Ordinamenti a cascata e Radix sort

Informatica@SEFA 2018/2019 - Lezione 18

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

Venerdì, 23 Novembre 2018

1 Ordinamento stabile

Nell'esempio precedente abbiamo visto che ci sono elementi diversi che hanno la stessa chiave di ordinamento. In generale una lista da ordinare può contenere elementi "uguali" nel senso che seppure distinti, per quanto riguarda l'ordinamento possono essere scambiati di posizione senza problemi, ad esempio `(1,'george')` e `(1,"pete")` possono essere invertiti nella ordinata, senza che l'ordinamento sia invalidato.

Si dice che un ordinamento è **stabile** se non modifica l'ordine relativo degli elementi che hanno la stessa chiave. Un'inversione nell'ordinamento di una sequenza `S` è una coppia di posizioni `i`, `j` nella lista, $0 \leq$ `i` $<$ `j` $<$ `len(S)` , tali che il valori in `S[i]` si trova dopo il valore in `S[j]` una volta finito l'ordinamento. Un ordinamento stabile minimizza il numero di inversioni.

Tutti gli ordinamenti che abbiamo visto fino ad ora sono stabili. Per esempio nel caso di ordinamenti per confronto è capitato di dover fare operazioni del tipo

```
if S[i] <= S[j]:
   operazioni che non causano un'inversione tra S[i] e S[j]
else:
   operazioni che causano un'inversione tra S[i] e S[j]
```

dove `i` è minore di `j`. Se invece dell'operatore <= avessimo utilizzato l'operatore < allora il comportamento dell'algoritmo sarebbe cambiato solo

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

nel caso in cui `S[i]` fosse stato uguale a `S[j]`. L'ordinamento sarebbe stato comunque valido ma non sarebbe più stato un ordinamento stabile.

1.1 Ordinamenti multipli a cascata

Se avete una lista di brani nel vostro lettore musicale tipicamente avrete i vostri brani ordinati, semplificando, per

1. Artista
2. Album
3. Traccia

nel senso che i brani sono ordinati per Artista, quelli dello stesso artista sono ordinati per Album, e quelli nello stesso album sono ordinati

Un modo per ottenere questo risultato è ordinare prima per Traccia, poi per Album, e poi per Artista. Questo avviene perché gli ordinamenti usati sono stabili. Quando si ordina per Album, gli elementi con lo stesso Album verrano mantenuti nelle loro posizioni relative, che erano ordinate per Traccia. Successivamente una volta ordinati per Artista, i brani dello stesso Artista mantengono il loro ordine relativo, ovvero per Album e Traccia.

In generale è possibile ordinare rispetto a una serie di chiavi differenti, `key1`, `key2`, ... `keyN`, in maniera gerarchica, ordinando prima rispetto `keyN` e poi andando su fino a `key1`. Modifichiamo `countinsort` per farlo lavorare su una chiave di ordinamento arbitraria.

```
def default_key(x):
    return x

def countingsort(seq,key=default_key):
    if len(seq)==0:
        return
    # n operazioni
    a = min(key(x) for x in seq)
    b = max(key(x) for x in seq)
    # creazione dei contatori
    counter=[0]*(b-a+1)
    for x in seq:
        counter[key(x)-a] = counter[key(x)-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 x in seq[:]:
        seq[posizioni[key(x)-a]]=x
```

```
        posizioni[key(x)-a] = posizioni[key(x)-a] + 1                    21
```

2 Ordinare sequenze di interi grandi Radixsort

Abbiamo già detto che il `countingsort` è un ordinamento in tempo lineare, adatto a ordinare elementi le cui chiavi di ordinamento hanno un range molto limitato. Ma se i numeri sono molto grandi che possiamo fare?

Non possiamo ordinare una lista di numeri positivi da 32 bit con il counting sort, perché la lista dei contatori sarebbe enorme (e piena di zeri). Però possiamo considerare un numero di 32 come una tupla di 32 elementi $b_{31} \dots b_0$ in $\{0, 1\}$ ed utilizzare un ordinamento stabile per

- ordinare rispetto a b_0
- ordinare rispetto a b_1
- ...
- ordinare rispetto a b_{31}

Oppure, invece di lavorare bit per bit, possiamo considerare un numero di 32 come una tupla di 4 elementi $b_3 b_2 b_1 b_0$ in $\{0, \dots, 255\}$ ed utilizzare un ordinamento stabile per

- ordinare rispetto a b_0
- ordinare rispetto a b_1
- ordinare rispetto a b_2
- ordinare rispetto a b_3

Naturalmente usare una decomposizione più fitta richiede più chiamate ad ordinamento, ma ognuno su un dominio più piccolo. Il giusto compromesso dipende dalle applicazioni. Ora calcoliamo le chiavi b_i utilizzando quattro funzioni.

```
def key0(x):                                                             1
    return x & 255                                                       2
                                                                         3
def key1(x):                                                             4
    return (x//256) & 255                                                5
                                                                         6
def key2(x):                                                             7
    return (x//(256*256)) & 255                                          8
                                                                         9
```

```
def key3(x):
    return (x//(256*256*256)) & 255

x =2**31 + 2**18 + 2**12 -1
print(key0(x), key1(x), key2(x), key3(x))
```

```
255 15 4 128
```

Dunque possiamo implementare `radixsort` (che ricordiamo, per come è stato realizzato funziona solo su numeri positivi di 32 bit).

```
def radixsort(seq):
    for my_key in [key0,key1,key2,key3]:
        countingsort(seq,key=my_key)


sequenza=[7,6,873823,5,7,9,3,2,12333,5,6132,7,8,1328,9,9,5,463432,4,3426,8,8]
radixsort(sequenza)
print(sequenza)
```

```
[2, 3, 4, 5, 5, 5, 6, 7, 7, 7, 8, 8, 8, 9, 9, 9, 1328, 3426, 6132, 12333, 463432, 873823]
```

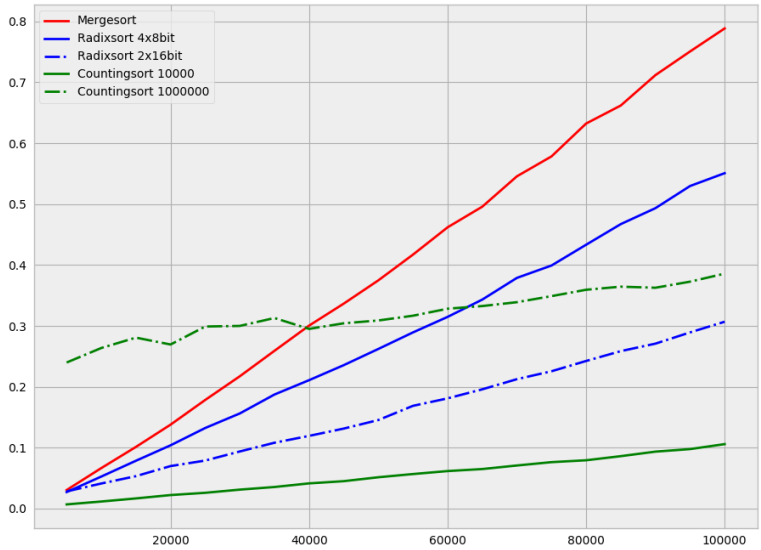
2.1 Plot di esempio

In questo plot vediamo il tempo impiegato da questi algoritmi per ordinare una lista di numeri tra 0 e 1000000. Questi algoritmi sono molto più veloci di `bubblesort` e `insertionsort` e questo si vede anche in pratica. Le liste di numeri non sono particolarmente lunghe (solo 100000 elementi), ma impossibili da ordinare utilizzando ordinamenti $\Theta(n^2)$. Vediamo tre algoritmi:

- Mergesort
- Countingsort con intervallo [0,10000] e [0,1000000]
- Radixsort con 4 chiavi da 8 bit e con 2 chiavi da 16 bit

Il running time di `countingsort` è molto più condizionato dall'intervallo di valori che dalla lunghezza della sequenza da ordinare (almeno per soli 100000 elementi da ordinare).

Radixsort utilizza più chiamate a `countingsort` ma su un dominio più piccolo. Due chiavi a 16 bit sono più efficienti di 4 chiavi a 8 bit, e 16 bit producono uno spazio delle chiavi di 65536 elementi. Uno spazio più semplice da gestire per `countingsort` rispetto ad un domi-nio di `1000000` elementi.


Mergesort
Radixsort 4x8bit
Radixsort 2x16bit
Countingsort 10000
Countingsort 1000000
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
20000
40000
60000
80000
100000