Algoritmi di ordinamento

Materia	Algoritmi
Anno	Secondo Anno
Data	@October 18, 2024

Complessità minima di ogni algoritmo di ordinamento basato per confronti

Possiamo matematicamente dimostrare che un algoritmo di ordinamento basato sui confronti non può fare meglio di $O(n \log n)$.

Consideriamo la lista:

$$[a, b, c]$$

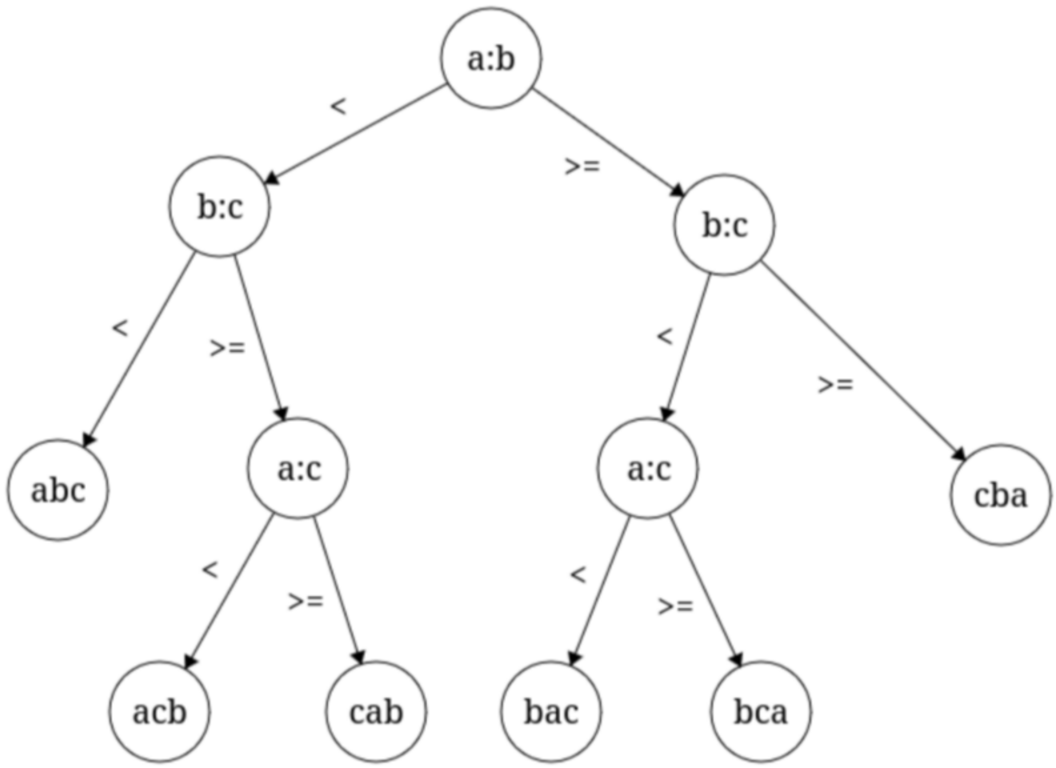
E supponiamo che l'algoritmo di ordinamento stia confrontando a e b. Abbiamo due casi:

- $a < b$;
- $a \geq b$.

Ora confrontiamo b e c. Per b, c ci sono pure due casi, ossia:

- $b < c$;
- $b \geq c$.

Quello che si crea è un **albero di decisione**:



Dove ogni foglia è l'insieme di soluzioni.
Ovviamente un algoritmo di ordinamento basato su confronti non è intelligente come questo, ossia che
non memorizza alcun passo precedente. Quindi supponendo che l'algoritmo sia anche molto intelligente, vediamo quanti sono i possibili confronti.

Se siamo fortunati, un algoritmo basato su confronti ordina l'array con $n-1$ confronti. Di conseguenza, **il ramo più corto dell'albero di decisione ha lunghezza n con $n-1$ confronti.** Questo caso, inoltre, coincide con il caso in cui **l'array sia già ordinato**.

Cerchiamo di capire il numero di foglie. Poiché per un array di n elementi vi sono $n!$ permutazioni, allora il numero di foglie è $n!$. Se, inoltre, tale albero fosse completo, allora vale che:

$$h = O(\log(n!)) = O(n \log n)$$

Di conseguenza il ramo più lungo di un ramo di decisione ha una lunghezza di $n \log n$, pertanto non può fare di meglio.

Ne segue che l'algoritmo di merge sort esegue un numero di confronti proporzionali all'altezza dell'albero, ma **non necessariamente uguale** (ricordiamo che $O(n \log n)$ nasconde una costante c).

Algoritmi di ordinamento non basati su confronti

Ci sono comunque algoritmi di ordinamento che operano in tempo lineare perché **non sono basati per confronti**.

Consideriamo il seguente array:

$$[14, 4, 7, 6, 3, 13, 9, 11, 2]$$

Abbiamo bisogno di un array di posto pari a $max(A)$. Tale array prende il nome di **tabella a indirizzamento diretto**: a partire da i, arrivo alla posizione dell'array C. Tale tabella mi permette di sapere se un elemento è presente o meno.
A questo punto possiamo inizializzare l'array
C nel seguente modo:

- $C[i] = 0$ se i non è presente in A;
- $C[i] = 1$ se i è presente in A.

Quindi il nostro array C è pari a:

$$[0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1]$$

Se un elemento appare più volte, l'insieme C potrebbe contenere non informazione booleana ma informazione **metrica**, ossia metteremo $C[i] = count(i)$, dove $count$ esprime **quante volte i è presente in A**.

Per inizializzare l'array C allora basta la procedura:

```
for i = 0 to n-1 do:
    C[A[i]] = C[A[i]]+1
```

Ora per risolvere basta che **scorro l'array C e per ogni elemento diverso da 0 inserisco $n = C[i]$ valori pari a i.** Tale ordinamento prende il nome di **Counting Sort:**

```
counting_sort(A, n):
    k = max(A,n)
    C = new array(k+1)
    for i = 0 to k do
```

```
        C[i] = 0
    for i = 0 to n-1 do
        C[A[i]] = C[A[i]] + 1
    p = 0
    for i = 0 to k - 1 do
        for j = 1 to C[i] do
            B[p] = i
            p = p + 1
```

La complessità lineare dell'algoritmo dipende dal fatto che k, ossia il numero più grande dell'array, dev'essere **proporzionale a n, ossia $k = O(n)$**. Nella maggior parte dei casi, inoltre, abbiamo sempre valori non troppo alti (il massimo nella maggior parte dei casi è una word di memoria). Possiamo anche sfruttare l'informazione del minimo per capire **quanto è largo il range di numeri presenti in A**, così da utilizzare un array C di dimensione $\max - \min$.

Ovviamente un problema si presenta quando abbiamo un **numero negativo**. Possiamo modificare la procedura nel seguente modo:

```
counting_sort(A, n):
    h = min(A,n)
    k = max(A,n)
    C = new array(k-h+1)
    for i = 0 to k do
        C[i] = 0
    for i = 0 to n-1 do
        C[A[i]-h] = C[A[i]-h] + 1
    p = 0
    for i = 0 to k - 1 do
        for j = 1 to C[i] do
            B[p] = i + h
            p = p + 1
```

Stabilità

Poiché sto ordinando gli elementi **creando un array secondario**, l'algoritmo **non è stabile**, perché **non siamo in grado di copiare gli elementi originali nel nuovo array**. Ad esempio, ordinare un array di studenti in base alla matricola **non è possibile**, perché creiamo solamente un array di matricole ordinate.

Costruiamo l'array C in modo tale che **in $C[i]$ è presente il numero di elementi minori o uguali a i.** Quindi consideriamo:

$$A = [14, 4, 4, 6, 4, 14, 9, 11, 2]$$

Allora avremo:

$$C = [0, 0, 1, 1, 4, 4, 5, 5, 5, 6, 6, 7, 7, 7, 9]$$

In questo modo l'**elemento i di A va in posizione $C[i] - 1$.** Una volta aver inserito l'elemento i-esimo, $C[i]$ va diminuito di 1. Di conseguenza il nostro algoritmo viene modificato:

```
counting_sort(A, n):
    h = min(A,n)
    k = max(A,n)
    C = new array(k-h+1)
    for i = 0 to k do
        C[i] = 0
    for i = 0 to n-1 do
        C[A[i]-h] = C[A[i]-h] + 1
    for i = 1 to k do
        C[i] = C[i] + C[i-1]
    p = 0
    for i = n-1 to 0 do
            B[C[A[i]-min] - 1] = A[i]
            C[A[i]-min] = C[A[i]-min] - 1
```

La particolarità di questo algoritmo è che **è stabile**, perché scorrendo da destra a sinistra **elementi con la stessa chiave verranno posti nella prima posizione disponibile a partire da destra**.

In loco

L'ordinamento può essere eseguito in loco se invece di creare un nuovo array eseguiamo operazioni di swap. Tuttavia, tale operatività **non garantisce stabilità**.

Esempio

Sia dato A:

$$[9_A, 9_B, 7_A, 10, 7_B, 7_C]$$

Inizializziamo il nostro array C con la frequenza degli elementi:

$$[3, 0, 2, 1]$$

Scorriamo l'array C per considerare quanti sono gli elementi minori o uguali a i:

$$[3, 3, 5, 6]$$

Scorriamo l'array da destra a sinistra. Otteniamo:

$$\begin{gathered}
[/, /, /, /, /, /] \\
\downarrow \\
[/, /, 7_C, /, /, /] \\
\downarrow \\
[/, 7_B, 7_C, /, /, /] \\
\downarrow \\
[/, 7_B, 7_C, /, /, 10] \\
\downarrow \\
[7_A, 7_B, 7_C, /, /, 10] \\
\downarrow \\
[7_A, 7_B, 7_C, /, 9_B, 10] \\
\downarrow \\
[7_A, 7_B, 7_C, 9_A, 9_B, 10]
\end{gathered}$$