Mergesort (cont.)

Informatica@SEFA 2018/2019 - Lezione 16

Massimo Lauria <massimo.lauria@uniroma1.it>*

Venerdì, 16 Novembre 2018

1 Mergesort

La comprensione della struttura dati pila ci permette di capire più agevolmente algoritmi ricorsivi. Ora vediamo il **mergesort** un algoritmo ricorsivo di ordinamento per confronto e che opera in tempo $O(n \log n)$ e quindi è ottimale rispetto agli algoritmi di ordinamento per confronto.

1.1 Un approccio divide-et-impera

Un algoritmo può cercare di risolvere un problema

- dividendo l'input in parti
- risolvendo il problema su ogni parte
- combinando le soluzioni parziali

Naturalmente per risolvere le parti più piccole si riutilizza lo stesso metodo, e quindi si genera una gerarchia di applicazioni del metodo, annidate le une dentro le altre, su parti di input sempre più piccole, fino ad arrivare a parti così piccole che possono essere elaborate direttamente.

Lo schema divide-et-impera viene utilizzato spesso nella progettazione di algoritmi. Questo schema si presta molto ad una implementazione ricorsiva.

*http://massimolauria.net/courses/infosefa2018/

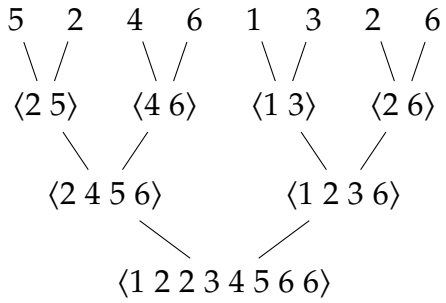
1.2 Schema principale del mergesort

1. dividere in due l'input
2. ordinare le due metà
3. fondere le due sequenze ordinate

Vediamo ad esempio come si comporta il mergesort sull'input

$$\langle 5\ 2\ 4\ 6\ 1\ 3\ 2\ 6\rangle$$

La sequenza ordinata viene ottenuta attraverso questa serie di fusioni.



1.3 Implementazione

Lo scheletro principale del mergesort è abbastanza semplice, e non è altro che la trasposizione in codice dello schema descritto in linguaggio naturale.

```
def mergesort(S,start=0,end=None):
    if end is None:
        end=len(S)-1
    if start>=end:
        return
    mid=(end+start)//2
    mergesort(S,start,mid)
    mergesort(S,mid+1,end)
    merge(S,start,mid,end)
```

1.4 Fusione dei segmenti ordinati

Dobbiamo fondere due sequenze ordinate, poste peraltro in due segmenti adiacenti della stessa lista. L'osservazione principale è che il minimo della sequenza fusa è il più piccolo tra i minimi delle due sequenze. Quindi si mantengono due indici che tengono conto degli elementi ancora da fondere e si fa progredire quello che indicizza l'elemento più piccolo. Quando una delle due sottosequenze è esaurita, allora si mette in coda la parte rimanente dell'altra. `merge` usa una **lista aggiuntiva temporanea** per fare la fusione. I dati sulla lista temporanea devono essere copiati sulla lista iniziale.

```
def merge(S,low,mid,high):
    a=low
    b=mid+1
    temp=[]
    # Parte 1 - Inserisci in testa il pi  piccolo
    while a<=mid and b<=high:
        if S[a]<=S[b]:
            temp.append(S[a])
            a=a+1
        else:
            temp.append(S[b])
            b=b+1
    # Parte 2 -  Esattamente UNA sequenza    esaurita. Va aggiunta l'altra
    if a<=mid:
        residuo = range(a,mid+1)
    else:
        residuo = range(b,high+1)
    for i in residuo:
        temp.append(S[i])
    # Parte 3 - Va tutto copiato su S[start:end+1]
    for i,value in enumerate(temp,start=low):
        S[i] = value
```

Questo conclude l'algoritmo

```
dati=[5,2,4,6,1,3,2,6]
mergesort(dati)
print(dati)
```

```
[1, 2, 2, 3, 4, 5, 6, 6]
```

1.5 Running time

Per cominciare osserviamo che nelle prime due parti di `merge` un elemento viene inserito nella lista temporanea ad ogni passo, e poi questo elemento non viene più considerato. La terza parte ricopia tutti gli elementi passando solo una volta su ognuno di essi. Pertanto è chiaro che `merge` di due segmenti adiacenti di lunghezza n_1 e n_2 impiega $\Theta(n_1 + n_2)$ operazioni.

Definiamo come $T(n)$ il numero di operazioni necessarie per ordinare una lista di n elementi con `mergesort`. Allora

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

quando $n > 1$, altrimenti $T(1) = \Theta(1)$ e dobbiamo risolvere **l'equazione di ricorrenza** rispetto a T. Prima di tutto per farlo fissiamo una costante $c > 0$ abbastanza grande per cui

$$T(n) \leq 2T(n/2) + cn \qquad T(1) \leq c\ . \tag{2}$$

Espandendo otteniamo

$$T(n) \leq 2T(n/2) + cn \leq 4T(n/4) + 2c(n/2) + cn = 4T(n/4) + 2cn \tag{3}$$

Si vede facilmente, ripetendo l'espansione, che

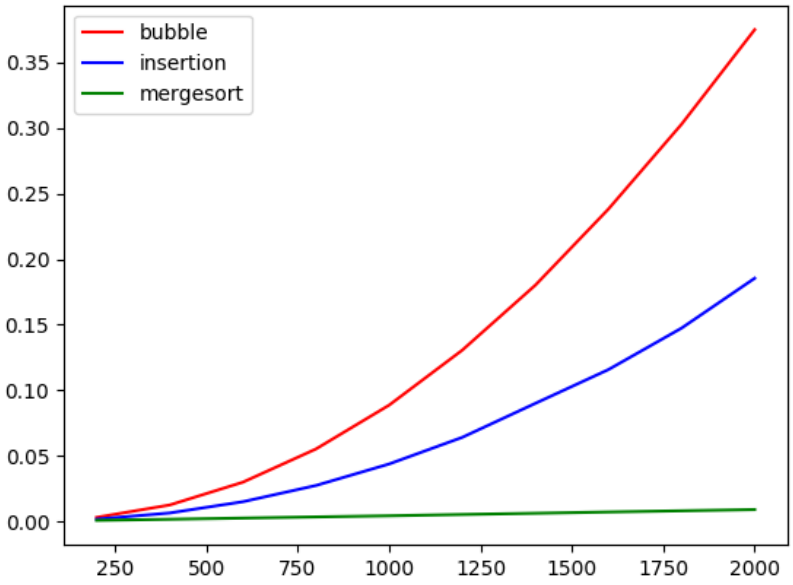
$$T(n) \leq 2^t T(n/2^t) + tcn \tag{4}$$

fino a che si arriva al passo t^* per cui $n/2^{t^*} \leq 1$, nel qual caso si ottiene $T(n) = c2^{t^*} + t^*cn \leq c(t^* + 1)n$.

Il più piccolo valore di t^* per cui $n/2^{t^*} \leq 1$ è $O(\log n)$, e quindi il running time totale è $O(n \log n)$.

Poichè il mergesort è un ordinamento per confronto il running time è $\Omega(n \log n)$, ed in ogni caso questo si può vedere anche direttamente dall'equazione di ricorrenza. Quindi il running time è in effetti $\Theta(n \log n)$.

1.6 Confronto sperimentale con insertion sort e bubblesort



1.7 Una piccola osservazione sulla memoria utilizzata

Mentre `bubblesort` e `insertionsort` non utilizzano molta memoria aggiuntiva oltre all'input stesso, `mergesort` produce una lista temporanea di dimensioni pari alla somma di quelle da fondere. E oltretutto deve ricopiarne il contenuto nella lista iniziale.

Con piccole modifiche al codice, che non vedremo, è possibile controllare meglio la gestione di queste liste temporanee e rendere il codice ancora più efficiente, dimezzando il tempo per le copie e riducendo quello per l'allocazione della memoria. In generale se nessuna di queste liste viene liberata prima della fine dell'algoritmo, la quantità di memoria aggiuntiva è $\Theta(n \log n)$, tuttavia se la memoria viene liberata in maniera più aggressiva allora quella aggiuntiva è $\Theta(n)$.