6° lezione algoritmi

Due Date	@October 17, 2024
Materia	Algoritmi e Laboratorio
Status	Done

Algoritmi di ordinamento

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:

$$[a, b, c]$$

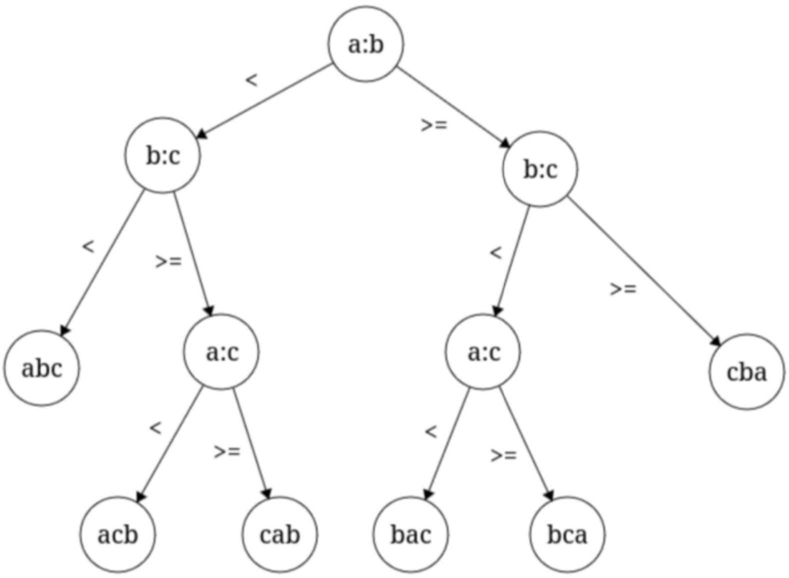
E supponiamo che l'algoritmo di ordinamento stia confrontando a e b .
Abbiamo due casi

$$a < b$$
$$a \geq b$$

Condrontiamo b e c si hanno anche due casi:

$$b < c$$
$$b \geq c$$

Cosi facendo si crea un **albero di decisione**



Dove ogni foglia è l'insieme di soluzioni.
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 algortmo basato su conferonti ordina un array di n elementi con n-1 confronti. Di conseguenza. **il ramo più corto dell'albero di decisone ha lunghezza n con n $n-1$ confronti. Questo caso, inoltre coincide con il caso in cui l'array si presentà già ordinato.**

Come capire quante sono le foglie?

Per un array n n elementi vi sono $n!$ allora si hanno $n!$ foglie.

Se quest' albero fosse completo si avra:

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

Di conseguenza il ramo più lungo di un ramo di decisione ha lunghezza $n \log n$. Non si può fare di meglio

Ne segue che **l'algoritmo merge s**ort esegue un numero di confronti proporzionali all' altezza dell'albero ma **non necessariamente uguali.**

$O(n \log n)$ 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 sui confronti.**

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

Abbiamo bisogno di un **array di posto pari a max(A)**.Tale array prende il nome di **tabella a indirizzamento diretto**. Partendo da i si arriva alla posizione dell' array C. Questa permette di sapere se un elemento è presente o meno nella struttura dati.

Dopo queste considerazioni possimo inizializzare l'array nel seguente modo (C):

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

Dall'esempio dell'array di cui sopra segue che

$$[0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 10, 1]$$

Se un elemento $C[i]$ appare più volte si avrebbe non un informazione booleana ma un informazione metrica che è data da **quante volte i è presente in A.**

'

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

Adesso per risolvere basta scorrere l'array C e **per ogni elemento diverso da 0 si inserisce $n = C[i]$ valori pari ad i. Tale ordinanto viene chiamato** `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 K - 1 DO
        FOR J =1 TO C[I] DO
```

```
            B[P] = I
            P = P+1
```

La complessità lineare dell'algoritmo dipende dipende dal fatto che k, ossia il numero più grande dell'array, deve essere **proporzionale a n,** ossia $k = O(n)$. La 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 vettore C di dimensione $max - min$.

Ovviamente, un problema si presenta quando si ha 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 si stanno ordinando gli elementi creando un vettore secondario, la algoritmo non è stabile, perché non non è possibile copiare gli elementi originali nel nuovo vettore. Ad esempio, ordinare un array di studenti in base alla matricola non è possibile perché si va a creare solo un vettore di matricola ordinate.

Costruiamo il vettore C in modo che in $C[i]$ sarà presente il numero di elementi minori o uguali a i. Pertanto, consideriamo il seguente vettore:

$$A = [14, 4, 4, 5, 4, 14,, 9, 11, 2]$$

Si avrà pertanto:

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

In questo modo l'elemento i di A va in posizione $C[i] - 1$. Una volta inserito l'elemento $i - esimo$, $C[i]$ sarà diminuiti di 1.

```
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
```

Questo algoritmo è stabile, perché scorrendo da destra a sinistra elementi con la stessa chiave verranno inseriti nella posizione disponibile a partire da destra

Può essere eseguito in loco?

L'ordinamento può essere eseguito se invece di creare un nuovo Ray vettore si esegue le operazioni di swap, però questa operatività **non garantisce stabilità**

Esempio

Dato A così formato:

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

1. Se inizializza il vettore C con la frequenza degli elementi nel modo seguente:

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

2. Si scorre il vettore C per considerare quanti sono gli elementi minori o uguali a i nel seguente modo:

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

3. Si scorre il vettore da destra a sinistra e si ottiene:

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