Corso di laurea in Informatica
Introduzione agli Algoritmi
Didattica blended

Il problema dell'ordinamento: algoritmo Merge Sort

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Sulla base delle slides a cura di T. Calamoneri e G. Bongiovanni per il corso di informatica generale AA 2019/2020

Nella precedente lezione…

…abbiamo dimostrato il seguente:

Teorema. Il costo computazionale di qualunque algoritmo di ordinamento basato su confronti è $\Omega(n \log n)$.

Riusciamo a progettare degli algoritmi che richiedono costo computazionale uguale proprio a $\Theta(n \log n)$ e sono, quindi, ottimi?

Merge Sort

L'algoritmo ***merge sort*** (***ordinamento per fusione***) è un algoritmo ricorsivo che adotta una tecnica algoritmica detta ***divide et impera***. Essa può essere descritta come segue:

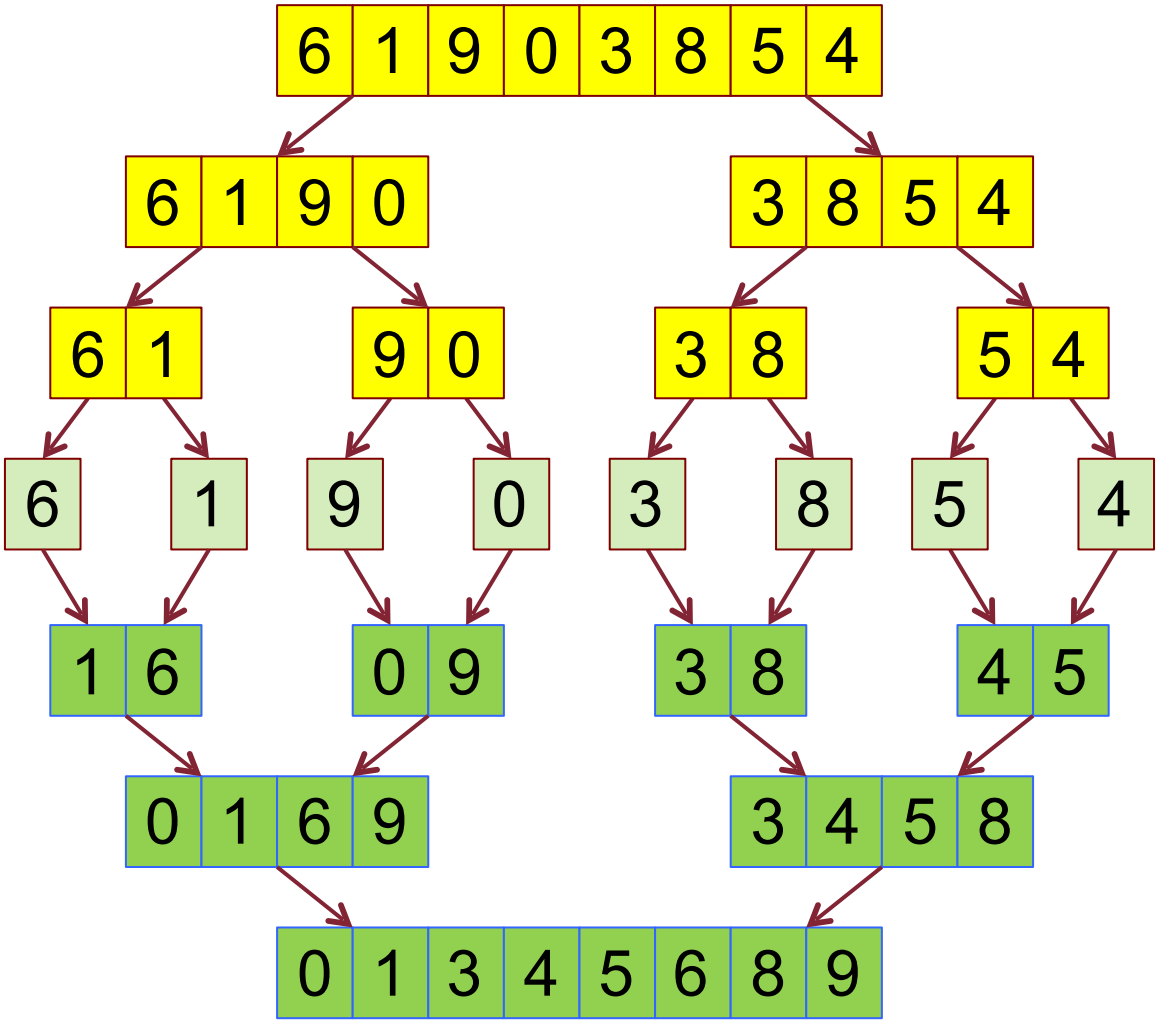
- il problema complessivo si suddivide in sottoproblemi di dimensione inferiore (***divide***)
- i sottoproblemi si risolvono ricorsivamente (***impera***)
- le soluzioni dei sottoproblemi si compongono per ottenere la soluzione al problema complessivo (***combina***).

Merge Sort

L'approccio dell'algoritmo Merge Sort è il seguente:

- ***divide***: la sequenza di *n* elementi viene divisa in due sottosequenze di $\frac{n}{2}$ elementi ciascuna;
- ***impera***: le due sottosequenze di $\frac{n}{2}$ elementi vengono ordinate ricorsivamente;
- ***passo base:*** la ricorsione termina quando la sottosequenza è costituita di un solo elemento, per cui è già ordinata;
- ***combina***: le due sottosequenze – ormai ordinate – di $\frac{n}{2}$ elementi ciascuna vengono "fuse" in un'unica sequenza ordinata di *n* elementi.

Merge Sort



- ***divide***: la sequenza di *n* elementi viene divisa in due sotto-sequenze di *n/2* elementi ciascuna;
- ***impera***: le due sotto-sequenze di *n/2* elementi vengono ordinate ricorsivamente;
- ***passo base:*** la ricorsione termina quando la sotto-sequenza è costituita di un solo elemento, per cui è già ordinata;
- ***combina***: le due sotto-sequenze – ormai ordinate – di *n/2* elementi ciascuna vengono "fuse" in un'unica sequenza ordinata di *n* elementi.

Merge Sort

```
def Merge_sort(lista, ind_primo, ind_ultimo):
    if ind_primo < ind_ultimo:
        ind_medio= (ind_primo+ind_ultimo)//2
        Merge_sort(lista, ind_primo, ind_medio)
        Merge_sort(lista, ind_medio + 1, ind_ultimo)
        Fondi(lista, ind_primo, ind_medio, ind_ultimo)
```

$T(n)$

$\Theta(1)$

$\Theta(1)$

$T(n/2)$

$T(n/2)$

$S(n)$

$$T(n) = \Theta(1) + 2 \cdot T(n/2) + S(n)$$

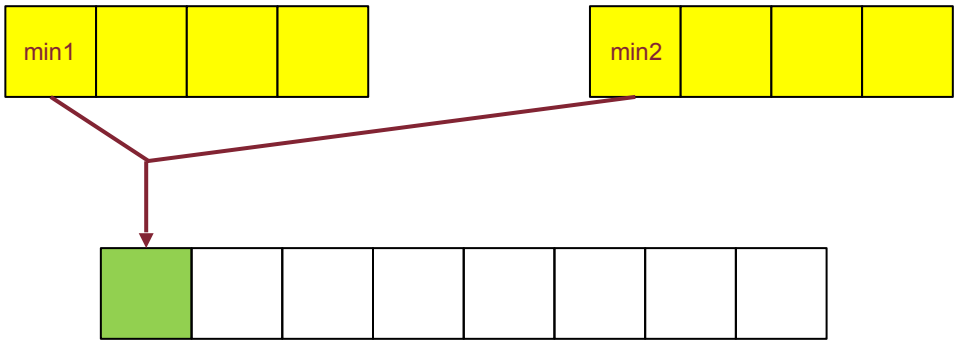
dove $S(n)$ è il costo di $Fondi(\,)$

$$T(1) = \Theta(1)$$

Merge Sort

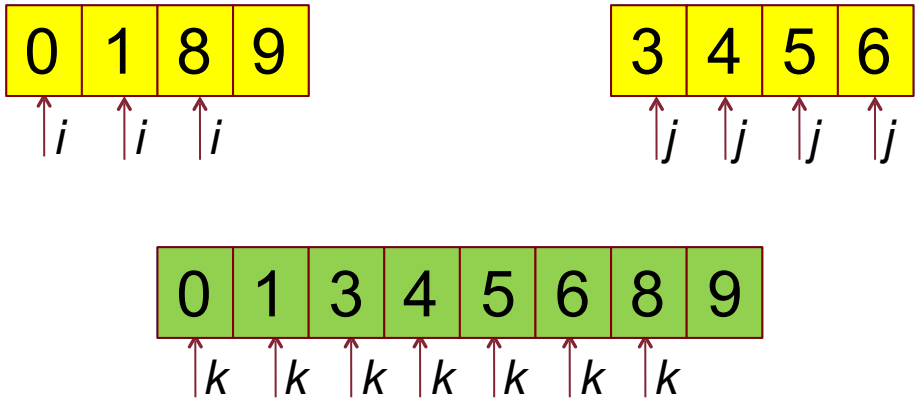
Funzionamento della funzione `Fondi()`:

- la funzione sfrutta il fatto che le due sottosequenze sono ordinate;
- il minimo della sequenza complessiva non può che essere ***il più piccolo fra i minimi delle due sottosequenze*** (se essi sono uguali, scegliere l'uno o l'altro non fa differenza);
- dopo aver eliminato da una delle due sottosequenze tale minimo, la proprietà rimane: il prossimo minimo non può che essere il più piccolo fra i minimi delle due parti rimanenti delle due sottosequenze.



Merge Sort

Esempio di funzionamento della funzione `Fondi()`:



Dopo aver ricopiato anche l'ultimo elemento il vettore complessivo risulta ordinato.

Merge Sort

```
def Fondi (lista, ind_primo, ind_medio, ind_ultimo, B):
   i = ind_primo;  j = ind_medio + 1;  k = ind_primo
   while i <= ind_medio  and  j <= ind_ultimo:
      if lista[i]<lista[j]:
         B[k] = lista[i]
         i += 1
      else:
         B[k] = lista[j]
         j += 1
      k += 1
   while i<= ind_medio: #la prima sottolista  non è terminata
      B[k] = lista[i]; i += 1; k += 1
   while j<=ind_ultimo: #la seconda sottolista non è terminata
      B[k] = lista[j]; j += 1; k += 1
   for i in range(ind_primo, ind_ultimo+1):
       lista[i]=B[i] #ricopio in lista gli elementi in B[ind_primo,ind_ultimo]
```

Merge Sort

Valutiamo il costo computazionale della funzione `Fondi()`:

- inizializzazione delle variabili: $\Theta(1)$;

- primo ciclo `while`
  - ogni iterazione ha costo $\Theta(1)$ e incrementa di 1 l'indice *i* oppure l'indice *j*. Quindi il costo del `while` varia da un minimo di *n/2* a un massimo di *n*, ossia è $\Theta(n)$.

- secondo e terzo `while` (mai eseguiti entrambi):
  - si ricopia nel vettore B l'eventuale "coda" di una delle due sottosequenze: $O(n)$;

- copia del vettore B nell'opportuna porzione del vettore A: $\Theta(n)$.

Dunque il costo S(n) della funzione Fondi() è:

$$S(n) = \Theta(1) + \Theta(n) + O(n) + \Theta(n) = \Theta(n)$$

Merge Sort

Quindi il costo computazionale del Merge Sort:

```
def Merge_sort (lista, ind_primo, ind_ultimo,B):
   if ind_primo < ind_ultimo:
      ind_medio = (ind_primo + ind_ultimo)// 2
      Merge_sort (lista, ind_primo, ind_medio,B)
      Merge_sort (lista, ind_medio + 1, ind_ultimo,B)
      Fondi(lista, ind_primo, ind_medio, ind_ultimo, B)
```

è

$$T(n) = 2T(n/2) + \Theta(n)$$
$$T(1) = \Theta(1)$$

$$T(n) = \Theta(n \log n)$$

La ricorsione va attivata dalla funzione:

```
def MS(lista):
    ind_primo=0; ind_ultimo=len(lista)-1
    B=[0 for _ in range(len(lista))]
    Merge_sort(lista, ind_primo, ind_ultimo, B)
```

Merge Sort

OSSERVAZIONE

L'operazione di fusione non si può fare "in loco", cioè aggiornando direttamente il vettore A, senza incorrere in un aggravio del costo.

Infatti, in A bisognerebbe fare spazio via via al minimo successivo, ma questo costringerebbe a spostare di una posizione tutta la sottosequenza rimanente per ogni nuovo minimo, il che costerebbe $\Theta(n)$ operazioni elementari per ciascun elemento da inserire, facendo lievitare quindi il costo computazionale della fusione da $\Theta(n)$ a $\Theta(n^2)$.

Ciò a sua volta risulterebbe nell'equazione di ricorrenza:

$$T(n) = 2T(n/2) + \Theta(n^2)$$

la cui soluzione, come sappiamo, è:

$$T(n) = \Theta(n^2)$$

Esercizio svolto - 1

Esercizio. Nonostante Merge Sort funzioni in tempo $\Theta(n \log n)$ mentre Insertion Sort in $O(n^2)$, i fattori costanti sono tali che l'Insertion Sort è più veloce del Merge Sort per valori piccoli di *n*. Quindi, ha senso usare l'Insertion Sort dentro il Merge Sort quando i sottoproblemi diventano sufficientemente piccoli.

- Si consideri una modifica del Merge Sort in cui il caso base si applica ad una porzione del vettore di lunghezza *k*, che viene ordinata usando Insertion Sort.
- Le porzioni vengono combinate usando il meccanismo standard di fusione.
- Si determini il valore di *k* come funzione di *n* per cui l'algoritmo modificato ha lo stesso tempo di esecuzione asintotico del Merge Sort.

Esercizio svolto

Soluzione. Il codice che realizza tale versione è il seguente.

```
def Merge_Insertion (lista, k, primo, ultimo, B):
   dim =ultimo – primo + 1
   if dim > k:
      medio = (primo + ultimo)// 2
      Merge_Insertion (lista, k, primo, medio)
      Merge_Insertion (lista, k, medio + 1, ultimo)
      Fondi(lista, primo, medio, ultimo, B)
   else InsertionSort(lista, primo, ultimo)
```

La ricorsione va attivata dalla funzione:

```
def MSI(lista,k):
    primo = 0, ultimo = len(lista)-1
    B=[0 for _ in range(len(lista))]
    Merge_Insertion(lista, k, primo, ultimo, B)
```

Esercizio 1 svolto

L'equazione di ricorrenza che lo caratterizza è la seguente:

$$T(n) = 2T(n/2) + \Theta(n)$$

$$T(k) = \Theta(k^2)$$

Risolviamola col metodo iterativo:

$$\begin{aligned}
T(n) &= 2T(n/2) + \Theta(n) \\
&= 2[2T(n/2^2) + \Theta(n/2^1)] + \Theta(n/2^0) \\
&= 2[2[2T(n/2^3) + \Theta(n/2^2)] + \Theta(n/2^1)] + \Theta(n/2^0) \\
&= 2^3 T(n/2^3) + 2^2\Theta(n/2^2) + 2^1\Theta(n/2^1) + 2^0\Theta(n/2^0) \\
&\quad \dots \\
&= 2^h T\left(\frac{n}{2^h}\right) + \sum_{i=0}^{h-1} 2^i \Theta\left(\frac{n}{2^i}\right) \\
&= 2^h T\left(\frac{n}{2^h}\right) + \Theta(h \cdot n)
\end{aligned}$$

Esercizio svolto

Ci fermiamo incontrando il caso base, il che succede quando

$$\frac{n}{2^h} = k, \text{ ossia } 2^h = \frac{n}{k} \Longrightarrow h = \log \frac{n}{k}$$

Sostituendo tale valore nell'espressione precedente otteniamo:

$$T(n) = 2^{\log \frac{n}{k}} T\left(\frac{n}{2^{\log \frac{n}{k}}}\right) + \Theta\left(n \log \frac{n}{k}\right)$$

$$= \frac{n}{k}\Theta(k^2) + \Theta\left(n \log \frac{n}{k}\right)$$

$$= \Theta(nk) + \Theta\left(n \log \frac{n}{k}\right)$$

$$= \Theta(nk) + \Theta(n \log n - n \log k)$$

Se $k = O(\log n)$ otteniamo:

$$T(n) = O(n \log n) + \Theta(n \log n - n \log \log n) = \Theta(n \log n)$$

Esercizio 2 svolto

Nell'algoritmo di Insertion sort è possibile ricercare la posizione in cui inserire l'elemento *i*-esimo tramite la ricerca binaria. Come cambia il costo computazionale dell'algoritmo?

Soluzione.

Per ogni j = 2 to n, l'algoritmo di Insertion Sort in versione standard:

- scorre la parte ordinata della lista da destra a sinistra
- confronta ciascun elemento con x
- trova la posizione corretta di x
- sposta tutti gli elementi che sono a destra per fare posto ad x

Consideriamo lo pseudocodice modificato dell'Insertion Sort:

```
def InsertionSort(lista):
    for i in range(1,len(lista):
        x = lista[i]
        k = RicercaBin(lista,x)
            # x dovrebbe stare in posizione k
        j = i - 1
        while j>0 and j>=k:
            A[j+1] = A[j]
            j -= 1
        A[k] = x
```

iterato n-1 volte

$\Theta(1)$

$O(\log(n-i))$

$\Theta(1)$

iterato t_i volte

$\Theta(1)$

$\Theta(1)$

$\Theta(1)$

- Il contributo della ricerca binaria non si sostituisce a t_i, ma si aggiunge ad esso.
- Anche se troviamo la posizione di x più velocemente, dobbiamo comunque spostare gli elementi a destra…
- pertanto il costo computazionale non migliora!

Corso di laurea in Informatica
Introduzione agli Algoritmi
Didattica blended

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZI

- Scrivere la versione iterativa dell'algoritmo di $Merge\ \ sort$.

- Scrivere la versione ricorsiva dell'algoritmo di Fusione.

- Si supponga di scrivere una variante del Merge sort, chiamata $4Merge\ \ sort$ che, invece di suddividere il vettore da ordinare in 2 parti (e ordinarle separatamente), lo suddivide in 4 parti, le ordina ognuna riapplicando $4Merge\ \ sort$, e le riunifica usando un'opportuna variante $4Fondi$ di $Fondi$ (che fa la fusione su 4 sottovettori invece che su 2).
  - Come cambia, se cambia, il costo computazionale di $4Merge\ \ sort$ rispetto a quello di Merge sort?
  - Come cambia, se cambia, il costo computazionale di un'ulteriore variante $kMerge\ \ sort$ che spezza il vettore in k sottovettori?