

# Equazioni di ricorrenza / Counting sort

Informatica@SEFA 2018/2019 - Lezione 17

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

Mercoledì, 21 Novembre 2018

## 1 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

$$T(n) = \begin{cases} C & \text{when } n \leq c_0 \\ \sum_i a_i T(g_i(n)) + f(n) & \text{when } n > c_0 \end{cases} \quad (1)$$

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  $T(n) = 2T(n/2) + \Theta(n)$ .<sup>1</sup> 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.

---

\*<http://massimolauria.net/courses/infosefa2018/>

<sup>1</sup>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  $f(n)$ : riscalarela incide sulla soluzione della ricorrenza per un fattore costante.

## 1.1 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  $T(n) = O(n)$ , 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.

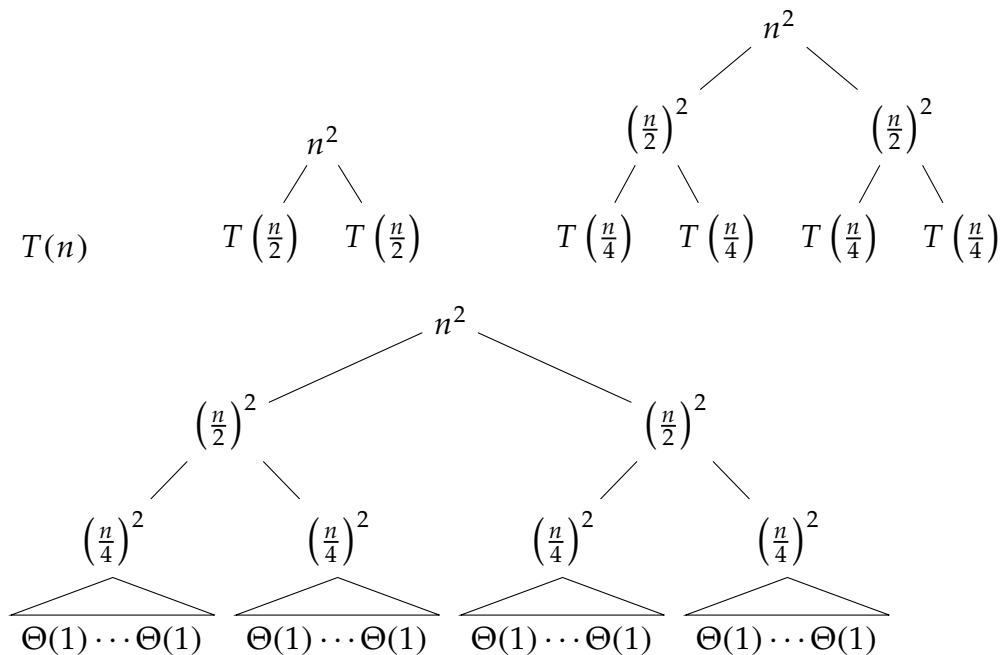
## 1.2 Metodo iterativo e alberi di ricorsione

È 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  $T(n) = 3T(\lfloor n/4 \rfloor) + n$

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

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  $n^2/4, \dots$
- L'ultimo ha  $\Theta(n)$  operazioni.

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

### 1.3 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  $b > 1$  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  $a f(n/b) < c f(n)$  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.

## 2 Ordinamenti in tempo lineare

Esistono modi di ordinare che impiegano solo  $\Theta(n)$ , ma questi metodi non sono, ovviamente, ordinamenti per confronto. Sfruttano invece il fatto che gli elementi da ordinare appartengano ad un dominio limitato.

### 2.1 Esempio: Counting Sort

Il counting sort si basa su un'idea molto semplice: se ad esempio dobbiamo ordinare una sequenza di  $n$  elementi, dove ognuno dei quali è un numero da 1 a 10, possiamo farlo facilmente in tempo lineare:

1. tenendo 10 contatori  $n_1, \dots, n_{10}$ ;
2. fare una scansione della lista aggiornando i contatori;
3. riporre nella lista  $n_1$  copie di 1,  $n_2$  copie di 2, ecc. . .

```
def countingsort1(seq):
    if len(seq)==0:
        return
    # n operazioni
    a = min(seq)
    b = max(seq)
    # creazione dei contatori
    counter=[0]*(b-a+1)
    for x in seq:
        counter[x-a] = counter[x-a] + 1
    # costruzione dell'output
    output=[]
    for v,nv in enumerate(counter,start=a):
        output.extend( [v]*nv )
    return output

print(countingsort1([7,6,3,5,7,9,3,2,3,5,6,7,8,8,9,9,5,4,3,2,2,3,4,6,8,8]))
```

```
[2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 5, 5, 5, 6, 6, 6, 7, 7, 7, 8, 8, 8, 8, 9, 9, 9]
```

Ovviamente qualunque tipo di dato ha un minimo e un massimo, in una lista finita. Tuttavia se la lista contiene elementi in un dominio molto grande (e.g. numeri tra 0 e  $n^{10}$  dove  $n$  è la lunghezza dell'input) allora questo algoritmo è meno efficiente degli algoritmi per confronti.

## 2.2 Dati contestuali

Negli algoritmi di ordinamento per confronto ci i dati originali vengono spostati o copiati all'interno della sequenza, e tra la sequenza ed eventuali liste temporanee.

Si immagini ad esempio il caso in cui ogni elemento nella lista di input sia una tupla  $(i, \text{dati})$  dove  $i$  è la **chiave** rispetto a cui ordinare, ma **dati** invece è informazione contestuale arbitraria. Negli ordinamenti per confronto l'informazione contestuale viene spostata insieme alla chiave. La nostra implementazione del countingsort non gestisce questo caso, e va modificata.

1. Dati i contatori  $n_i$ , calcoliamo in quale intervallo della sequenza di output vadano inseriti gli elementi in input con chiave  $i$ . L'intervallo è tra le due quantità  $\sum_{j=0}^{i-1} n_j$  (incluso) e  $\sum_{j=0}^i n_j - 1$  (escluso).
2. Scorriamo l'input nuovamente e copiamo gli elementi in input nella lista di output.

```
def countingsort2(seq):  
    if len(seq)==0:  
        return  
    # n operazioni  
    a = min(k for k,_ in seq)  
    b = max(k for k,_ in seq)  
  
    # creazione dei contatori  
    counter=[0]*(b-a+1)  
    for k,_ in seq:  
        counter[k-a] += 1  
    # posizioni finali di memorizzazione  
    posizioni=[0]*(b-a+1)  
    for i in range(1,len(counter)):  
        posizioni[i]=posizioni[i-1]+counter[i-1]  
  
    # costruzione dell'output  
    for k,data in seq[:]:  
        seq[posizioni[k-a]]=(k,data)  
        posizioni[k-a] = posizioni[k-a] + 1  
  
sequenza=[(3, "paul"), (4, "ringo"), (1, "george"), (1, "pete"), (3, "stuart"), (4, "john")]  
countingsort2(sequenza)  
print(sequenza)
```

```
[(1, 'george'), (1, 'pete'), (3, 'paul'), (3, 'stuart'), (4, 'ringo'), (4, 'john')]
```