Sommario

Un algoritmo di ordinamento di complessità lineare:

CountingSort

Utilizzabile solo sotto determinate ipotesi!

[CLRS10] cap. 8 par.8.2

Ordinamento in tempo lineare.

Il limite inferiore $\Omega(n \log n)$ vale per tutti gli algoritmi di ordinamento *generali* nel modello basato su confronti.

Se facciamo delle ipotesi restrittive sul tipo degli elementi della sequenza da ordinare possiamo trovare algoritmi più efficienti.
Naturalmente il limite inferiore banale $\Omega(n)$ vale comunque per tutti gli algoritmi di ordinamento.
Se poi utilizziamo altre operazioni al posto dei confronti usciamo dal modello basato sui confronti e il limite inferiore non vale più.

Un esempio banale

Problema: ordinare n interi presi nell'intervallo [1,n], senza duplicati.

Quale complessità è necessaria per ottenere il risultato?

O(n)

perché basta scrivere il risultato:

1,2,…,n-1,n

E se ci sono duplicati o non ci sono tutti?

CountingSort: le ipotesi

Assumiamo che gli elementi dell'array siano interi compresi tra 0 e *k* per una certa costante *k, non troppo grande.*

Studieremo prima una versione semplificata. Per ordinare un array A[1..*n*] l'algoritmo *CountingSort* richiede un secondo array ausiliario C[0..*k*] e lavora correttamente se vale l'ipotesi di cui sopra, e in Θ(n) con k=O(n).

CountingSort semplificato

Come primo passo nell'array ausiliario C[0..*k*] si memorizzano le frequenze degli elementi in A.
Si calcola C[i] = numero di occorrenze di i in A, per 0≤i≤k, previa inizializzazione a 0 di C.

Esempio: A contiene elementi in [0,4].

Dopo il primo passo:

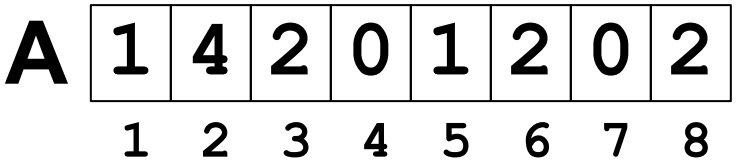
A	1	4	2	0	1	2	0	2
	1	2	3	4	5	6	7	8

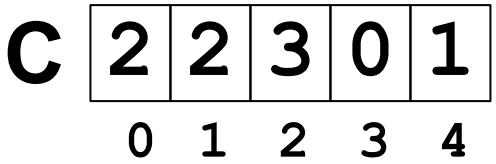
C	2	2	3	0	1
	0	1	2	3	4

CountingSort semplificato

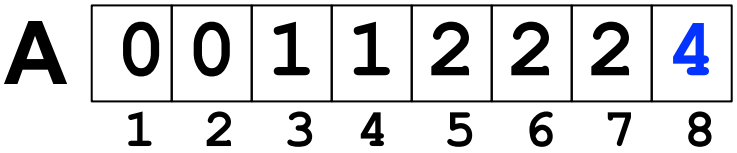
Nel secondo passo si riscrivono gli elementi in A, decrementandone la frequenza in C.

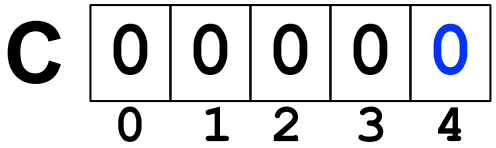
Esempio: A contiene elementi in [0,4].
Dopo il primo passo:





Dopo il secondo passo:





CountingSort semplificato: passo 1

```
for j = 1 to n do
    i = A[j]   A[j] è un numero compreso tra 0 e k
    C[i] = C[i] + 1
```

C[i] è il numero delle occorrenze di A[j]=i

Esempio di esecuzione: A contiene elementi in [0,4]

A	1	4	2	0	1	2	0	2
	1	2	3	4	5	6	7	8

C	0	0	0	0	0
	0	1	2	3	4

C	0	1	0	0	0
	0	1	2	3	4

C	0	1	0	0	1
	0	1	2	3	4

C	0	1	1	0	1
	0	1	2	3	4

C	1	1	1	0	1
	0	1	2	3	4

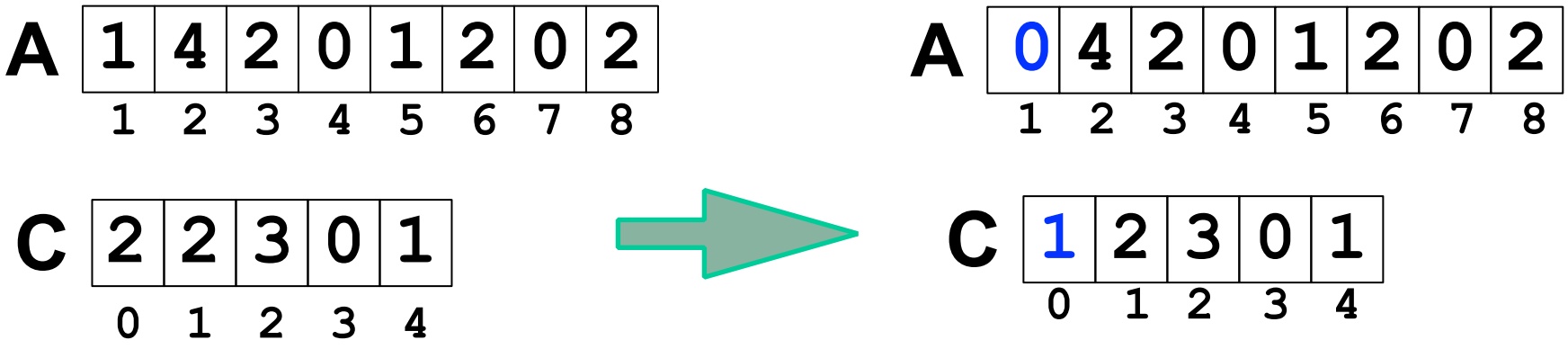
C	1	2	1	0	1
	0	1	2	3	4

…

CountingSort: passo 2

Nel secondo passo si riscrivono gli elementi in A, decrementandone la frequenza in C,

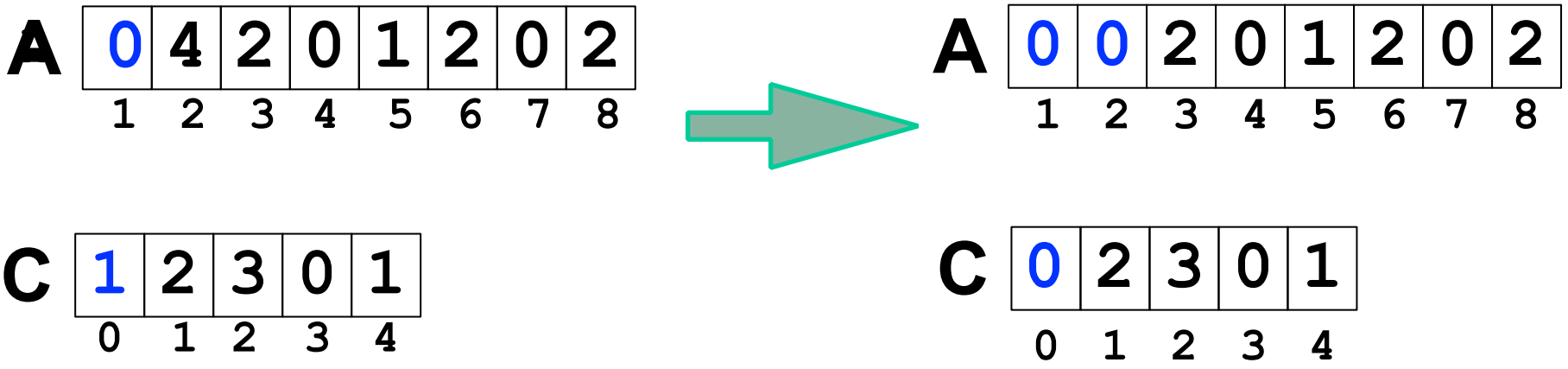
```
j = 1
for i = 0 to k
        while (C[i] > 0)
                A[j] = i;
                C[i] = C[i] - 1
                j = j + 1
```



secondo passo di esecuzione

Anche il secondo elemento viene "sistemato"

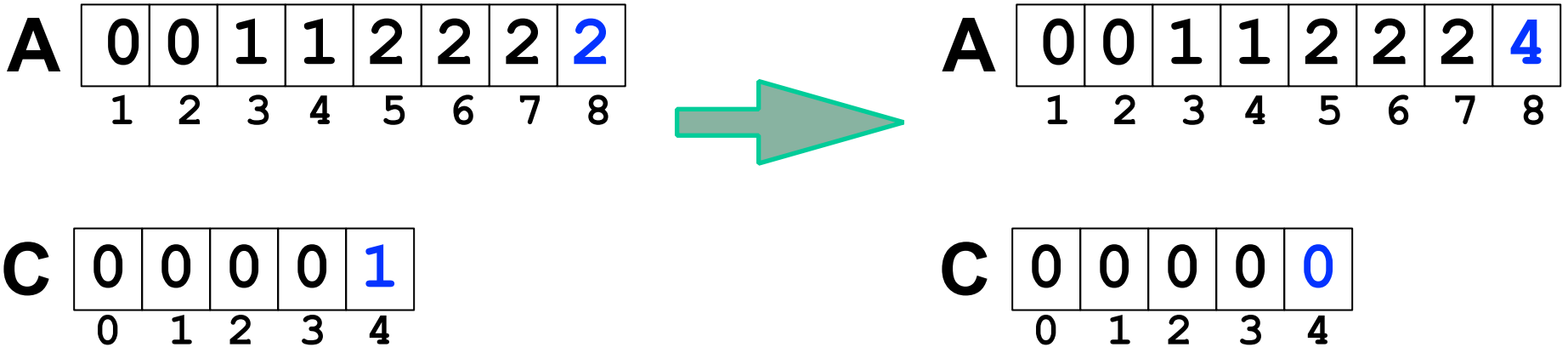
```
j = 1
for i = 0 to k
        while (C[i] > 0)
                A[j] = i;
                C[i] = C[i] - 1
                j = j + 1
```



Esempio ultimo passo di esecuzione

E infine

```
j = 1
for i = 0 to k
        while (C[i] > 0)
                A[j] = i;
                C[i] = C[i] - 1
                j = j + 1
```



CountingSortSemplice(A,k)

input: un array A di n elementi e un intero $k > 0$

prec: gli elementi di A sono interi in [0,k]

postc: restituisce A ordinato, $A[i] \leq A[i+1]$, $1 \leq i < n$

```
for i = 0 to k do C[i] = 0                                  Θ(k)
for j = 1 to n do                                           Θ(n)
    i = A[j]   A[j] è un numero compreso tra 0 e k
    C[i] = C[i] + 1
```

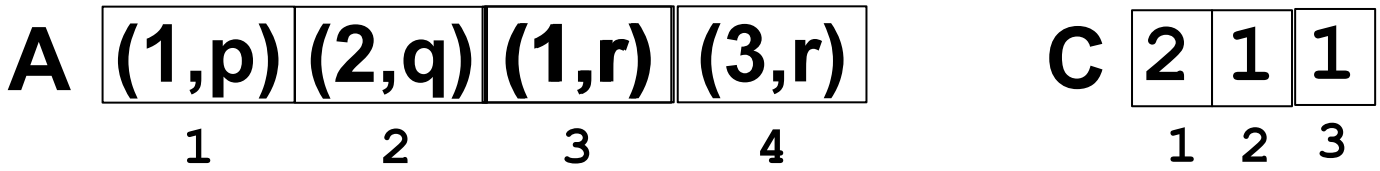
C[i] è il numero delle occorrenze di A[j]=i

```
j=1
for i = 0 to k do                                           Θ(max{n,k})
    while (C[i] > 0)
        A[j] = i
        C[i] = C[i] - 1
        j = j + 1
```

Tempo di esecuzione in $\Theta(n+k)$

CountingSort il caso generale

Se i numeri però sono chiavi di dati complessi, che vengono spostati con le chiavi, è necessario utilizzare un array ausiliario B per ricopiare i dati nell'ordine giusto.
Esempio:

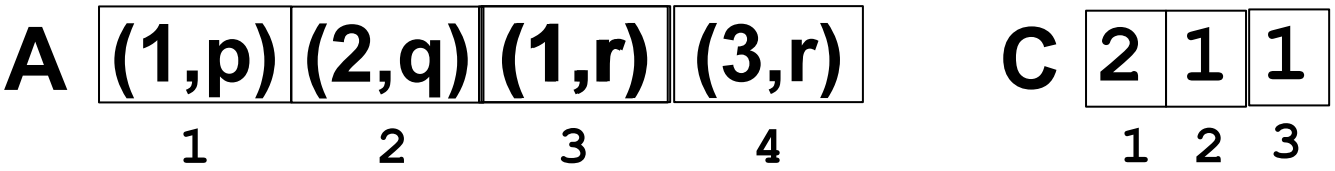


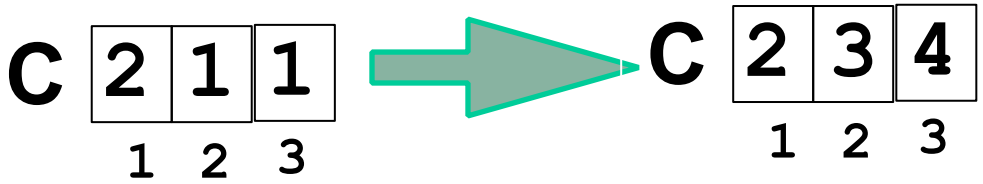
C mi dice che ci sono due elementi con chiave 1, ma se li inserissi in A nelle prime due posizioni perderei informazione: (2,q) sarebbe cancellato.
Inoltre se copio gli elementi in B, ma scorrendo A prima leggo l'elemento (1,p) e in C vedo che ne ho due, potrei copiarlo nella prima posizione di B, poi vedo (2,q), che è l'unico di chiave 2, ma non so dove sistemare in B.
Bisogna allora modificare C in modo che per ogni chiave contenga il numero di elementi di chiave minore.

CountingSort il caso generale

Se i numeri però sono chiavi di dati complessi, che vengono spostati con le chiavi, è necessario utilizzare un array ausiliario B per ricopiare i dati nell'ordine giusto e avere in C per ogni chiave il numero di elementi di chiave minore

Esempio:

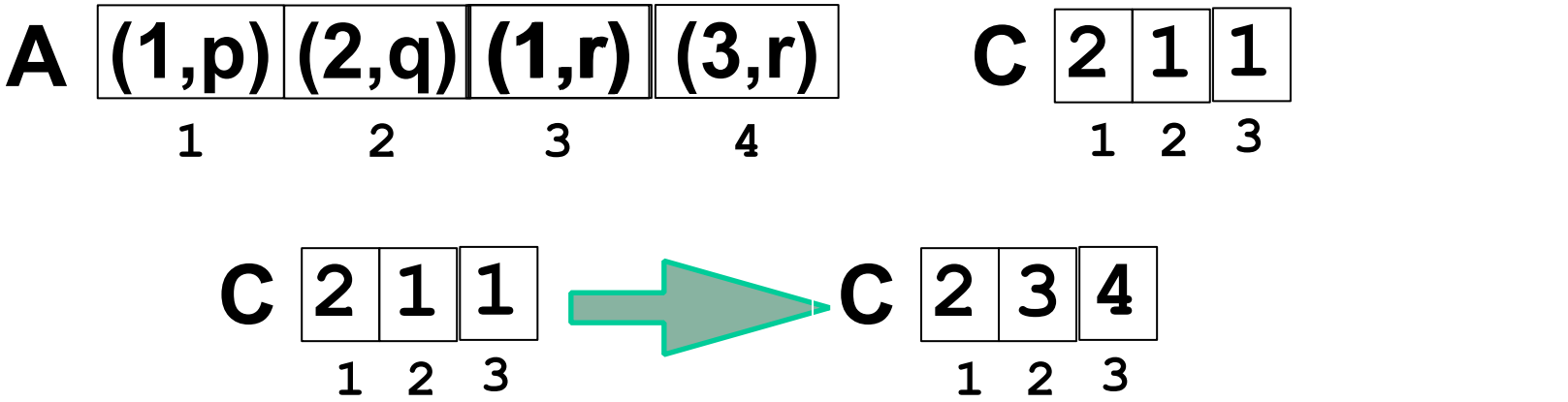




Per calcolare il numero degli elementi con chiave minore di i basta sommare C[i-1] a C[i], partendo con i = 1.

CountingSort il caso generale

Esempio:



CountingSort: modifica del passo 1

L'algoritmo viene descritto ignorando il link ai dati satellitari.
Nel primo passo nell'array ausiliario C[0..*k*] si calcola il rango, cioè il numero degli elementi minori o uguali, di ogni elemento in A.
Il rango fornisce la posizione che ogni elementi di A deve assumere nell'array ordinato.
Se C[i] contiene il numero delle occorrenze di i in A, per calcolare il rango di i basta sommare C[i-1] a C[i], partendo con i = 1.
In C[i] si avrà la posizione più a destra per i in A.

CountingSort1

Nel primo passo nell'array ausiliario C[0..*k*] si calcola il rango di ogni elemento in A.

Nel secondo passo quando si legge A[i], si cerca la posizione che A[i] deve assumere in B nell'array C, decrementandone il rango in C per poter inserire nella corretta posizione eventuali duplicati.

CountingSort1: passo 1

Se C[i] contiene il numero delle occorrenze di i in A, per calcolare il rango di i basta sommare C[i-1] a C[i], partendo da 1.
Quindi basta aggiungere questa operazione su C al passo 1 del CountingSort semplificato

```
for j = 1 to n do
    i = A[j]   A[j] è un numero compreso tra 0 e k
    C[i] = C[i] + 1
```

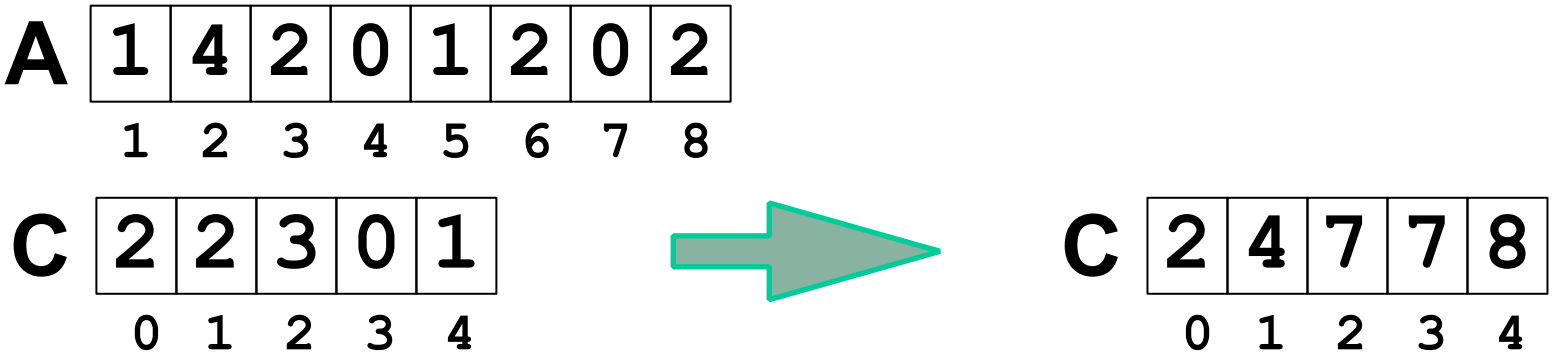
C[i] è il numero delle occorrenze di A[j]=i

```
for i = 1 to k do
      C[i] = C[i] + C[i-1]
```

C[i] è il numero di elementi minori o uguali a i

Esempio passo 1

Esempio: A contiene elementi in [0,4].



CountingSort1: secondo passo

Nel secondo passo si memorizzano nell'array ausiliario B gli elementi di A nell'ordine, per ogni elemento di A si vede in C qual'è la sua posizione e lo si sistema in B, si decrementa il valore in C per poter inserire eventuali duplicati.

```
for j = 1 to n do
```

in ogni passo di esecuzione k = A[j] e C[k] è il numero di elementi minori o uguali a k meno quelli uguali a k già copiati in B

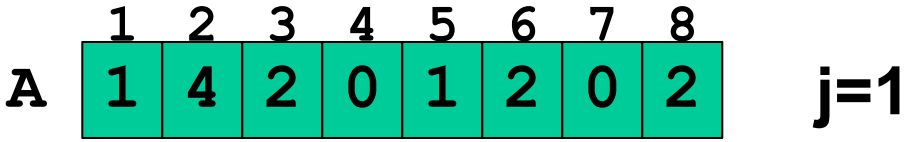
```
    m = A[j]
    i = C[m],
    B[i] = m
    C[m] = C[m] - 1
```

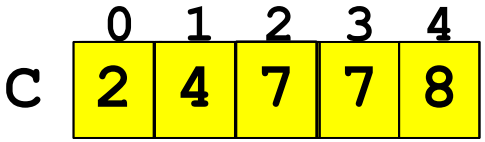
Secondo passo: esempio di esecuzione

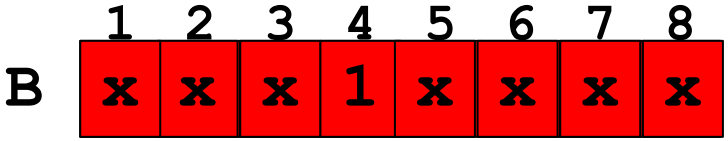
for j = 1 **to** n **do**

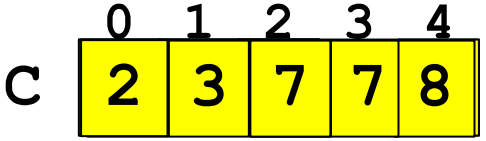
in ogni passo di esecuzione k = A[j] e C[k] è il numero di elementi minori o uguali a k meno quelli uguali a k già copiati in B

```
m = A[j]
i = C[m],
B[i] = m
C[m] = C[m] - 1
```









Secondo passo: esempio di esecuzione

for j = 1 to n do

in ogni passo di esecuzione k = A[j] e C[k] è il numero di elementi minori o uguali a k meno quelli uguali a k già copiati in B

```
m = A[j]
i = C[m],
B[i] = m
C[m] = C[m] - 1
```

	1	2	3	4	5	6	7	8
A	1	4	2	0	1	2	0	2

j=1 j=2 j=3 j=4

	0	1	2	3	4
C	1	3	6	7	7

	1	2	3	4	5	6	7	8
B	0	x	x	1	x	x	2	4

j=5

	1	2	3	4	5	6	7	8
B	0	x	1	1	x	x	2	4

	0	1	2	3	4
C	1	2	6	7	7

```
CountingSort1(A,B,k)
input: un array A di n elementi.
prec: gli elementi di A hanno chiavi intere in [0,k] per una costante k > 0
postc: restituisce un array B in cui gli elementi di A sono ordinati
for j=1 to n do
    i=A[j]  A[j] è un numero compreso tra 0 e k
    C[i]=C[i] + 1 C[i] è il numero delle occorrenze di A[j]=i
for i=1 to k do
     C[i]=C[i] + C[i-1] C[i] è il numero di elementi ≤ i
for j = 1 to n do
in ogni passo di esecuzione k = A[j] e C[k] è il numero di elementi ≤ k meno quelli uguali a k già copiati in B
      m = A[j]
      i = C[m],
      B[i] = m
      C[m] = C[m] - 1
```

CountingSort1(A,B,k)

input: un array A di n elementi.

prec: gli elementi di A hanno chiavi intere in [0,k] per una costante $k > 0$

postc: restituisce un array B in cui gli elementi di A sono ordinati

```
for j=1 to n do                                   Θ(n)
    i=A[j]  A[j] è un numero compreso tra 0 e k
    C[i]=C[i] + 1 C[i] è il numero delle occorrenze di A[j]=i
for i=1 to k do                                   Θ(k)
     C[i]=C[i] + C[i-1] C[i] è il numero di elementi ≤ i
for j = 1 to n do                                 Θ(n)
```

in ogni passo di esecuzione k = A[j] e C[k] è il numero di elementi ≤ k meno quelli uguali a k già copiati in B

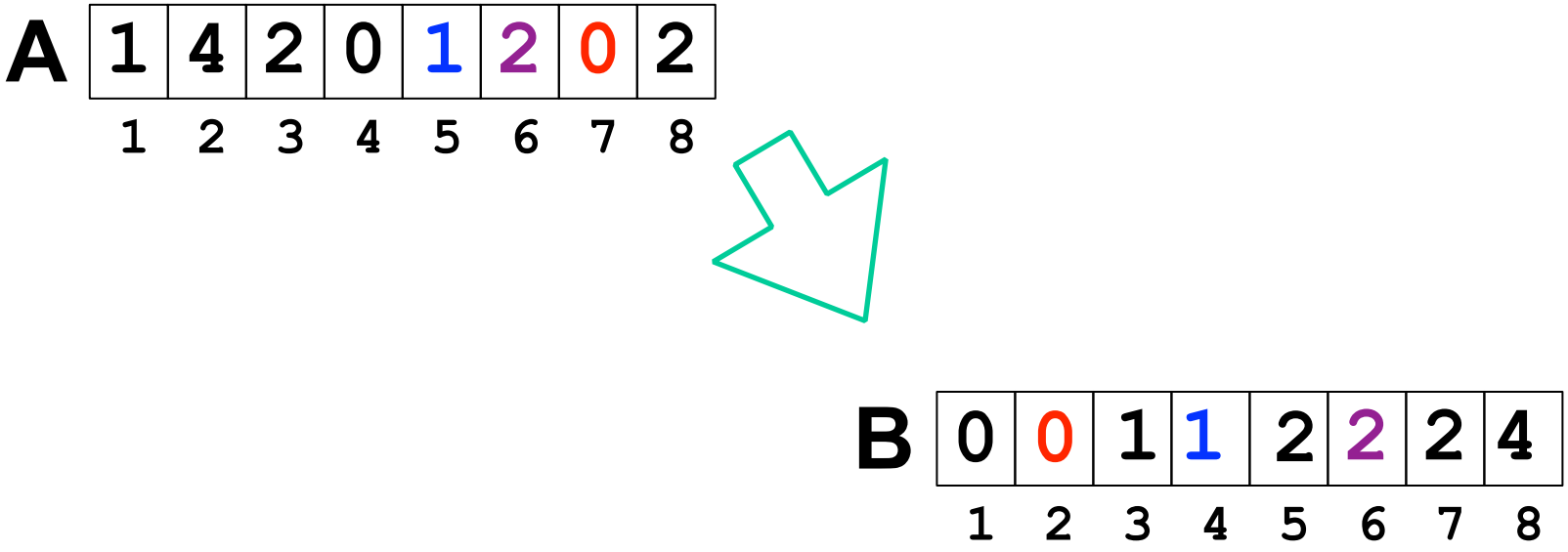
```
     m = A[j]
     i = C[m],
     B[i] = m
     C[m] = C[m] - 1
```

Totale $\Theta(n+k) = \Theta(n)$

Se k = O(n)

Stabilità

Un algoritmo di ordinamento è stabile se non cambia l'ordine relativo fra elementi uguali.



Stabilità: Esempio

voli	*ordinati per orario*	*ordinati per destinazioni (non stabile)*	*ordinati per destinazioni (stabile)*
	Londra 09:05:00	*Berlino 09:25:00*	*Berlino 09:25:00*
	Londra 09:20:00	*Berlino 10:00:00*	*Berlino 10:00:00*
	Berlino 09:25:00	*Londra 09:30:00*	*Londra 09:05:00*
	Londra 09:30:00	*Londra 09:05:00*	*Londra 09:20:00*
	Parigi 09:30:00	*Londra 09:20:00*	*Londra 09:30:00*
	Parigi 09:40:00	*Parigi 10:00:00*	*Parigi 09:30:00*
	Parigi 09:50:00	*Parigi 09:50:00*	*Parigi 09:50:00*
	Berlino 10:00:00	*Parigi 09:30:00*	*Parigi 10:00:00*

CountingSort1 non è stabile

…

for j = 1 **to** n **do**

in ogni passo di esecuzione k = A[j] e C[k] è il numero di elementi ≤ k meno quelli uguali a k già copiati in B

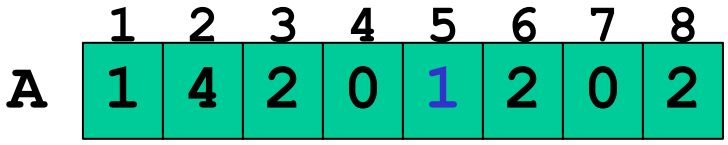
m = A[j]

i = C[m],

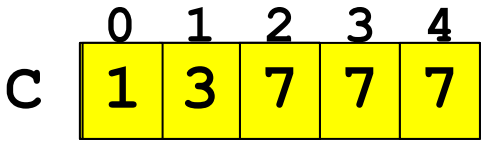
B[i] = m

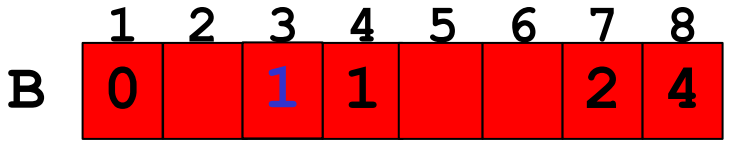
C[m] = C[m] - 1

Gli elementi uguali non sono nello stesso ordine relativo!



j=1 j=2 j=3 j=4 j=5



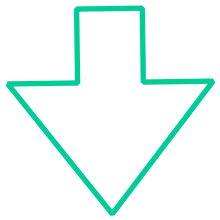


Come modificare il ciclo per la stabilità?

for j = 1 to n do

in ogni passo di esecuzione k = A[j] e C[k] è il numero di elementi ≤ k meno quelli uguali a k già copiati in B

```
m = A[j]
i = C[m],
B[i] = m
C[m] = C[m] - 1
```

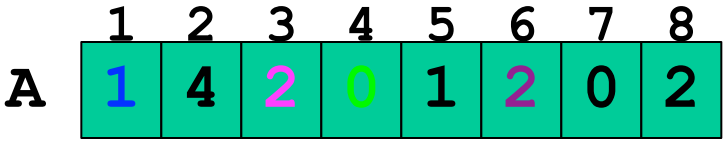


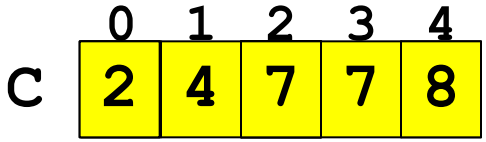
for j = n downto 1 do

C[A[j]] è il numero di elementi minori o uguali ad A[j] meno quelli uguali ad A[j] già copiati in B

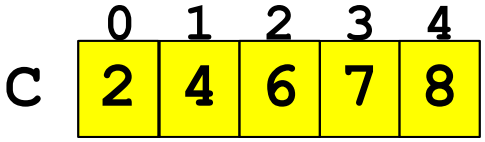
```
m = A[j]
i = C[m],
B[i] = m
C[m] = C[m] - 1
```

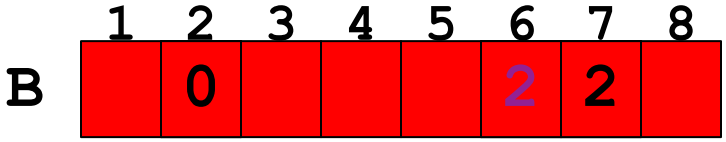
Esempio di esecuzione versione stabile





j=8 j=7 j=6





```
CountingSort(A,B,k)
input: due arrays A, B  di n elementi e una costante k > 0
prec: gli elementi di A hanno chiavi intere in [0,k]
postc: nell'array  B gli elementi di A ordinati
for j=1 to n do
    i=A[j]  A[j] è un numero compreso tra 0 e k
    C[i]=C[i] + 1 C[i] è il numero delle occorrenze di A[j]=i
for i=1 to k do
     C[i]=C[i] + C[i-1] C[i] è il numero di elementi ≤ i
for j=n downto 1 do
in ogni passo di esecuzione k = A[j] e C[k] è il numero di
elementi ≤ k meno quelli uguali a k già copiati in B
     m = A[j]
      i = C[m],
      B[i] = m
      C[m] = C[m] - 1
```

Complessità CountingSort.

```
CountingSort(A,B,k)
for j=1 to n do                          ——— Θ(n)
      i=A[j]
      C[i]=C[i] + 1
for i=1 to k do                          ——— Θ(k)
      C[i]=C[i] + C[i-1]
for j = n downto 1 do                    ——— Θ(n)
      m = A[j]
      i = C[m],
      B[i] = m
      C[m] = C[m] - 1
```

Complessità:
$T_{CS}(n,k) = \Theta(n+k)$

Se $k = O(n)$ allora $T_{CS}(n,k) = \Theta(n)$

Conclusione

il Counting Sort assume che gli elementi siano in [0,k] e si basa sull'array dei ranghi, il suo tempo di esecuzione è $\Theta(n+k)$, se n è il numero degli elementi.

Se k = O(n) allora è lineare in n e infatti è molto utile se k è relativamente piccolo rispetto a n.

Non ordina in loco né per confronti ed è stabile.

Esercizi

Quali degli ordinamenti studiati è stabile?

Trovate un modo semplice per rendere stabile qualsiasi algoritmo di ordinamento e stabilite quanto tempo e/o spazio richiede in più.

Descrivi un algoritmo che, dato un array A di n interi in [0,k], predispone A in modo che si possa sapere in tempo costante quanti valori di A sono nell'intervallo [a,b], per ogni $0 \le a \le b \le k$.

L'algoritmo dovrebbe avere tempo di esecuzione $\Theta(n + k)$