa un array ausiliario `b` per memorizzare computazioni intermedie:

```
Merge(a,i,c,j)
s=i
d=c+1
k=0
WHILE ((s<=c)&&(d<=j)) {
    IF (a[s]<=a[d]) {
      b[k]=a[s]
      s=s+1
    } ELSE {
      b[k]=a[d]
      d=d+1
      }
    k=k+1
```

```
   }
FOR ( ; s<=c; s=s+1; k=k+1)
   b[k]=a[s]
FOR ( ; d<=j; d=d+1; k=k+1)
   b[k]=a[d]
FOR (s=i;s<=j; s=s+1)
   a[s]=b[s-i]
```

Per ordinare `a=a[0]a[1]...a[n-1]` chiameremo quindi l'algoritmo `MergeSort(a,0,n-1)`. Poichè l'algoritmo `Merge(a,0,c,n-1)` richiede tempo dn, per qualche costante d, otteniamo che l'equazione di ricorrenza che la complessità $T(n)$ di `MergeSort(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 $T(n) = O(n \log n)$.

Diamo un'idea sul perchè la complessità di `MergeSort` è asintoticamente ottima, almeno nella classe degli algoritmi di ordinamento che operano mediante confronti tra elementi.

Consideriamo un *arbitrario* algoritmo $\mathcal{A}$ di ordinamento, capace di ordinare arbitrarie sequenze `a=a[0]...a[n-1]` mediante confronti tra elementi di `a`. É chiaro che il numero di possibili ordinamenti di `a=a[0]...a[n-1]` è pari a $n!$ e che l'algoritmo $\mathcal{A}$ dovrà essere in grado di scoprire quale, tra gli $n!$ possibili, è quello corretto. Sia `(a[i],a[j])` la generica prima coppia di elementi che l'algoritmo $\mathcal{A}$ confronterà. Denotiamo con A l'insieme di tutti i possibili ordinamenti di `a=a[0]...a[n-1]` in cui `a[i]`$\leq$ `a[j]` e con B l'insieme di tutti i possibili ordinamenti di `a=a[0]...a[n-1]` in cui `a[i]`$>$ `a[j]`. É ovvio che

$$|A| + |B| = n!$$

da cui ne segue che o vale che $|A| \geq n!/2$ oppure sicuramente vale che $|B| \geq n!/2$. Senza perdita di generalità, assumiamo che $|A| \geq n!/2$. Visto che analizziamo la complessità degli algoritmi in base al loro caso peggiore, possiamo senz'altro assumere che la risposta al confronto tra gli elementi della coppia `(a[i],a[j])` ci dice che `a[i]`$\leq$ `a[j]`, per cui l'ordinamento incognito di `a=a[0]...a[n-1]` che stiamo cercando sappiamo essere in A. Consideriamo allora il secondo confronto che l'algoritmo $\mathcal{A}$ eseguirà, sia `(a[k],a[s])` la relativa coppia di elementi coinvolta nel confronto. Possiamo ripetere lo stesso ragionamento di prima, e partizionare l'insieme A in due sottoinsiemi C e D, dove C è fatto da tutti gli ordinamenti in A per cui vale che `(a[k]`$\leq$ `a[s])`, e D è composto da tutti gli ordinamenti in A per cui vale che `(a[k]`$>$ `a[s])`. Come prima, almeno uno degli insiemi C e D avrà cardinalità $\geq |A|/2 \geq n!/4$. Sempre perchè stiamo valutando il caso peggiore, possiamo supporre che la risposta al confronto tra gli elementi `(a[k],a[s])` ci dirà che l'ordinamento incognito si trova proprio in quel sottoinsieme tra C e D di cardinalità $\geq n!/4$. Detto in altri termini, dopo due domande l'ordinamento incognito si trova (nel caso peggiore) in un insieme di cardinalità $\geq n!/2^2$.

É evidente che possiamo iterare il ragionamento ad un numero arbitrario i di domande, e che dopo averle eseguite, l'ordinamento incognito si trova, nel caso peggiore, in un insieme di cardinalità $\geq n!/2^i$. Quando l'algoritmo $\mathcal{A}$ si potrà fermare, dichiarando di aver scoperto con esattezza l'ordimanento giusto di `a=a[0]...a[n-1]`? Lo potrà fare solo quando l'insieme in cui è contenuto l'ordinamento incognito è di cardinalità pari ad 1, e ciò comporta necessariamente che il numero i di domande deve essere tale che $n!/2^i \leq 1$, ovvero $i \geq \log n!$.

Sulla base di quanto prima detto, ci basterà provare che $\log n! = \Omega(n \log n)$. A tal fine osserviamo che

$$n! = n \cdot (n-1) \cdots 2 \cdot 1 \geq n \cdot (n-1) \cdots \frac{n}{2} \geq \underbrace{\frac{n}{2} \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} = \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

da cui

$$\log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log \frac{n}{2} = \Omega(n \log n).$$

Mettendo tutto insieme, ogni algoritmo di ordinamento, basato su confronti, richiederà $\Omega(n \log n)$ confronti nel caso peggiore per ordinare arbitrarie sequenze composte da n elementi. Quindi, `MergeSort` è asintoticamente ottimo in quanto esegue un numero di operazioni $O(n \log n)$.

Dovrebbe essere chiaro che un modo di procedere analogo può permettere di provare l'ottimalità asintotica di altri algoritmi, ad esempio l'algoritmo di ricerca binaria visto la lezione scorsa. Infatti, quando cerchiamo la presenza (o meno) di un elemento `x` in una sequenza arbitraria `a=a[0]...a[n-1]`, il numero di possibili risposte è pari a $n+1$. Ogni confronto ci potrà far eliminare al più metà delle possibili risposte. La condizione di terminazione di ogni algoritmo è che vi sia una sola possibile risposta. Pertando, iterando, avremmo bisogno di un numero di confronti i tale che il numero di possibili risposte sia al più pari ad 1, ovvero $\frac{n+1}{2^i} \leq 1$, da cui $i \geq \log(n+1)$.

$\diamond$

Consideriamo il seguente problema algoritmico:

Input: vettore `a=a[1]a[2]...a[n]`, ordinato in senso crescente, di n numeri distinti, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in `a`

Per esercizio, progettare ed analizzare un algoritmo basato sulla tecnica Divide-et-Impera per la sua soluzione.

Analisi del problema: Poichè manca un unico valore k,

tutti i valori i, con $1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore `a=a[1]a[2]...a[n]`

tutti i valori i, con $k < i$ sono memorizzati nella posizione $i-1$-esima del vettore `a=a[1]a[2]...a[n]`.

Troviamo quindi il **più grande** indice i per cui `a[i]=i` ed il valore mancante sarà $k = i + 1$.

Analizzando le posizioni di `a[i]...a[j]` calcoliamo $m = \lceil (i+j)/2 \rceil$.

• Se `a[m]=m` il più grande indice s per cui `a[s]=s` è compreso tra m e j.

• Se `a[m]>m` il più grande indice s per cui `a[s]=s` è compreso tra i e $m-1$.

Quando ci siamo ridotti a considerare un unico numero, ovvero quando $i = j$, abbiamo trovato il numero mancante!

```
Mancante(a,i,j)   % [cerca in a[i]...a[j], 1≤ i ≤ j≤ n]
1. IF(i==j) {
2.      IF(a[i]==i) {
3.        return i+1
           } ELSE {
4.        return i
       }
   }
5. m=(i+j)/2
6. IF(a[m]==m) {
7.      return Mancante(a,m+1,j)
        } ELSE {
8.      return Mancante(a,i,m-1)
}
```

Complessità: $T(n) = T(n/2) + c \Longrightarrow T(n) = O(\log n)$

$\diamond$

Supponiamo di avere un array A di n numeri positivi, che rappresentano il prezzo di una data azione in n giorni consecutivi (cioè $A[i]$ =prezzo dell'azione al giorno i-esimo).

Vogliamo calcolare il *massimo* profitto che possiamo ottenere comprando un'azione nel giorno i e vendendola nel giorno $j \geq i$, detto in altri temini vogliamo risolvere il seguente problema algoritmico:

Input: Array $A = A[1...n]$ di n numeri interi positivi.
Output:

$$\max_{1 \leq i \leq j \leq n} (A[j] - A[i]).$$

Esempio: Sia $A = A[1...8] = [3, 8, 1, 5, 6, 7, 2, 4]$.

Si può vedere che

$$\max_{1 \leq i \leq j \leq 8} (A[j] - A[i]) = A[6] - A[3] = 6.$$

Prima idea: calcola *tutti* i valori $A[j] - A[i]$ e ritorna il massimo.

```
MaxProfitto1(A)
   bestProfit = 0
   FOR(i = 1; i < n + 1; i = i + 1)
     FOR(j = i; j < n + 1; i = i + 1)
       bestProfit = max(bestProfit,A[j]-A[i])
   return bestProfit
```

Complessità $T(n) = \Theta(n^2)$.

Seconda idea: applica D&I.

```
MaxProfitto2(A,i,j)
1.    if(i == j) {
2.     return 0
      }
3.    m = (i + j)/2
4.    miglioreaS=MaxProfitto2(A,i,m)
5.    miglioreaD=MaxProfitto2(A,m+1,j)
6.    miglioreaC=max(A[m+1...j])-min(A[i...m])
7.    return max(miglioreaS, miglioreaD, miglioreaC)
```

Complessità $T(n) = 2T(n/2) + O(n) \Longrightarrow O(n \log n)$

Terza idea: applica D&I *anche* per calcolare il `max(A[m+1...j])` e `min(A[i...m])`.

```
MaxProfitto3(A,i,j)
1.    if(i == j) {
2.     return(0,A[i],A[j])
      }
3.    m = (i + j)/2
4.    (miglioreaS,minaS,maxaS)=MaxProfitto3(A,i,m)
5.    (miglioreaD,minaD,maxaD)=MaxProfitto3(A,m+1,j)
6.    migliore=max(miglioreaS,miglioreaD,maxaD-minaS)
7.    return(migliore,min(minaS,minaD),max(maxaS,maxaD))
```

Complessità $T(n) = 2T(n/2) + O(1) \Longrightarrow T(n) = O(n)$