Mergesort (cont.) e equazioni di ricorrenza

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.

5 2 4 6 1 3 2 6

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

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

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

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

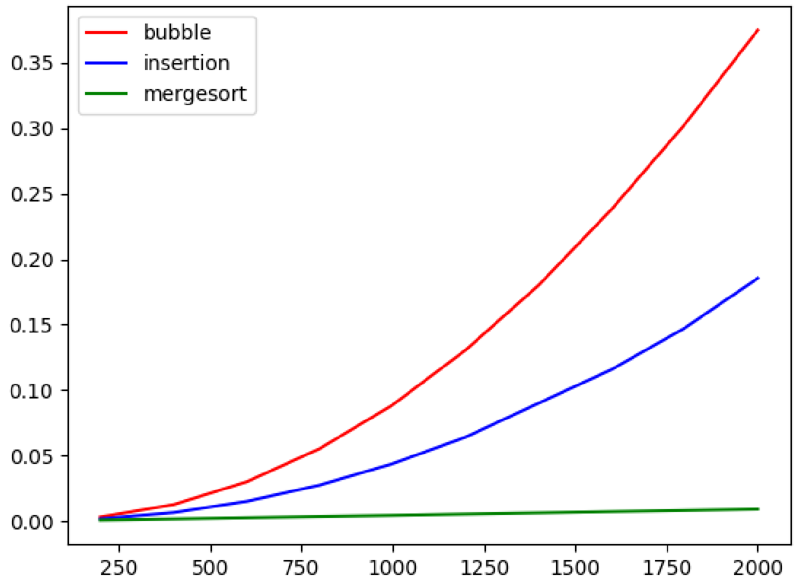
$$T(n) \leq 2^t T(n/2^t) + tcn \tag{4}$$

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 $\Theta(n)$.

2 Ricorsione ed equazioni di ricorrenza

Analizzando il Mergesort abbiamo visto che il tempo di esecuzione di un algoritmo ricorsivo può essere espresso come **un'equazione di ricorrenza**, ovvero un'equazione del tipo

$$T(n) = \begin{cases} C & \text{when } n \leq c_0 \\ \sum_i a_i T(g_i(n)) + f(n) & \text{when } n > c_0 \end{cases} \tag{5}$$

dove a_i, C e c_0 costanti positive intere e g_i e f sono funzioni da $\mathbb{N}$ a $\mathbb{N}$, e vale sempre che $g_i(n) < n$.

Ad esempio il running time di Mergesort è $T(n) = 2T(n/2)+\Theta(n)$.[1] Mentre il running time della ricerca binaria è:

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

Ci sono diversi metodi per risolvere le equazioni di ricorrenza, o comunque per determinare se $T(n)$ è $O(g)$ oppure $\Theta(g)$ per qualche funzione g. Spesso ci interessa solo l'asintotica e per di più a volte ci interessa solo una limitazione superiore dell'ordine di crescita.

2.1 Metodo di sostituzione

Si tratta di indovinare la soluzione della ricorrenza e verificarla dimostrandone la correttezza via induzione matematica. Ad esempio risolviamo la ricorrenza del Mergesort utilizzando come tentativo di soluzione $T(n) \leq cn \log n$ per c "grande abbastanza". Il caso base è verificato scegliendo $c > T(1)$. Assumiamo poi che `merge` utilizzi dn operazioni e che $c > d$. E utilizziamo l'ipotesi induttiva per sostituire nella ricorrenza.

$$T(n) = 2T(n/2)+dn \leq 2c(n/2)\log(n/2)+dn \leq cn\log n - cn + dn \leq cn \log n$$

L'uso dell'induzione per risolvere la ricorrenza può portare ad errori legati alla notazione asintotica. Facciamo conto che vogliamo dimostrare che

[1] In molti casi non è necessario essere precisi nel quantificare $T(1)$, oppure quali sono i valori esatti di C e c_0. Nella maggior parte quei valori condizionano $T(n)$ solo di un fattore costante, che viene comunque ignorato dalla notazione asintotica. Lo stesso vale per la funzione $f(n)$: riscalarla incide sulla soluzione della ricorrenza per un fattore costante.

$T(n) = O(n)$, ovvero $T(n) \leq cn$ per qualche c.

$$T(n) = 2T(n/2) + dn \leq 2cn/2 + dn \leq cn + dn$$

Si sarebbe tentati di dire che $(c + d)n = O(n)$ e che quindi ci siamo riusciti. Tuttavia la dimostrazione usa l'ipotesi induttiva che $T(n') \leq cn'$ per $n' < n$ e quindi se da questa ipotesi non deduciamo la stessa forma $T(n) \leq cn$ l'induzione non procede correttamente.

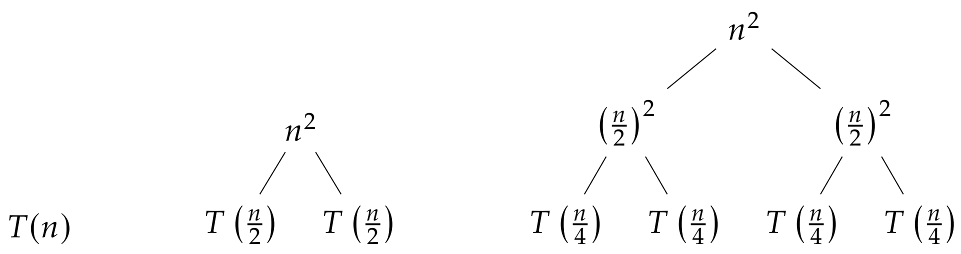
2.2 Metodo iterativo e alberi di ricorsione

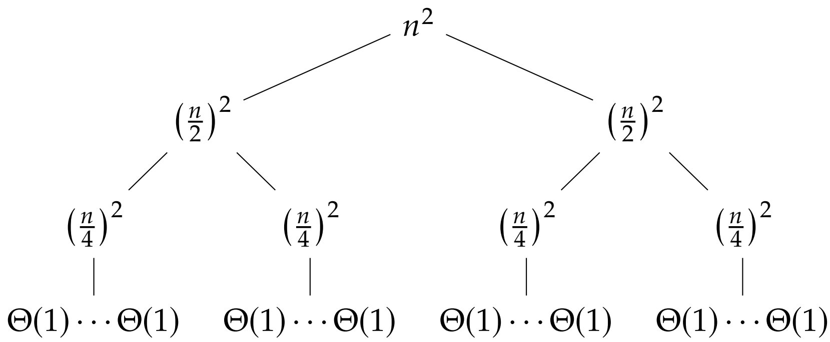
É il metodo che abbiamo utilizzato durante l'analisi delle performance di Mergesort. L'idea è quella di iterare l'applicazione della ricorrenza fino al caso base, sviluppando la formula risultante e utilizzando manipolazioni algebriche per determinarne il tasso di crescita.

Esempio: Analizziamo la ricorrenza $T(n) = 3T(\lfloor n/4 \rfloor) + n$

$$\begin{aligned}
T(n) = \quad & n + 3T(\lfloor n/4 \rfloor) \\
= \quad & n + 3\lfloor n/4 \rfloor + 9T(\lfloor n/16 \rfloor) \\
= \quad & n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor) \\
= \quad & \dots \\
\leq \quad & n + 3n/4 + 9n/16 + 27n/64 + \dots + 3^{\log_4(n)}\Theta(1) \\
\leq \quad & n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + \Theta(n^{\log_4(3)}) \\
= \quad & 4n + o(n) = O(n)
\end{aligned}$$

Per visualizzare questa manipolazione è utile usare un **albero di ricorsione**. Ovvero una struttura ad albero che descrive l'evoluzione dei termini della somma. Vediamo ad esempio $T(n) = 2T(n/2) + n^2$





- L'albero ha $\log n$ livelli
- Il primo livello ha n^2 operazioni, il secondo ne ha $n^2/2$, il terzo ne ha $n^2/4, \ldots$
- L'ultimo ha $\Theta(n)$ operazioni.

il numero totale di operazioni è $\Theta(n) + n^2\left(1 + \frac{1}{2} + \frac{1}{4} + \ldots\right) = \Theta(n^2)$

2.3 Master Theorem

Questi due metodi richiedono un po' di abilità e soprattutto un po' di improvvisazione, per sfruttare le caratteristiche di ogni esempio. Esiste un teorema che raccoglie i casi più comuni e fornisce la soluzione della ricorrenza direttamente.

Teorema 1. *Siano $a \geq 1$ e $b \geq 1$ costanti e $f(n)$ una funzione, e $T(n)$ definito sugli interi non negativi dalla ricorrenza:*

$$T(n) = aT(n/b) + f(n) \ ,$$

dove n/b rappresenta $\lceil n/b \rceil$ o $\lfloor n/b \rfloor$. Allora $T(n)$ può essere asintoticamente limitato come segue

1. *Se $f(n) = O(n^{\log_b(a)-\epsilon})$ per qualche costante $\epsilon > 0$, allora $T(n) = \Theta(n^{\log_b(a)})$;*
2. *Se $f(n) = \Theta(n^{\log_b(a)})$ allora $T(n) = \Theta(n^{\log_b(a)} \log n)$;*
3. *Se $f(n) = \Omega(n^{\log_b(a)+\epsilon})$, per qualche costante $\epsilon > 0$, e se $af(n/b) < cf(n)$ per qualche $c < 1$ e per ogni n sufficientemente grande, allora $T(n) = \Theta(f(n))$.*

Notate che il teorema non copre tutti i casi. Esistono versioni molto più sofisticate che coprono molti più casi, ma questa versione è più che sufficiente per i nostri algoritmi.

- Mergesort è il caso 2, con $a = b = 2$ e $f(n) = \Theta(n)$.
- Ricerca binaria è il caso , con $a = 1$, $b = 2$ e $f(n) = \Theta(1)$.
- $T(n) = 2T(n/2) + n^2$ è il caso 3.

Non vedremo la dimostrazione ma è sufficiente fare uno sketch dell'abero di ricorsione per vedere che questo ha

- altezza $\log_b n$;
- ogni nodo ha a figli;
- al livello più basso ci sono $a^{\log_b(n)} = n^{\log_b(a)}$ nodi che costano $\Theta(1)$ ciascuno;
- i nodi a distanza i da quello iniziale costano, complessivamente, $a^i f(n/b^i)$.

Dunque il costo totale è: $\Theta(n^{\log_b(a)}) + \sum_{j=0}^{\log_b(n)-1} a^j f(n/b^j)$. In ognuno dei tre casi enunciati dal teorema, l'asintotica è quella indicata.